Spatio-temporal knowledge bases in a constraint logic programming framework with multiple theories

Alessandra Raffaetà

March 2000
Thesis Supervisors:
Prof. Paolo Mancarella and Prof. Franco Turini
Abstract

This thesis is aimed at designing a formalism that combines the ability to represent and reason on temporal and spatial data by means of high-level mechanisms, with the ability to integrate and transparently interact with different information sources. Such a formalism is obtained by amalgamating a repertoire of programming facilities coming from program composition approaches and constraint logic programming.

In detail, we define a language MuTACLP that joins the advantages of Temporal Annotated Constraint Logic Programming in handling temporal information, with the ability to structure and compose programs. The pieces of temporal information are given by temporal annotations whereas spatial data are represented by using constraints in the style of the constraint databases approach. MuTACLP retains the appealing feature of (constraint) logic programming of having a clear semantics. A top-down semantics, relying on a meta-interpreter, and a bottom-up semantics based on an immediate consequence operator are defined and proved to be equivalent.

To highlight the usefulness of the framework we present two applications. The first one consists in employing MuTACLP on top of Geographical Information Systems (GIS) in order to provide a more friendly interface for GIS analysis. The second application, which, in a certain sense, generalizes the former, consists in showing that MuTACLP can favor the construction of a software layer, the layer of mediators, that supplies the user with a declarative interaction with complex and, often, non declarative systems, like heterogeneous databases, GIS and the web.
To my grandmother Olga
Acknowledgments

I am in debt with many people who helped me during all these years. First of all, I am very grateful to my advisors Paolo Mancarella and Franco Turini for their friendly and patient guidance. They have always encouraged and supported me with affection and high competence.

A very stimulating experience has been to participate at the DeduGIS Working Group. I thank the members of this working group, and in particular Fosca Giannotti, Dino Pedreschi, Annalisa di Deo and Christine Parent, who have contributed to the development of this thesis. I cannot forget to thank Gianluca Nerbini for the work we have done together on the material that is included in Chapter 6. Furthermore, I am indebted to Elisa Bertino, Barbara Catania, Thom Frühwirth and Giorgio Ghelli, with whom it has always been a pleasure working.

Many thanks go to Antonio Brogi for his precious comments on a draft of the thesis and to Maria Simi who carefully followed the evolution of my work in the annual advancement reports. I would like also to acknowledge the reviewers of the thesis for their useful suggestions.

Then I express my gratitude to all my colleagues and friends at the Department of Computer Science in Pisa, with whom I have shared discussions, courses and indeed many many days and nights. First I mention my roommates: the “polemic” Andrea Bracciali, the “wise” Simone Contiero, the “kind” Chiara Renso and the “indefatigable” Salvatore Ruggieri. I would like also to remember Chiara Bodei, Roberta Gori, Paolo Volpe, Vincenzo Gervasi, Roberto Bruni, Marco Comini, Nadia Pisanti, Mercè Llabrés Segura, Francesca Scozzari, Emilio Tuosto, Stefano Bistarelli, whose friendship has greatly improved the quality of my staying at the department.

In Pisa during these years I have found a second family: Elisabetta Boldrini, Ilaria Cinelli, Francesca Di Cresce, Emanuela Garibaldi and Serena Tonelli. Special thanks to them, it was really nice to come back home and stay together in Via Paoli.

Finally, I would like to thank my family, who has always helped me and stimulated my interests in reading and discovering. Last but most important this thesis would not exist without my husband Paolo. His love and his constant support make everything easier and more pleasant.
# Contents

1 Introduction
   1.1 Main Contributions of the thesis ................................. 3
   1.2 Plan of the thesis .................................................. 5

2 Preliminaries ............................................................. 7
   2.1 Domain theory ....................................................... 7
   2.2 Logic Programming .................................................. 8
      2.2.1 Syntax and Terminology ....................................... 9
   2.3 Constraint Logic Programs ....................................... 12

3 Spatio-temporal Data models and Languages ....................... 15
   3.1 Time and temporal languages ..................................... 16
      3.1.1 Overview on Logical Temporal languages .................. 17
      3.1.2 Time in Constraint Databases .............................. 20
   3.2 Spatial data models and languages ................................ 21
      3.2.1 Space in Constraint Databases ............................. 26
   3.3 Spatio-temporal data models and languages ..................... 28

4 Operators for combining logic programs ............................. 33
   4.1 Syntax ..................................................................... 33
   4.2 Semantics .............................................................. 35
   4.3 Message passing ...................................................... 38

5 Time in a Multi-Theory Framework .................................... 41
   5.1 Time associated with clauses ...................................... 42
      5.1.1 Semantics ......................................................... 44
      5.1.2 An example ....................................................... 48
      5.1.3 Derived operators on time intervals ....................... 49
      5.1.4 Discussion ......................................................... 51
   5.2 Temporal Annotated Constraint Logic Programming ............. 52
      5.2.1 The language ..................................................... 53
      5.2.2 Semantics of TACLIP ........................................... 58
      5.2.3 Related Work ..................................................... 62
5.3 Multi-theory TACLp (MuTACLp) ........................................ 64
  5.3.1 MuTACLp language ............................................... 64
  5.3.2 Semantics of MuTACLp ......................................... 67
5.4 Application to legal reasoning ...................................... 72
5.5 From programs with annotated clauses to MuTACLp programs .............................. 77
5.6 Conclusion ............................................................ 81

6 Modeling spatial data in MuTACLp .................................. 85
  6.1 Introducing spatial information: the simple way ................. 86
  6.2 Declarative GIS analysis: a preliminary investigation ........ 90
    6.2.1 Integration with GIS by invoking GIS functions .......... 90
  6.3 Declarative GIS analysis in MuTACLp ............................ 94
    6.3.1 A logical representation of GIS spatial data ............. 94
    6.3.2 Operators on spatial objects ................................ 99
    6.3.3 Creation of GIS layers ..................................... 102
    6.3.4 A module for handling spatial data ....................... 103
  6.4 Some examples .................................................... 106
  6.5 Conclusion ........................................................ 111

7 Towards interoperability .............................................. 115
  7.1 STILan: Space, Time and Interoperability ...................... 116
    7.1.1 Mediators ............................................... 116
    7.1.2 Which language? ......................................... 117
  7.2 An example: the management of a library ...................... 120
  7.3 Conclusion ........................................................ 124

8 Conclusions .......................................................... 127

A Proofs of soundness and completeness ............................ 129
  A.1 Meta-interpreter for programs with annotated clauses ........ 129
    A.1.1 Soundness ................................................. 129
    A.1.2 Completeness ............................................. 132
  A.2 MuTACLp meta-interpreter ...................................... 135
    A.2.1 Soundness ................................................. 136
    A.2.2 Completeness ............................................. 141

Bibliography .......................................................... 153
List of Figures

3.1 A generalized temporal relation ........................................... 20
3.2 A temporal table and the global constraint $G$ ....................... 21
3.3 Some spatial objects. ......................................................... 23
3.4 Topological relations between two 2-dimensional objects .......... 25
3.5 Representation of a polygon with a hole .............................. 27
3.6 ST-complex ................................................................. 29
3.7 Parametric 2-spaghetti model. ........................................... 30

6.1 Scheme of the system. ....................................................... 93
6.2 The translation process from GIS to MuTACLIP. .................... 95
6.3 Some spatial objects. ....................................................... 97
6.4 Constraint of a half-plane ................................................ 98
6.5 Our region of interest. ..................................................... 100
6.6 Favorable habitat ........................................................... 107

7.1 A mediation architecture with wrappers. ............................. 116
7.2 Three layers of a mediator architecture ............................... 117
7.3 Architecture of the library mediator system. ........................ 121
Chapter 1

Introduction

The manipulation of complex data of very large size, such as spatial and temporal information, and often of very different nature, has become one of the challenges of today's research [112, 45, 117, 92, 20]. There are many applications that need the ability of storing and manipulating geometric and temporal data, such as geographic information systems (GIS), geometric modeling systems (CAD), and temporal databases. Actually, space and time are closely interconnected [78]: much information which is referenced to space is also referenced to time. For instance, cadastral data consist of sets of spatial objects (real estate) associated with other information about the owner, the category of the object and its estimated value. The boundary of real estates and the information associated with them can change during the years, due to the fact that ownership is affected by contracts, death and inheritance. Since taxes, census and other statistics are related to these data, it is required an information system containing historical spatial data and capable of reasoning on such data. Other important examples where spatio-temporal data are strongly required arise in scientific applications, such as geology and environmental monitoring. Traditional databases are not able to manage these complex data at a high level of abstraction. Actually, spatial and temporal data differ from conventional data in particular for the fact that the domains are interpreted and that they often model infinite objects. To fill the gap, many spatial database systems and temporal databases have been defined (e.g., [59, 117, 92, 56, 112, 45, 103, 34]). However, only recently the problem of dealing with correlated spatial and temporal data has been addressed and the existing models are not completely satisfactory, especially because they do not provide an explicit and flexible reasoning mechanism, whereas spatio-temporal information requires it much more than ordinary data.

Another important problem put in evidence by the wide use of Internet and the World Wide Web capabilities is the need for organizations to access and integrate different sources of information. By interoperability we mean making information systems work together [115]. Data sharing between different applications and different sites is often the preferable mode of interoperation. But sharing of data and application programs developed around it, facilitated by the advances in network
technology, is hampered by the incompatibility of different data models and formats used at different sites. Semantically identical data may be structured in different ways. We distinguish interoperability at two levels of abstraction: *syntactic* (low) level and *semantic* (high) level. In the context of syntactic interoperability, the “meaning” of terms is irrelevant. Examples of this kind of interoperation are different Unix systems which work together or cooperating SQL systems of different vendors. On the other hand, semantic interoperability tries to solve problems deriving from the use of different data models, of schemas which do not match, from missing or inconsistent data and so on. A further distinction is between *data* and *query* semantic interoperability [37, 38]. In the former, the data of a database are translated into a different model, making direct data sharing possible, whereas the latter concerns the translation of queries, enabling the users of a database to request the evaluation of queries within another one. In this case data sharing is indirect.

In this context, the general objective of our project is the design of a formalism which fulfills the following requirements: the ability to represent temporal and spatial data and to express spatio-temporal correlations by means of high-level mechanisms, and the ability to integrate and transparently interact with heterogeneous information sources.

In order to define the desired formalism, we amalgamate a repertoire of programming facilities, coming from program composition approaches and constraint logic programming.

Most modern computing systems consist of large numbers of software components that interact with one another. Correspondingly, the process of software development is more and more focusing on methodologies for composing generic packages to build complex applications. The programming activity is hence changing from writing large pieces of software in some programming language into writing small pieces of “gluing” software for putting existing components at work together.

The logic programming paradigm has demonstrated to be very well-suited for supporting program composition approaches [31, 25, 97]. In this thesis, we adopt the approach proposed by Broggi et al. [23, 25], which is centered on the definition of a family of meta-level composition operations over named definite logic programs. From a programming perspective, the introduction of the operations has been shown to enhance the expressive power and knowledge representation capabilities of logic programming, and in [13, 97] such operations are used to support query semantic interoperability. Thus this multi-theory framework partially fulfills our requirements but it completely lacks of mechanisms to represent spatial and temporal information. The problem is that logic programming seems not to be adequate to tackle the complexity of spatial data, hence the idea is to extend the multi-theory framework by considering a more powerful language, based on Constraint Logic Programming, to represent knowledge, i.e., to describe object level programs.

Constraint Logic Programming (CLP) [64, 65, 66] is a merger of two declarative paradigms: constraint solving and logic programming. This combination helps making CLP programs both expressive and flexible, and in some cases, more efficient
than other kinds of programs. The CLP scheme extends traditional logic programming by generalizing the term equations of logic programming to constraints from any pre-defined computation domain. The CLP Scheme defines a family of languages, CLP($\mathcal{C}$), which are parametric in the constraint domain $\mathcal{C}$, and which share rich operational, denotational and logical semantic theories [66], thus preserving one of the most appealing aspects of logic programming.

The use of constraints offers many advantages. At the data level, constraints are able to finitely represent possibly infinite sets. In particular, they are shown to be a powerful mechanism for modeling spatial and temporal concepts, where often infinite sets have to be represented [69, 93, 71, 56, 37, 16]. For instance the constraint $0 \leq x \leq 3 \land 0 \leq y \leq 3$ represents the square area with corners (0,0), (0,3), (3,0) and (3,3) and 1983 $\leq t \leq$ 1997 represents the interval of time between 1983 and 1997. With respect to data modeling, the main advantage is that constraints serve as a unifying data type for the (conceptual) representation of heterogeneous data and, by varying the constraint theory, one can accommodate a variety of different data models. At the query language level, constraints increase the expressive power of logic programming by allowing mathematical computations.

Moreover, it is worth noting that the meaning of constraints is defined by a constraint theory whose reasoning capability is implemented by some efficient algorithm in the so-called constraint solver. In this way, efficient special-purpose algorithms can be integrated in a sound way into logic programming. To allow for more flexibility and application-oriented customization of constraint systems, now several systems offer constructs for writing constraint solvers. We recall Constraint Handling Rules, proposed by Frühwirth [47, 51], which are a declarative language extension especially designed for writing constraint solvers. They are essentially a committed-choice language consisting of guarded rules that rewrite constraints into simpler ones until they are solved.

1.1 Main Contributions of the thesis

The main achievement of the thesis is the definition of a framework where temporal and spatial information can be represented and handled, and, at the same time, knowledge can be separated into different theories and combined by means of meta-level composition operations.

Our framework is multi-theory and the basic components are TACLCP programs. Temporal Annotated Constraint Logic Programming (TACLCP) [48, 49, 50] combines annotated logics [70] with constraint logic programming concepts [64, 65, 66], and it has been shown to be a natural and powerful way of formalizing temporal information and reasoning. In particular, the pieces of temporal information are given by temporal annotations which say at what time(s) the formula to which they are attached is valid. The advantage of having labels is well summarized by what D. Gabbay says in the introduction of his book on labelled deductive systems [53],
that allow to label a general class of logics, “This sounds very simple but it ... makes a serious difference, like the difference between using one hand only or allowing for the coordinated use of two hands”.

On the other hand, spatial data are represented by using constraints in the style of the constraint databases approaches [69, 93, 92, 16, 56, 57, 37, 38, 73] and, to the best of our knowledge, it is the first time that TACLp is shown to be able to express both spatial and temporal data. The facilities offered by the framework to handle time allows one to easily establish spatio-temporal correlations, for instance time-varying areas, or, more generally, moving objects, supporting either discrete or continuous change.

Moreover, the language, Multi-theory Temporal Annotated Constraint Logic programming (MuTACLp), provided by the framework, retains the appealing aspects of (constraint) logic programming of having a well-founded definition of alternative equivalent semantics, more precisely a top-down semantics via a meta-interpreter and a bottom-up semantics based on an immediate consequence operator.

To highlight the usefulness of the framework we present two applications.

The first one consists in employing MuTACLp on top of Geographical Information Systems (GIS) in order to provide a more friendly interface for GIS analysis. Actually, one of the more frequently stated problems for GIS is that these systems have a complex functionality which is not accessible to non expert end-users. Today GIS user interfaces are not easy to use and require much time to get used to them [117, 39, 94]. For instance, as pointed out by Egenhofer in [39], to solve spatial problems, the primary operations offered by current GIS user interfaces include the selection of data layers, the identification of objects by location, name and elementary spatial relations, and the modification of graphical output parameters such as color and patterns.

Adding a declarative programming layer on top of GIS can allow a better use of them, essentially because a declarative approach is much closer to the natural ways of expressing analysis rules than a procedural approach. Furthermore, since MuTACLp allows the representation and the handling of temporal information, this layer can be a means to introduce temporal information in GIS, for instance by relating time to the data stored in GIS via MuTACLp rules. This is a great advantage because time is almost completely ignored in current GIS, even if it is recognized to be an essential component of geographical information [78, 117]. Finally, our multi-theory framework, that provides tools for combining different knowledge components encoded in distinct programs, seems particularly suited to handle the naturally fragmented knowledge used in GIS analysis.

The second application, which, in a certain sense, generalizes the former, consists in showing that MuTACLp can favor the construction of a software layer, the layer of mediators, that supplies the user with a declarative interaction with complex and, almost always, non declarative systems, like heterogeneous databases, the web, GIS etc.
1.2 Plan of the thesis

In Chapter 2 some basic mathematical concepts from domain theory which we will largely use in the thesis are presented, and some background material on the logic programming paradigm and constraint logic programming scheme is reported. Furthermore the notation adopted throughout the thesis is fixed.

In Chapter 3 we briefly recall the basic concepts of the modeling of temporal and spatial data and we describe some approaches in literature that are related to how we handle temporal and spatial information.

Chapter 4 is devoted to introduce the program composition approach by Brogi, Mancarella, Pedreschi and Turini, which is the starting point of our research.

Chapter 5 contains the description of our framework for temporal reasoning. The
resulting language, called MuTACL, joins the advantages of Temporal Annotated Constraint Logic Programming in handling temporal information with the ability to structure and compose logic programs coming from the program composition approach by Brogi, Mancarella, Pedreschi and Turini.

Chapter 6 shows how we can model spatial data in MuTACL, thus obtaining a language to represent and reason on spatio-temporal data and capable of establishing spatio-temporal correlations. Then we present the first application: employing MuTACL on top of Geographical Information Systems in order to provide a more friendly interface to make GIS analysis.

Chapter 7 describes the completion of our proposal of a language able to express spatio-temporal correlations and to interact with heterogeneous sources of data. This is the second application: MuTACL, augmented with a message passing mechanism, is used to define mediators.

Finally, Chapter 8 contains some concluding remarks. The appendix collects the proofs of the theorems of soundness and completeness for the languages presented in the thesis.
Chapter 2

Preliminaries

2.1 Domain theory

In this section we recall some basic mathematical concepts from domain theory which we will largely use in the thesis.

Partial Orders, Complete Partial Orders and Lattices

A binary relation $\sqsubseteq$ on a set $S$ is a partial order if, for each $x, y, z \in S$

\[
    x \sqsubseteq x \quad \text{(reflexivity)}
\]

\[
    x \sqsubseteq y, y \sqsubseteq x \Rightarrow x = y \quad \text{(antisymmetry)}
\]

\[
    x \sqsubseteq y, y \sqsubseteq z \Rightarrow x \sqsubseteq z \quad \text{(transitivity)}
\]

A partially ordered set (poset) $(S, \sqsubseteq)$ is a set $S$ endowed with a partial order $\sqsubseteq$. A set $S$ is totally ordered if it is partially ordered and, for each $x, y \in S$, $x \sqsubseteq y$ or $y \sqsubseteq x$.

Given a poset $(S, \sqsubseteq)$ and a subset $X \subseteq S$, $y \in S$ is an upper bound of $S$ if and only if for each $x \in X$, $x \sqsubseteq y$. Moreover, $y \in S$ is the least upper bound of $X$ if and only if $y$ is an upper bound of $X$ and, for all upper bounds $y'$ of $X$, $y \sqsubseteq y'$. When a subset $X$ of a poset has a least upper bound it is denoted by $\bigsqcup X$. We also write $\bigsqcup \{d_1, \ldots, d_n\}$ as $d_1 \sqcup \ldots \sqcup d_n$. Dually an element $y \in S$ is a lower bound for $X$ if and only if for each $x \in X$, $y \sqsubseteq x$. Moreover, $y \in S$ is the greatest lower bound of $X$ if and only if $y$ is a lower bound of $X$ and, for all lower bounds $y'$ of $X$, $y \sqsubseteq y'$. When a subset $X$ of a poset has a greatest lower bound it is denoted by $\bigsqcap X$. We also write $\bigsqcap \{d_1, \ldots, d_n\}$ as $d_1 \sqcap \ldots \sqcap d_n$.

An $\omega$-chain of a poset $(S, \sqsubseteq)$ is an increasing chain $d_0 \sqsubseteq d_1 \sqsubseteq \ldots \sqsubseteq d_n \sqsubseteq \ldots$ of elements of the set.

A complete partial order (cpo) is a poset $(S, \sqsubseteq)$ such that every $\omega$-chain $D$ has the least upper bound (i.e., there exists $\bigsqcup D$). We say that $(S, \sqsubseteq)$ is a cpo with bottom
A complete lattice is a poset $(S, \subseteq)$ such that for every subset $X$ of $S$ there exists $\bigsqcup X$. Let $\top$ denote the top element $\bigcup S = \bigsqcup \emptyset$ and $\bot$ denote the bottom element $\bigcap S = \bigcap \emptyset$ of $S$. It is easy to check that, for any set $S$, the powerset of $S$, denoted by $\mathcal{P}(S)$, equipped with the subset ordering $\subseteq$, is a complete lattice, where $\sqcup$ is union, $\sqcap$ is intersection, the top element is $S$ and the bottom element is $\emptyset$.

### Continuous functions and the Fixpoint Theorem

A function $f : D \to E$ between cpos $(D, \sqsubseteq)$ and $(E, \leq)$ is monotonic if and only if

$$\forall x, y \in D. \ x \sqsubseteq y \implies f(x) \leq f(y)$$

Moreover, $f$ is continuous if and only if, it is monotonic and for all $\omega$-chains $d_0 \sqsubseteq d_1 \sqsubseteq \ldots \sqsubseteq d_n \sqsubseteq \ldots$ in $D$ we have

$$\bigcup_{i \in \mathbb{N}} f(d_i) = f \left( \bigcup_{i \in \mathbb{N}} d_i \right)$$

Given a continuous function $f : D \to D$ on a cpo $(D, \sqsubseteq)$, a fixpoint of $f$ is an element $d \in D$ such that $f(d) = d$. A pre-fixpoint of $f$ is an element $d \in D$ such that $f(d) \sqsubseteq d$.

The following important theorem gives an explicit construction of the least fixpoint of a continuous function $f$ on a cpo $D$.

**Theorem 1 (Fixpoint Theorem)** Let $f : D \to D$ be a continuous function on a cpo with bottom $\bot$. Define

$$f^\omega = \bigsqcup_{i \in \mathbb{N}} f^i(\bot)$$

where $f^0(\bot) = \bot$ and $f^{i+1}(\bot) = f(f^i(\bot))$. Then $f^\omega$ is the least pre-fixpoint and least fixpoint of $f$.

### 2.2 Logic Programming

We briefly introduce the standard syntax and terminology of logic programming as well as some fundamental results. Self-contained introductions to logic programming can be found in Lloyd’s book [81], in Apt’s contribution to the *Handbook of Theoretical Computer Science* [7] and in Apt’s book [8].
2.2.1 Syntax and Terminology

A **signature** is a pair \( \langle \Sigma_L, \Pi_L \rangle \) of sets of **function** and **predicate** symbols. To each function symbol a non-negative arity is assigned. In particular, 0-ary function symbols are called **constants**.

A **(first order) alphabet** consists of a signature, of an infinite and fixed set of **variables**, the connectives \( \lnot, \land, \lor, \rightarrow, \leftarrow, \leftrightarrow \), the quantifiers \( \forall \) and \( \exists \), and the punctuation symbols \( "(" "\)" \) and \( "." \). Variables are denoted by identifiers starting with an upper case letter.

The sets of **terms**, **atoms**, **literals**, **clauses** and **well-formed formulae** on an alphabet are defined in the standard way. Terms, atoms, etc. with no variables are called **ground**. Ground terms are denoted by identifiers starting with a lower case letter.

The (first order) language given by an alphabet consists of the set of all well-formed formulae on the alphabet. Therefore, once fixed the set of variables, first order languages are determined by their signature. In this sense, we confuse language and signature.

Let \( L \) be a language. We denote by \( \mathcal{U}_L \) the set of ground terms on \( L \), and call it the **Herbrand Universe** of \( L \) and by \( \mathcal{B}_L \) the set of ground atoms on \( L \), and call it **Herbrand Base** of \( L \).

**Queries and Programs**

A **query** is a finite sequence of atoms \( A_1, \ldots, A_n \) \((n \geq 1)\).

A **logic program** (or, simply, a program) is a finite set of **definite Horn clauses**, that is clauses which contain exactly one positive literal. In logic programming notation, clauses are written in the form

\[
A \leftarrow B_1, \ldots, B_n
\]

where \( n \geq 0 \) and \( A \) and \( B_1, \ldots, B_n \) are atoms. \( A \) is called the clause **head**, while \( B_1, \ldots, B_n \) the clause **body**. If \( n = 0 \) the clause is called **unit clause** and is also written in the form \( A \). The symbols \( "\)" and \( "\leftarrow\) are used to denote conjunction and reverse implication, respectively. All the variables in a clause are universally quantified. We denote programs by \( P, V, \ldots \).

We write \( A \leftarrow B_1, \ldots, B_n \in \text{ground}_L(P) \) if and only if \( A \leftarrow B_1, \ldots, B_n \) is a ground instance of a clause from \( P \).

Throughout this thesis, we consider a fixed language \( L \) in which programs and queries are written.

**Semantics**

The semantics of a logic program can be formalized according to three different, albeit equivalent, approaches: model-theoretic, fixpoint and operational. We will use the standard notions of Herbrand interpretation, Herbrand model and immediate consequence operator [81].
A *Herbrand interpretation* is any subset of the Herbrand base \( \mathcal{B}_L \). A *Herbrand model* for a program \( P \) is a Herbrand interpretation which is also a model for \( P \).

Let us recall the definition of the *immediate consequence operator* \( T_P \) of logic programming, which will be largely used in this thesis. Given a program \( P \) and a Herbrand interpretation \( I \), the immediate consequence operator \( T_P \) maps Herbrand interpretations into Herbrand interpretations and it is defined as follows:

\[
T_P(I) = \{ A \mid A \leftarrow B_1, \ldots, B_n \in \text{ground}_L(P) \land \{B_1, \ldots, B_n\} \subseteq I \}
\]

The immediate consequence operator \( T_P \) is a continuous mapping from \( \wp(\mathcal{B}_L) \) to \( \wp(\mathcal{B}_L) \) [113]. The powerset \( \wp(\mathcal{B}_L) \) is a complete lattice under set inclusion where \( \emptyset \) is the bottom element and \( \mathcal{B}_L \) is the top element. Therefore, for any program \( P \), the least fixpoint of \( T_P \) is the union of the finite powers of \( T_P \) applied to the bottom element of \( \wp(\mathcal{B}_L) \), i.e., \( T_P^* = \bigcup_{n \in \mathbb{N}} T_P^n \).

Finally, we recall the characterization theorem stating the equivalence of model-theoretic, fixpoint and operational semantics of logic programming.

**Theorem 2** Let \( P \) be a logic program. Then \( P \) has a model \( M \) which satisfies the following properties:

1. \( M \) is the least Herbrand model of \( P \).
2. \( M \) is the least fixpoint of \( T_P \).
3. \( M = T_P^\omega \)
4. \( M \) is the set of ground atoms that have a SLD-refutation.

**The vanilla meta-interpreter**

A *meta-interpreter* for a language is an interpreter for the language written in the language itself. The ability to write a meta-interpreter easily is a very powerful feature a programming language may have. The vanilla meta-interpreter presented in [109] is the simplest application of meta-programming in logic. A general formulation of the vanilla meta-interpreter can be given by means of the \( \text{demo} \) predicate used to represent provability. Namely, \( \text{demo}(G) \) means that the formula \( G \) is provable in the object program.

\[
\text{demo}(\text{empty}). \hspace{1.5cm} (2.1)
\]
\[
\text{demo}((G_1, G_2)) \leftarrow \text{demo}(G_1), \text{demo}(G_2) \hspace{1.5cm} (2.2)
\]
\[
\text{demo}(A) \leftarrow \text{clause}(A, G), \text{demo}(G) \hspace{1.5cm} (2.3)
\]

The unit clause states that the empty goal, represented by the constant symbol \( \text{empty} \), is always solved. The second clause deals with conjunctive goals. It states that a conjunction \((G_1, G_2)\) is solved if \( G_1 \) is solved and \( G_2 \) is solved. Finally, the
third clause deals with the case of atomic goal reduction. To solve an atomic goal
A, a clause from the program is chosen whose head unifies with A and the body
of the clause is recursively solved. An object level program P is represented at the
meta-level by a set of axioms of the kind clause(A, G), one for each object level
clause A ← G in P. For example, the logic program

\[
 \text{nats}.
\]

\[
 \text{nats}((s(N))) ← \text{nats}(N)
\]

is represented at the meta-level by the clauses

\[
 \text{demo} (\text{nats} (\text{zero}), \text{empty}).
\]

\[
 \text{demo} (\text{nats} (s(N)), \text{nats}(N)).
\]

**Semantics of Vanilla Meta-programs**

The semantics of meta-logic is a quite debated issue (see e.g [61, 74, 87, 6]) and it
is however outside the scope of the thesis. One of the crucial problems is how to
represent object level terms at the meta-level. Two basic alternative representations
are employed: in the ground representation object level expressions are represented
by ground meta-level terms whereas in the non-ground representation they are rep-
resented by non-ground terms. In particular, an object level variable is represented
by a ground meta-level term in the first case, and by a meta-level variable in the
latter case, as in the vanilla meta-interpreter. We refer to the cited papers for a
more detailed comparison between the two approaches.

In this thesis, we will introduce extended vanilla meta-interpreters using the non-
ground representation of object level terms. The advantage of this representation
is that it is both simpler and more efficient than the other approach, thanks to the
fact that it directly exploits the basic unification mechanisms of the meta-language.

In the spirit of [28, 87], we will define the semantics of the extended vanilla meta-
interpreter by relating the semantics of an object program to the semantics of the
corresponding vanilla meta-program (i.e., including the meta-level representation
of the object program). When stating the correspondence between the object program
and the meta-program we consider only formulae of interest. For instance, in the
case of the vanilla meta-interpreter presented in this section, this corresponds to
showing that for any object level program P and for any object level ground atom
A, A is provable in the object program P if and only if demo(A) is provable in the
corresponding meta-program V, that is

\[
 \text{demo}(A) \in T_V^\omega \iff A \in T_P^\omega
\]

where V is the meta-program containing the meta-level representation of the object
program P and clauses (2.1)—(2.3).
2.3 Constraint Logic Programs

The Constraint Logic Programming (CLP) Scheme was introduced by Jaffar and Lassèz [64]. It merges logic programming with constraint solving, by generalizing the term equations of logic programming to constraints from any pre-defined computation domain. The CLP Scheme defines a family of languages, CLP(C), which are parametric in the constraint domain C, and which share rich operational, denotational and logical semantic theories.

The survey of Jaffar and Maher [65] provides a comprehensive introduction to motivations, foundations, and applications of CLP languages. A formal presentation of the semantics can be found in a recent work of Jaffar et al. [66].

Constraint Domains

A constraint domain C is a tuple \( \langle S_C, L_C, D_C, T_C, \text{solv}_C \rangle \).

\( S_C = \langle \Sigma_C, \Pi_C \rangle \) is the constraint domain signature. The class of constraints, \( L_C \), is a set of first-order \( S_C \) formulae. We denote constraints by \( c, d \), etc. The domain of computation, \( D_C \), is a \( S_C \)-structure that is the intended interpretation of the constraints. \( D_C \) is the domain (or support) of \( D_C \). The constraint theory, \( T_C \), is a \( S_C \) theory describing the logical semantics of the constraints. The constraint solver, \( \text{solv}_C \), is a (computable) function which maps each formula in \( L_C \) to one of \text{true}, \text{false}, \text{unknown}, indicating if the formula is satisfiable, unsatisfiable or it cannot be told.

We assume that the predicate symbol "=" is in \( \Pi_C \) and that it is interpreted as identity in \( D_C \). Also, we assume that \( L_C \) contains all atoms constructed from "=" , the always satisfiable constraint \text{true} and the unsatisfiable constraint \text{false}, and that \( L_C \) is closed under variable renaming, existential quantification and conjunction. A primitive constraint is an atom of the form \( p(t_1, \ldots, t_n) \) where \( p \) is a predicate in \( \Pi_C \) and \( t_1, \ldots, t_n \) are terms on \( \Sigma_C \). The smallest set of constraints which satisfies the above assumptions and contains all primitive constraints is called the set of constraints generated by primitive constraints.

We assume that the solver does not take variable names into account. Also, the domain, the theory and the solver agree in the sense that \( D_C \) is a model of \( T_C \) and for every \( c \in L_C \):

- \( \text{solv}_C(c) = \text{true} \) implies \( T_C \models \exists x, \) and
- \( \text{solv}_C(c) = \text{false} \) implies \( T_C \models \neg \exists x. \)

Example 1 (REAL) The constraint domain Real has \( <, \leq, =, \geq, > \) as relation symbols; \( +, -, \ast, / \) as function symbols; and sequences of digits (possibly with a decimal point) as constant symbols. Examples of primitive constraints are the following: \( x + 3 \leq y * 1.1, x/2 > 10 \). The domain of computation is the structure with
2.3. Constraint Logic Programs

reals as domain, and where the predicate symbols <, ≤, =, ≥, > and the function symbols +, −, *, / are interpreted as the usual relations and functions over reals. Finally, the theory $T_{\text{Real}}$ is the theory of real closed fields.

A possible constraint solver is provided by the CLP($\mathbb{R}$) system [67], which relies on Gauss-Jordan elimination to handle linear constraints. Non-linear constraints are not taken into account by the solver (i.e., their evaluation is delayed) until they become linear.

Example 2 (Logic Programming) The constraint domain Term has $=$ as relation symbol and strings of alphanumeric characters as function or constant symbols. The domain of computation of Term is the set of finite trees (or, equivalently, of finite terms), Tree, while the theory $T_{\text{Term}}$ is Clark’s equality theory.

The interpretation of a constant is a tree with a single node labeled with the constant. The interpretation of an $n$-ary function symbol $f$ is the function $f_{\text{Tree}} : \text{Tree}^n \rightarrow \text{Tree}$ mapping the trees $t_1, \ldots, t_n$ to a new tree with root labeled with $f$ and with $t_1, \ldots, t_n$ as children.

A constraint solver is provided by the unification algorithm. CLP(Term) coincides then with logic programming.

CLP Programs

A CLP language is determined by its constraint domain. For a particular constraint domain $C$, we denote by CLP($C$) the CLP language based on $C$.

As in the case of logic programs, our results are parametric to a language $L$ in which all programs and queries under consideration are included. $\Sigma_L$ coincides with the set of function symbols of $\mathcal{L}_C$, while $\Pi_L$ includes the predicate symbols of $\mathcal{L}_C$.

A constraint logic program, or program, is a finite set of rules of the form:

$$A \leftarrow C_1, \ldots, C_n, B_1, \ldots, B_m$$

where $A$ is an atom, called the head, $B_1, \ldots, B_m$ ($m \geq 0$) are atoms (whose predicate symbols are in $L$ but not in $\Pi_C$), and $C_1, \ldots, C_n$ ($n \geq 0$) are primitive constraints\(^1\). If $m = 0$, then the clause is called a fact. A query is a sequence of atoms and/or constraints.

Interpretations and Fixpoints

A $C$-interpretation for a CLP($C$) program is an interpretation which agrees with $D_C$ on the interpretations of the symbols in $\mathcal{L}_C$. Since the meaning of primitive constraints is fixed by $C$, we represent a $C$-interpretation $I$ by a subset of the $C$-base, written $C\text{-base}_L$, which is the set:

$$\{ p(d_1, \ldots, d_n) \mid p \text{ predicate in } \Pi_L \setminus \Pi_C, d_1, \ldots, d_n \in D_C \}.$$  

\(^1\)Constraints and atoms can be in any position inside the body of the clause.
The definitions of $\mathcal{C}$-models, and least $\mathcal{C}$-model are natural extensions of the logic programming concepts.

A valuation $\sigma$ is a function that maps variables into $D_\mathcal{C}$. A $\mathcal{C}$-ground instance $A'$ of a literal $A$ is obtained by applying a valuation $\sigma$ to the literal, thus producing a construct of the form $p(a_1, \ldots, a_n)$ with $a_1, \ldots, a_n$ elements from $D_\mathcal{C}$. Similarly $\mathcal{C}$-ground instances are defined for queries and clauses.

We denote by $\text{ground}_c(P)$ the set of $\mathcal{C}$-ground instances of clauses from $P$.

Finally the immediate consequence operator for a CLP($\mathcal{C}$) program $P$ is a function $T^c_P : \wp(\mathcal{C}-\text{base}_L) \rightarrow \wp(\mathcal{C}-\text{base}_L)$ defined as follows:

$$T^c_P(I) = \left\{ A \mid \begin{array}{l}
A \leftarrow C_1, \ldots, C_k, B_1, \ldots, B_n, \in \text{ground}_c(P), \\
\{B_1, \ldots, B_n\} \subseteq I, \quad D_\mathcal{C} \models C_1, \ldots, C_k
\end{array} \right\}$$

The operator $T^c_P$ is continuous, and therefore it has a least fixpoint which can be computed as the least upper bound of the $\omega$-chain $\{(T^c_P)^i\}_{i \in \mathbb{N}}$ of the iterated applications of $T^c_P$ starting from the empty set, i.e., $(T^c_P)^\omega = \bigcup_{i \in \mathbb{N}} (T^c_P)^i$. 
Chapter 3
Spatio-temporal Data models and Languages

In this chapter, we briefly recall the basic concepts of the modeling of temporal and spatial data, and we describe some approaches in literature that are related to how we handle temporal and spatial information.

Before going on with the presentation, we introduce constraint databases, a promising paradigm to model spatial and temporal information. We focus our attention on such kind of databases because they share with the language proposed in this thesis many ideas for the treatment of spatial data (see Section 6).

Constraint Databases
Constraint Databases [68, 69] generalize the classical relational model of data by introducing generalized tuples: quantifier-free formulae in an appropriate constraint theory. The key intuition of this approach is that the generalization of a tuple is a conjunction of constraints.

Let $\Phi$ be a class of constraints.

- A generalized $k$-tuple over variables $x_1, \ldots, x_k$ is a finite conjunction $\varphi_1 \land \ldots \land \varphi_N$, where each $\varphi_i, 1 \leq i \leq N$, is a constraint in $\Phi$. The variables in each $\varphi_i$ are all free and among $x_1, \ldots, x_k$.

- A generalized relation $r$ of arity $k$ is a finite set $r = \{\psi_1, \ldots, \psi_M\}$ where each $\psi_i, 1 \leq i \leq M$, is a generalized tuple over the same variables $x_1, \ldots, x_k$.

- The formula corresponding to a generalized relation $r$ is the disjunction $\psi_1 \lor \ldots \lor \psi_M$. We use $\phi_r$ to denote such a quantifier-free formula.

- A generalized database is a finite set of generalized relations.

The term unrestricted relation will be used for finite or infinite sets of points in a $k$-dimensional space. Notice that, in database theory, a $k$-ary relation $r$ is only
a finite set of $k$-tuples (or points in a $k$-dimensional space) and a database is a finite set of relations. In order to be able to manage unrestricted relations, a finite representation is needed, and this is exactly what the generalized tuples provide.

Let $\Phi$ be a class of constraints interpreted over the domain $D$, $r$ a generalized relation, and $\phi_r = \phi_r(x_1, \ldots, x_k)$ its corresponding formula with free variables $x_1, \ldots, x_k$. The generalized relation $r$ represents the unrestricted $k$-ary relation which consists of all $(a_1, \ldots, a_k)$ in $D^k$ such that $\phi_r(a_1, \ldots, a_k)$ is true. A generalized database represents the finite set of unrestricted relations that are represented by its generalized relations.

### 3.1 Time and temporal languages

In this section we introduce the glossary we are going to use when dealing with time. In particular we list the alternatives choices for the structure of time, temporal relations, and the different kinds of time that are usually modeled. We refer the reader to *The Consensus Glossary of Temporal Database* [45] for a complete collection of definitions of concepts related to time.

Then we present an overview of logic-based temporal languages and Koubarakis’ approach [71] for the handling of time in constraint databases.

**Structure of time**

- **Instants vs. time intervals.** An instant is a time point on an underlying time axis, while a time interval is the time between two instants. Correspondingly we can distinguish two view points according to the granularity of time: point-based and interval-based. In the database context the point-based view is predominant (e.g., see [112, 34]), on the other hand, there is a lot of Artificial Intelligence research (e.g., the Allen’s seminal work [4, 5]) that takes the interval-based view.

- **Discrete vs. dense vs. continuous.** The instants in a discrete model of time are isomorphic to the natural numbers, i.e., there is the notion that every instant has a unique successor. Instants in the dense model of time are isomorphic to (either) the real or rational numbers: between any two instants there is always another one. Time is continuous if it is isomorphic to real numbers.

- **Bounded vs. unbounded.** This concerns the question of whether time is considered infinite in either or both directions.

- **Linear vs. branching vs. cyclic.** On time instants an order is imposed. Time is linear if it is provided with a total ordering relation. A partial order that satisfies left-linearity is used to model branching time [42], i.e., time is
linear from the past to the present and branching from the present to the future in order to represent the possible evolutions of the world. Moreover, a linear transitive order which is not irreflexive or anti-symmetric is used to model cyclic time.

**Temporal relations**

Time is said to be absolute if it is independent of, e.g., the time of another fact and of the current time, now, for instance it is expressed in terms of dates. Time is said to be relative if it is related to some other time, e.g., the time of another fact or the current time, now. Relationships between times can be qualitative, such as before, after, as well as quantitative if they involve some kind of metrics, such as “interval \( I \) lasts 2 hours” or “3 days before”.

**Multidimensionality of time**

Multiple temporal dimensions are necessary to model intervals (as pairs of points) or multiple kinds of time, e.g., valid time (the time when a fact is true in the modeled reality) and transaction time (the time when a fact is stored in the database). Based on this classification of the time, four models of data are defined, depending on which time they support:

- Snapshot databases do not offer any temporal support.
- Transaction-time databases support transaction time.
- Valid-time databases support valid time.
- Bitemporal databases support both valid and transaction time.

### 3.1.1 Overview on Logical Temporal languages

Interest in research concerning the handling of temporal information has been growing steadily over the past two decades. On the one hand, much effort has been spent in developing extensions of logical languages capable to deal with time (see, e.g., [46, 89]). On the other hand, in the field of databases, many approaches have been proposed to extend existing data models, such as the relational, the object-oriented and the deductive model, to cope with temporal data (see, e.g., the books [112, 45] and references therein). It is clear that a close connection exists between these two strands of research, since temporal logic languages can provide solid theoretical foundations for temporal databases, and powerful knowledge representation and query languages for them [34, 54, 88, 29].

Two main logical formalisms have been developed for time: modal logic, with modal temporal operators, such as since and until, and classical logic with temporal variables.
Temporal and modal logic languages

Several languages based on temporal logic have been defined. The languages such as Chronolog [90], Templog [1], Temporal Datalog [88], Gabbay's Temporal Prolog [52], Sakuragawa's Temporal Prolog [98], MTL [30] directly extend logic programming with temporal operators. Most of them use temporal versions of resolution-based proof procedures and they differ in which temporal constructs they allow, and, therefore, in which classes of problems they can naturally express. We discuss only Templog as an example of this class of languages. For a survey of temporal and modal logic programming consider the overview [89] and [46] for recent trends.

Templog [1] extends classical Horn logic programming languages, such as Prolog [109], to include programs with the following temporal constructs: $\bigcirc u$ ($u$ is true at the next instant of time), $\square u$ ($u$ is always true (from now on)), $\lozenge u$ ($u$ is eventually true). Programs are sets of temporal clauses divided into permanent and initial clauses with particular constraints on the use of the temporal operators in the head and the body of clauses. This approach is point based and it is difficult to ask queries such as *In which period was Frank a researcher?*, that is to extract information about duration of actions, intervals in which some facts hold. This language, as most of the others mentioned, establish relations among instants of time and, for this reason, a disadvantage is that they cannot naturally represent explicit references to time. They are designed to handle relative timing information rather than quantitative timing one. Templog is based on temporal SLD resolution, a sound and complete proof procedure. Moreover two equivalent formulations of Templog's declarative semantics are given: in terms of a least fixpoint and in terms of a minimal Herbrand model [14].

The language MTL, a temporal logic programming language with metric and past operators, presented in [30] subsumes Templog. It allows not only qualitative, but also quantitative temporal information, such as $\square_{[c_1,c_2]} A$, meaning that $A$ always holds between $c_1$ and $c_2$. An interesting feature of this approach is the possibility of translating MTL into an instance of the CLP-scheme [65] over a suitable algebra, where (due to the particular form of the involved constraints) the MTL-resolution can be expressed as a quite efficient restriction of CLP-resolution.

Classical logic languages with temporal variables

In this category of languages we recall Datalog$_{1S}$ [35, 15], Hrycej's Temporal Prolog [62], Temporal Annotated Constraint Logic Programming [50] and Event Calculus [76, 107].

Datalog$_{1S}$ [35, 15] is a language for temporal deductive databases [15], which allows for the implicit representation of the extension of the temporal attributes. The main advantage of implicit representations is expressiveness: they make the representation of infinite extensions possible and they often allow for a more compact representation of finite extensions. Datalog$_{1S}$ is an extension of Datalog such that all
the predicates are extended with an extra parameter for time. The time parameter is constructed using a unary function symbol denoting the successor function. A distinctive feature of Datalog$_{15S}$ is the possibility of representing periodic data.

Hrycej's Temporal Prolog [62] extends Prolog with two additional clause types: temporal references, denoted by $P$ in $T$, which are used to assert that a certain statement $P$ holds exactly during a time interval $T$, and temporal constraints, which axiomatize Allen's relationships between time intervals. This approach attaches time intervals to clauses, it uses an interval-based algebra and it gives a meta-level representation of the knowledge via the predicate HOLD. It is interesting the way the language has been designed in order to have an efficient implementation based on a temporal constraint solver.

While in Hrycej's Temporal Prolog time is associated with clauses, in Temporal Constraint Annotated Logic Programming (TACLP), presented in [48, 49, 50], each formula can be labelled (annotated) by temporal information, for instance

\[
\text{owns}_\text{car}(X) \text{ th}[T_1, T_2] \leftarrow \text{buys}_\text{car}(X) \text{ at } T_1 \land \text{sells}_\text{car}(X) \text{ at } T_2
\]

means that if a person $X$ buys a car at $T_1$ and she/he sells it at $T_2$ then she/he owns the car in the time period from $T_1$ to $T_2$. Formulae can have three possible temporal annotations: $A \text{ at } T$ ($A$ holds at time point $T$), $A \text{ th}[T_1, T_2]$ ($A$ holds throughout the time period from $T_1$ to $T_2$) and $A \text{ in}[T_1, T_2]$ ($A$ holds at some point(s) in $[T_1, T_2]$ - indefinite temporal information). Such approach supports both qualitative and quantitative temporal reasoning about definite and indefinite information with time points and time periods.

Finally we mention Event Calculus [76, 75]. Event Calculus is a treatment of time, based on the notion of events, in the Horn clause subset of classical logic augmented with negation as failure. It is closely related to Allen's interval temporal logic [4, 5]. For example, let $E_1$ be an event in which Bob gives the Book to John and $E_2$ be an event in which John gives Mary the Book. Assume that $E_2$ occurs after $E_1$. Given these event descriptions, we can deduce that there is a period started by the event $E_1$ in which John possesses the book and that there is a period terminated by $E_1$ in which Bob possesses the book. This situation is represented pictorially as

\[
\begin{array}{cccc}
\text{Bob has the Book} & \circ & \text{John has the Book} & \\
\text{E1} & \text{John has the Book} & \circ & \text{Mary has the Book} \\
\text{E2}
\end{array}
\]

A series of axioms for deducing the existence of time periods and the Start and End of each time period are given by using the Holds predicate.

\[
\text{Holds} (\text{before}(e, r)) \text{ if } \text{Terminates}(e, r)
\]
<table>
<thead>
<tr>
<th>Veh</th>
<th>City</th>
<th>Time</th>
<th>Con</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V_1$</td>
<td>Athens</td>
<td>$i_1$</td>
<td>$9 \leq i_1 L, i_1 R \leq 15$</td>
</tr>
<tr>
<td>$V_2$</td>
<td>Patra</td>
<td>$i_2$</td>
<td>$10 \leq i_2 L, i_2 R \leq 14$</td>
</tr>
</tbody>
</table>

Figure 3.1: A generalized temporal relation

means that the relationship $r$ holds in the time period $\text{before}(e, r)$ that denotes a time period terminated by the event $e$. $\text{Holds}(\text{after}(e, r))$ is defined in an analogous way. Event Calculus provides a natural treatment of valid time in databases, and it has been extended in [107, 108] to include the concept of transaction time.

### 3.1.2 Time in Constraint Databases

An interesting application of constraint databases to the modeling of time is the approach by Koubarakis [71]. The aim of Koubarakis’ proposal is to unify into a single framework the representation of definite, indefinite, finite, and infinite temporal information about valid time of tuples by introducing a hierarchy of temporal data models: temporal relations, generalized temporal relations, and temporal tables. The temporal relations model is simply a theoretical tool for providing semantics for the other two models. The model of generalized temporal relations offers the ability to represent infinite temporal information in a finite way. The model of temporal tables extends the model of generalized temporal relations by allowing indefinite (i.e., imprecise and/or qualitative) temporal information.

Time is assumed to be linear, dense and unbounded. There is a single time line which is isomorphic to the set of rational numbers $\mathbb{Q}$. The time entities are points and intervals, defined as pairs of points.

In order to show how infinite and indefinite information is represented in this approach we present two examples taken from [71]. In Figure 3.1 the generalized temporal relation represents the availability of some vehicles: vehicle $V_1$ is on service from 9am to 3pm in Athens. The constraint $9 \leq i_1 L, i_1 R \leq 15$ allows one to express that the vehicle is on service for each subinterval in the interval [9am, 3pm], thus a generalized relational tuple is a finite representation of an infinite set of relational tuples. Generalized temporal relations extend temporal relations in the following way:

- **variables** are allowed as values for temporal attributes, and
- every tuple is tagged with a local temporal constraint on its variables.

Variables in a generalized relational tuple are essentially universally quantified.

On the other hand, Figure 3.2 shows a temporal table which allows the representation of indefinite information by using existentially quantified variables called
### 3.2 Spatial data models and languages

Following Güting [59], a spatial database is a database system that offers spatial data types in its data model and query language and supports such data types in its implementation, providing at least spatial indexing and efficient algorithms for spatial join.

The spatial data models proposed in the literature have been classified as follows [92]:

- the Raster Model [58, 99]. An object is given by a finite number of raster points. Raster points are uniformly distributed following an easy geometric pattern, which is normally a square.

- the Vector Model or Spaghetti Model [79]. An object is intensionally deduced from its contour, which is approximated by a polyline, and it is usually represented by a list of points.
• The Peano Model [79] also uses a finite number of object-points, but here these points are distributed non-uniformly, according to the form of the object. This distribution is based on the Peano curve.

• Polynomial Model [68, 93]. The spatial information is stored in relations. Each relation contains at most one spatial attribute for representing spatial objects described by a semi-algebraic set of the form

\[ \{(x_1, \ldots, x_n) \mid x_1, \ldots, x_n \in \mathbb{R} \land \phi(x_1, \ldots, x_n)\} \]

where \( \phi(x_1, \ldots, x_n) \) is a semi-algebraic formula that contains boolean operators, existential and universal quantifiers and whose terms are comparisons, using \(<, \leq, >, \geq, =, \neq\), between polynomials whose coefficients are rational numbers. \( x_1, \ldots, x_n \) are free variables of \( \phi \).

• PLA-Model [79]. Only some kind of topological information is handled without dealing with the exact position and form of the spatial objects.

For further references, we give a more detailed account of the 2-Spaghetti Model (two-dimensional space), and the Polynomial Model. Finally, the work of Egenhofer [40] on topological relationships is discussed.

2-spaghetti Model

The 2-spaghetti model allows one to represent only spatial objects that are composed of a finite set of closed polygons. Actually, each object can be decomposed into a set of triangles (some are degenerate triangles like line segments or points), where each triangle is represented by its three corners in a single relational database table. Worboys [118] uses the notion of simplex to capture this basic component of the object.

Example 3 Let us consider Figure 3.3. In the 2-spaghetti data model this figure is represented by the following relation.

<table>
<thead>
<tr>
<th>ID</th>
<th>x</th>
<th>y</th>
<th>x'</th>
<th>y'</th>
<th>x''</th>
<th>y''</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p_1 )</td>
<td>10</td>
<td>4</td>
<td>10</td>
<td>4</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>( l_1 )</td>
<td>5</td>
<td>10</td>
<td>9</td>
<td>6</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>( l_1 )</td>
<td>9</td>
<td>6</td>
<td>9</td>
<td>6</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>( t_1 )</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>7</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>( r_1 )</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>( r_1 )</td>
<td>12</td>
<td>11</td>
<td>12</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>( p_2 )</td>
<td>3</td>
<td>8</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>( p_2 )</td>
<td>3</td>
<td>8</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>( p_2 )</td>
<td>4</td>
<td>9</td>
<td>6</td>
<td>5</td>
<td>7</td>
<td>6</td>
</tr>
</tbody>
</table>

Each tuple specifies the name of the object it belongs to, the vertex coordinates of the simplex. For instance, notice that a rectangle is represented by two triangles.
3.2. Spatial data models and languages

![Image of spatial objects]

Figure 3.3: Some spatial objects.

The abstract semantics of a 2-spaghetti data model is, for each object, the set of points that belong to the area of the plane that is within any of the triangles associated with that object. Thus, it is point-based. Usually, the query languages hide the internal representation of spatial objects, referring explicitly to the objects themselves. They provide operators that work on polygons, e.g., overlap, union, intersection, however the semantics of those languages is also point-based.

This model is popular because there exist very efficient algorithms for triangulating polygons [99] and for detecting properties, such as whether two polygons overlap, whether a point lies in a polygon and so on.

**Polynomial Model**

The Polynomial Model is more general than the Spaghetti Model since it allows one to represent all the geometrical figures definable in elementary geometry, i.e., first-order logic over the reals. As a representative of this class, we describe the approach of Paredaens et al. [93] which extends constraint databases and offers a calculus for specifying spatial queries.

A spatial database scheme, $S$, is a finite set of relation names. Each relation name, $R$, has a type $\tau(R)$ which is a pair of natural numbers $[n,m]$. Here, $n$ denotes the number of non-spatial columns and $m$ the dimension of the single spatial column of $R$. Consider a relation type $[n,m]$. An intensional tuple (or tuple) of type $[n,m]$ has the form $(a_1, \ldots, a_n; \phi(x_1, \ldots, x_m))$, with $a_1, \ldots, a_n$ non-spatial values of some domain, $\mathbb{E}$, and with $\phi(x_1, \ldots, x_m)$ a quantifier-free real formula of arity $m$. An $i$-relation of type $[n,m]$ is a finite set of tuples of type $[n,m]$. An $i$-instance of a
scheme $S$ is a mapping on $S$, assigning to each relation name $R$ of $S$ an i-relation of type $\tau(R)$.

This model differs from constraint databases because it allows more general formulae in the spatial component (\wedge and \lor of constraints) and it distinguishes a spatial and a value part (ordinary attribute) of an ituple. This distinction is needed to define the notion of genericity (consistency criterion) for spatial queries.

**Example 4** In the following table, we show how Figure 3.3 is modeled in the Polynomial model of Paredaens et al. [93].

<table>
<thead>
<tr>
<th>ID</th>
<th>$x, y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>$x = 10 \land y = 4$</td>
</tr>
<tr>
<td>$l_1$</td>
<td>$(5 \leq x \land x \leq 9 \land y = -x + 15) \lor (x = 9, 3 \leq y, y \leq 6)$</td>
</tr>
<tr>
<td>$t_1$</td>
<td>$2 \leq x, x \leq 6, 3 \leq y, y \leq 7, y \leq -x + 9$</td>
</tr>
<tr>
<td>$r_1$</td>
<td>$1 \leq x, x \leq 12, 2 \leq y, y \leq 11$</td>
</tr>
<tr>
<td>$p_2$</td>
<td>$3 \leq x, 5 \leq y, x - 1 \leq y, y \leq -x + 13, x + 5 \geq y$</td>
</tr>
</tbody>
</table>

The spatial component is modeled by its analytic representation.

The semantics of an ituple $t = (a_1, \ldots, a_n; \phi(x_1, \ldots, x_m))$ of type $[n,m]$ is the possibly infinite subset of $\mathbb{R}^n \times \mathbb{R}^m$ called e-relation, denoted by $\text{ext}(t)$ and defined as the Cartesian product $\{(a_1, \ldots, a_n)\} \times S$, in which $S \subseteq \mathbb{R}^m$ is the semi-algebraic set defined by $\{(x_1, \ldots, x_m) \mid \phi(x_1, \ldots, x_m)\}$. The semantics of an i-relation $r$ is the e-relation denoted as $\text{ext}(r)$ and defined as $\bigcup_{t \in r} \text{ext}(t)$. The semantics of an i-instance $I$ is defined in the obvious way in term of the e-relations of $I(R)$, where $R$ is a relation name of the database.

The provided query language gives the usual set of operations of the relational algebra (union, difference, Cartesian product, selection, projection) suitably modified in order to deal with the spatial component of the ituples. For instance the Cartesian product between two i-relations $r_1$ of type $[n,m]$ and $r_2$ of type $[n',m']$ is defined as follows:

$r_1 \times r_2 = \{(\text{val}(t), \text{val}(t')); \text{spat}(t) \land \text{spat}(t') \mid t \in r_1, t' \in r_2\}$, of type $[n+n',m+m']$

where $\text{val}(t)$ and $\text{spat}(t)$ respectively denote the value and spatial part of the ituple $t$. The ituples belonging to the Cartesian product have as value part the value parts of an ituple $t$ in $r_1$ and an ituple $t'$ in $r_2$ and as spatial part the conjunction ($\land$) of the spatial parts of $t$ and $t'$.

The disadvantage of this approach is that it may be too general in an implementation perspective. For this reason, the set of spatial objects has been restricted in various ways [114, 16, 56]. The basic idea is to focus on "linear" data types, i.e., objects represented by formulae built from linear polynomials. This choice is motivated by two factors: this class of data types includes most of the commonly used spatial data, and operations on typical linear data, such as lines, and polygons, have efficient algorithms. In Section 3.2.1 we give further hints on the linear constraint model.
9-intersection model

Finally, we briefly recall the work by Egenhofer [41, 40, 39]. It is the spatial counterpart of Allen’s work on time intervals. He focuses on the class of topological relationships between spatial objects. A topological relation is a property invariant under homeomorphisms, for instance it is preserved if the objects are translated or scaled or rotated. We restrict our attention to a space with only two dimensions and we present the 9-intersection model. Such a model is based on the intersection of the interior \((A^\circ,B^\circ)\), the complement \((A^-,B^-)\) and the boundary \((\delta A,\delta B)\) of two 2-dimensional connected objects \(A\) and \(B\). Therefore a relation between \(A\) and \(B\) is represented by \(R(A,B)\) as follows:

\[
R(A, B) = \begin{pmatrix}
\delta A \cap \delta B & \delta A \cap B^\circ & \delta A \cap B^- \\
A^\circ \cap \delta B & A^\circ \cap B^\circ & A^\circ \cap B^- \\
A^- \cap \delta B & A^- \cap B^\circ & A^- \cap B^-
\end{pmatrix}
\]

Each of these intersections can be empty or not empty. Of the \(2^9\) possible different topological relations, only eight of these relations can be realized between two 2-dimensional objects and they are illustrated in Figure 3.4. As observed in [91], these relations are not sufficient to fully characterize the topological relationships between objects. For instance, if three regions \(A\), \(B\) and \(C\) pairwise overlap, the intersection of the three together might or might not be empty but this cannot be expressed by the previous relations. [91] defines languages which are complete for topological properties.
3.2.1 Space in Constraint Databases

As already pointed out, the constraint databases found their key intuition on finitely representing possibly infinite sets, their data models offer a promising paradigm for the representation of many sorts of data in a unified framework. These motivations lead to their suitability for modeling spatial objects. In this way, a spatial object, seen as an infinite set of points, can be dealt with as a first class citizen with an explicit representation. For instance a convex polygon, which is the intersection of a set of half-planes, is defined by the conjunction of the inequalities defining each half-plane. A non convex polygon by the union (logical disjunction) of a set of convex polygons.

To model spatial information in constraint databases, the linear constraints over rationals or real polynomial constraints are often used as logical theory of constraints. The trend, initiated by [93], of developing first-order based algebraic query languages, using as underlying theory the real polynomial constraint theory, is moving to the definition of languages based on the linear constraint theory because of its efficient algorithms. Linear constraints have been shown to fit the needs of spatial data in the vector mode. The fundamental difference between the two approaches relies on the explicit definition of the spatial objects in the data model (e.g. a polygon is explicitly the infinite set of points it contains versus the implicit definition by the sequence of border points). It is possible to manipulate spatial objects through standard set operations (e.g. those of relational algebra).

A lot of research has developed, starting from Kanellakis et al. generalized relational model [68], leading to new models that can represent better spatial objects. Most of the existing proposals define new generalized relational models using particular classes of constraints to represent the tuples and algebras that correspond exactly to the traditional relational algebra, and whose semantics is defined with respect to the effect of the usual algebraic operators on infinite relations. This is the case of the real polynomial model of [93], as described in the previous section, and the linear constraint model of Grumbach et al. [56]. Other proposals redefine logically the generalized relational model to express complex objects as, the nested generalized relational model of Beuhi et al. [16], and design algebras that apply on special generalized tuples. In the following we describe the linear constraint model and the nested generalized relational model.

**Linear constraint model.** Linear constraint models consider linear constraints in the first-order language $\mathcal{L} = \{\leq, +\} \cup \mathbb{Q}$ over the structure $\mathcal{Q} = \langle \mathbb{Q}, \leq, +, (q)_{q \in \mathbb{Q}} \rangle$ of the linearly ordered set of the rational numbers with rational constants and addition. Constraints are in this case linear equations or inequalities of the form $\sum_{i=1}^{p} a_i x_i \Theta a_0$, where $\Theta$ is a predicate among $= \text{ or } \leq$, the $x_i$'s denote variables and the $a_i$'s are integer constants.

Let $\sigma = \{R_1, \ldots, R_n\}$ be a database schema such that $\mathcal{L} \cap \sigma = \emptyset$, where $R_1, \ldots, R_n$ are relation symbols. We distinguish between logical predicates (e.g., $=, \leq$) in $\mathcal{L}$ and
relations in $\sigma$. Then linear constraint relations are defined as follows:

**Definition (linear constraint relation).** Let $S \subseteq Q^k$ be a $k$-ary relation. The relation $S$ is a linear constraint relation if there exists a formula $\phi(x_1, \ldots, x_k)$ in $\mathcal{L}$ with $k$ distinct free variables $x_1, \ldots, x_k$ (called representation of $S$) such that:

$$Q \models \forall x_1 \cdots x_k (S(x_1, \ldots, x_k) \leftrightarrow \phi(x_1, \ldots, x_k))$$

Two more restrictions are imposed on the class of all infinite relations: the first one is to represent relations by quantifier free formulae and the second one enforces the formulae to be in disjunctive normal form (DNF), as in the generalized relational model of [68]. In terms of 2-dimensional applications, the second restriction implies that polygons must be decomposed into convex component polygons (convexified). Grumbach et al. in [56] call this restriction the convex normal form and generalize it, motivated both by its logical simplicity and its algorithmic efficiency, to a representation of objects with “holes” (see Figure 3.5):

**Definition (convex with holes normal form).** A formula is in the convex with holes normal form (CHNF) if it is either quantifier free in DNF, or it is of the form $F - F'$, where $F$ and $F'$ are two formulae in CHNF of the same arity, and $-$ is the set difference.

Nested generalized relational model. In [16, 17] the definition of generalized tuples is extended, in order to be able to express more general set in their extension. This is possible by using additional logical connectives in generalized tuples. For instance, to model concave sets inside a generalized tuple, disjunction is used. Thus generalized relations are defined as follows:

**Definition (generalized relation).** Let $\Phi$ be a decidable logical theory and $\Sigma$ a set of first-order logical connectives without quantifiers. A generalized tuple on $\Phi$ and $\Sigma$ over variables $x_1, \ldots, x_k$ is a first-order formula whose free variables belong to $x_1, \ldots, x_k$, atoms are atomic formulae on $\Phi$, and logical connectives belong to $\Sigma$. A generalized relation on $\Phi$ and $\Sigma$ is a set of generalized tuples on $\Phi$ and $\Sigma$, and a generalized database on $\Phi$ and $\Sigma$ is a set of generalized relations on $\Phi$ and $\Sigma$.  

![Figure 3.5: Representation of a polygon with a hole](image-url)
Notice that the generalized tuples introduced by Kanellakis et al. in [69] are generalized tuples on $\Phi$ and $\Sigma = \{\land\}$.

A new semantics, called nested semantics, is assigned to generalized relations. If generalized relations are interpreted as nested relations, each generalized tuple represents a possibly infinite set of relational tuples, implicitly represented by its extension, and a generalized relation is not any longer an (infinite) set of relational tuples (i.e., points of the embedding space) but is a finite set of sets, each representing a (possibly infinite) set of relational tuples (i.e., each set represents a spatial object).

Based on the nested semantics, an algebra is proposed that is able to manipulate generalized relations under two different points of view, either handling each generalized relation as a finite set of objects or as a possibly infinite set of points (i.e., relational tuples). Indeed two classes of operators correspond to these points of view, respectively called set operators and tuple operators.

### 3.3 Spatio-temporal data models and languages

Among the temporal and spatial approaches we presented in the previous sections only constraint databases can provide a uniform data model for both time and space. In fact, if we consider as class of constraints linear polynomials, we can model both temporal and spatial data with efficient implementations. For instance, Grumbach et al. and Behussi et al. have extended their approaches for handling spatial information to include also the treatment of time, respectively in [57] and [17].

Another unified model for spatial and temporal information has been introduced by Worboys [118]. The basic idea is to attach to each elemental spatial object a bitemporal element which is a finite set of pairs of intervals, representing respectively the transaction and the valid time of the object. The elemental spatial object is a simplex which is either a single point, a finite straight line segment or a triangular area. The pair \langle simplex, set of bitemporal elements\rangle is called ST-simplex. A complex spatial object, temporally referenced, is called ST-complex and it is modeled by a finite set of ST-simplices, subject to some constraints that we do not report.

**Example 5** [116] An example of ST-complexes is given in Figure 3.6. To each vertex, line, and to the area (i.e. the simplices) of the triangle a bitemporal element is associated, having on the horizontal axis the transaction time and on the vertical axis the valid time.

The query language provides operators to make the union, difference and intersection of two ST-complexes, and the selection on the temporal and spatial component. Moreover, it is also given a topological operator to return the boundary of an ST-complex.

In the Worboys model, the temporal and the spatial dimensions are independent and this prevents from describing spatial relations which may be parametric with
respect to time, i.e. with respect to their evolution, for instance it is not possible to model moving points.

Recently, Chomicki and Revesz [37, 38] have introduced a new model, called Parametric 2-spaghetti model, that generalizes the 2-spaghetti model by allowing an interaction between spatial and temporal attributes. Vertex coordinates can now be linear functions of time.

**Example 6** [37] Let us suppose that we have a rectangular area on a shore and a tide is coming in. The front edge of the tide water is a linear function of time. The behavior of the tide is described in Figure 3.7: at 1:00 am the area flooded by the water will be a point, at 8:00 a triangle and so on. These data are represented in the following table.

<table>
<thead>
<tr>
<th>ID</th>
<th>x</th>
<th>y</th>
<th>x′</th>
<th>y′</th>
<th>x″</th>
<th>y″</th>
<th>From</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>r₁</td>
<td>3</td>
<td>10</td>
<td>3</td>
<td>10</td>
<td>3</td>
<td>10</td>
<td>1</td>
<td>+∞</td>
</tr>
<tr>
<td>r₁</td>
<td>3</td>
<td>10</td>
<td>3</td>
<td>11 − t</td>
<td>2 + t</td>
<td>10</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>r₁</td>
<td>3</td>
<td>10</td>
<td>3</td>
<td>3</td>
<td>10</td>
<td>10</td>
<td>8</td>
<td>+∞</td>
</tr>
<tr>
<td>r₁</td>
<td>3</td>
<td>3</td>
<td>10</td>
<td>10</td>
<td>3</td>
<td>11 − t</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>r₁</td>
<td>10</td>
<td>10</td>
<td>3</td>
<td>11 − t</td>
<td>10</td>
<td>18 − t</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>r₁</td>
<td>3</td>
<td>3</td>
<td>10</td>
<td>10</td>
<td>3</td>
<td>1</td>
<td>10</td>
<td>+∞</td>
</tr>
<tr>
<td>r₁</td>
<td>10</td>
<td>10</td>
<td>3</td>
<td>1</td>
<td>10</td>
<td>8</td>
<td>10</td>
<td>+∞</td>
</tr>
<tr>
<td>r₁</td>
<td>3</td>
<td>1</td>
<td>10</td>
<td>8</td>
<td>t − 7</td>
<td>1</td>
<td>10</td>
<td>17</td>
</tr>
<tr>
<td>r₁</td>
<td>t − 7</td>
<td>1</td>
<td>10</td>
<td>18 − t</td>
<td>10</td>
<td>8</td>
<td>10</td>
<td>17</td>
</tr>
<tr>
<td>r₁</td>
<td>3</td>
<td>1</td>
<td>10</td>
<td>1</td>
<td>10</td>
<td>8</td>
<td>17</td>
<td>+∞</td>
</tr>
<tr>
<td>r₁</td>
<td>10</td>
<td>1</td>
<td>10</td>
<td>1</td>
<td>10</td>
<td>1</td>
<td>17</td>
<td>+∞</td>
</tr>
</tbody>
</table>

*Observe that this relation is not a standard one because its values are parametric.*
Query languages for the Parametric 2-spaghetti data model have not been defined yet. This model has been used in [36] to animate spatio-temporal objects.

Finally, we want to analyse the approach proposed by Erwig, Gütting, Schneider and Vazirgiannis [43, 60, 44]. They define collections of abstract data types for spatial values changing over time. Essentially, they introduce data types for moving points and moving regions together with a set of operations on such entities.

The design of the model for spatio-temporal data is based on a type constructor $\tau$ which transforms any given atomic data type $\alpha$ into a type $\tau(\alpha) = \text{time} \rightarrow \alpha$ where time $= \mathbb{R}$, that is a continuous model of time is supported. In particular this constructor is applied to two spatial data types, point and region, allowing one to represent two fundamental abstractions moving point and moving region. A moving point is of type $\text{mpoint} = \text{time} \rightarrow \text{point}$ and it is used to represent objects for which only the position in space is relevant, whereas a moving region is of type $\text{mregion} = \text{time} \rightarrow \text{region}$ and both the position and extent are of interest, i.e., the object can change its position and/or grow or shrink.

Then a set of operations are defined on these types. For instance consider the following functions

$$
\text{mdistance} : \text{mpoint} \times \text{mpoint} \rightarrow \text{mreal}
$$

$$
\text{trajectory} : \text{mpoint} \rightarrow \text{line}
$$

$$
\text{length} : \text{line} \rightarrow \text{real}
$$

where $\text{mreal} = \tau(\text{real})$ (a moving real) and line is a data type describing a curve in two-dimension space. The function $\text{mdistance}$ returns the distance between two moving points at all times, and hence returns a time changing real number, whereas $\text{trajectory}$ is the projection of a moving point onto the plane and $\text{length}$ returns the total length of a line value.

The presented data types can be embedded into any DBMS data model as attributes data types, and the operations be used in queries. In [44] this approach has
been extended in order to provide a framework which allows one to build more and more complex spatio-temporal predicates starting from a small set of elementary ones. These predicates are particularly suitable to characterize developments, for instance, a predicate for capturing the scenario of a point entering or crossing a region can be easily modeled.
Chapter 4

Operators for combining logic programs

Our research starts from the results presented in [23, 25], where a logic programming approach to program composition was presented. The approach introduced in [23, 25] centers on the definition of a family of meta-level composition operations over named logic programs. Such operations allow to build complex programs by combining simpler programs together. From a programming perspective, the introduction of the operations has been shown to enhance the expressive power and knowledge representation capabilities of logic programming by supporting a number of programming techniques, ranging from software engineering to artificial intelligence applications [25, 97, 9].

The chapter is devoted to introduce the set of such composition operations, originally defined by Mancarella and Pedreschi [83] and further developed by Brogi et al. in [23, 25].

4.1 Syntax

In [23, 25], a set of three basic meta-level operations for composing logic programs is introduced: Union (denoted by $\cup$), Intersection ($\cap$), Encapsulation (*).

The operations define the language of program expressions, formally defined with the following abstract syntax:

$$
Exp ::= Pname \mid Exp \cup Exp \mid Exp \cap Exp \mid (Exp)^*
$$

where $Pname$ is the syntactic category of program names, i.e., constant symbols denoting the name of the program. A program name univocally identifies a finite set of clauses. Hereafter we will call plain programs a set of clauses identified by a name and we often refer to the language of program expressions as the multi-theory framework.
This language is a conservative extension of logic programming which allows one to move from a single logic program to a collection of logic programs and provides a set of composition operators over such programs. The language of program expressions has been largely examined both from the theoretical point of view and from the application point of view (e.g., in [23, 25, 9]). For instance, it is proved that the operations $\cup$, $\cap$ and $*$ satisfy a number of algebraic properties such as associativity, commutativity and distributivity. Moreover, other operations have been defined to handle specific problems, for instance the operation Import ($\triangleright$) allows one to have a fine grained notion of information hiding/export. Other composition operations can be found in [25, 10].

We recall that the language $L$ in which programs and queries are written is fixed. Indeed, $L$ is a multi-sorted language: we have two disjoint sets of function symbols, one consisting of program names and program composition operations, the other containing all other constant and function symbols that may occur in programs.

The following example shows the use of the meta-level operations $\cup$ and $\cap$ for programming: these are the two operations we will largely employ throughout the thesis.

**Example 7** A movie ticket is $\$5 for kids and $\$7 for adults. To represent such knowledge we define a program BoxOffice.

**BoxOffice:**

\begin{align*}
ticket-cost(5, X) & \leftarrow \text{age}(X, Y), \text{lesseq}(Y, 16) \\
ticket-cost(7, X) & \leftarrow \text{age}(X, Y), \text{greaterthan}(Y, 16) \\
\text{age}(X, Y) & \leftarrow \text{today}(D1), \text{born}(X, D2), \text{year-diff}(D2, D1, Y)
\end{align*}

where $\text{year-diff}(D2, D1, Y)$ computes the difference in years between two dates and the predicates $\text{lesseq}$ and $\text{greaterthan}$ represent the relation less or equal and greater than between natural numbers, their definitions are the obvious ones.

Notice that the program is parametric with respect to the actual day of the year and the customer’s birthday. The predicate $\text{today}$ is defined in a separate program which represents the current date, e.g.,

**Today:** \text{today(May 28 1996)}.

and the customer’s birthday is given in a separate program like

**Tom:** \text{born(tom, May 7 1981)}.

The program expression BoxOffice $\cup$ Tom $\cup$ Today behaves as a plain program containing the clauses of BoxOffice, Tom and Today. Hence it is exactly this program expression that is requested to know how much Tom has to pay to go to the cinema today.

Now suppose we want to model the fact that a cinema is a club, i.e., one has to be a member to watch movies. The general program BoxOffice has to be “constrained”
in order to prevent people who are not members from buying tickets. To obtain this behavior we intersect the program BoxOffice with the following program.

Club:

\[
ticket\text{-}cost(C,X) \leftarrow \text{registered}(X)
\]

\[
age(X,Y).
\]

The program Club establishes a further constraint on how to compute the predicate ticket-cost, i.e., X has to be registered, and leaves unchanged the way the age of a person is computed. Summing up the program expression BoxOffice \( \cap \) Club behaves as:

\[
ticket\text{-}cost(5,X) \leftarrow age(X,Y), \text{lesseq}(Y,16), \text{registered}(X)
\]

\[
ticket\text{-}cost(7,X) \leftarrow age(X,Y), \text{greaterthan}(Y,16), \text{registered}(X)
\]

\[
age(X,Y) \leftarrow \text{today}(D1), \text{born}(X,D2), \text{year-diff}(D2,D1,Y)
\]

which satisfies the requirements to be a private cinema.

### 4.2 Semantics

In this section two semantics for the language of program expressions are presented: a top-down semantics obtained by extending the vanilla meta-interpreter, and a bottom-up semantics based on the definition of an immediate consequence operator. Then the meta-interpreter is proved to be sound and complete with respect to the least fixpoint of the bottom-up semantics [23].

**Top-down semantics via meta-interpreter**

Program composition operations are implemented by a meta-interpreter which exploits the knowledge of the separate programs and combines them at the meta-level, without actually building a new program. The meta-level definition of the operations is given by adding new clauses to the vanilla meta-interpreter (see Section 2.2.1). The reading of the resulting meta-interpreter is straightforward and, most importantly, the meta-logical definition shows that the multi-theory framework can be expressed from inside logic programming itself.

Following Bowen and Kowalski [21], the two-argument predicate demo is employed to represent provability. The extra argument of demo is used to explicitly denote the interpreted object program, or more generally, in our setting, a program expression. Namely, \( \text{demo}(\mathcal{E}, G) \) represents that the formula \( G \) is provable in the program expression \( \mathcal{E} \).
Moreover, the simple naming convention used by Kowalski and Kim in [77] is adopted. Object programs are named by constant symbols, denoted by capital letters such as $P$ and $Q$. Object level expressions are represented by themselves at the meta-level. In particular, object level variables are denoted by meta-level variables, according to the non-ground representation [61]. An object level program $P$ is represented at the meta-level by a set of axioms of the kind $\text{clause}(P, A, B)$, one for each object level clause $A \leftarrow B$ in $P$. Each program composition operation is represented at the meta-level by a functor. The meaning of each functor is defined by new clauses extending the definition of the $\text{clause}$ predicate.

The resulting meta-interpreter consists of the following clauses

\[
demo(\mathcal{E}, \text{empty}) \leftarrow
\]

\[
demo(\mathcal{E}, (G_1, G_2)) \leftarrow demo(\mathcal{E}, G_1), demo(\mathcal{E}, G_2)
\]

\[
demo(\mathcal{E}, A) \leftarrow \text{clause}(\mathcal{E}, A, G), demo(\mathcal{E}, G)
\]

\[
\text{clause}(\mathcal{P} \cup \mathcal{Q}, A, G) \leftarrow \text{clause}(\mathcal{P}, A, G)
\]

\[
\text{clause}(\mathcal{P} \cup \mathcal{Q}, A, G) \leftarrow \text{clause}(\mathcal{Q}, A, G)
\]

\[
\text{clause}(\mathcal{P} \cap \mathcal{Q}, A, (B, C)) \leftarrow \text{clause}(\mathcal{P}, A, B), \text{clause}(\mathcal{Q}, A, C)
\]

\[
\text{clause}(\mathcal{P}^*, A, \text{empty}) \leftarrow demo(\mathcal{P}, A)
\]

Clauses (4.1)–(4.3) simulate the computational model of logic programming and extend the clauses of the vanilla meta-interpreter (clauses (2.1)–(2.3)) in the sense that now we have to specify in which program expression the goal is computed.

Clauses (4.4) and (4.5) state that a clause $A \leftarrow G$ belongs to the union of two program expressions $\mathcal{P}$ and $\mathcal{Q}$ if it belongs either to $\mathcal{P}$ or to $\mathcal{Q}$. A clause $(A \leftarrow B, C)$ belongs to the intersection of two program expressions $\mathcal{P}$ and $\mathcal{Q}$ if $A \leftarrow B$ belongs to $\mathcal{P}$ and $A \leftarrow C$ belongs to $\mathcal{Q}$ (clause 4.6). A unit clause $A$, belongs to the encapsulated program $\mathcal{P}^*$ if $A$ is provable in $\mathcal{P}$ (clause (4.7)). In this way, the code of $\mathcal{P}$ is hidden to all the other programs, which may only refer to the set of atoms which are provable in $\mathcal{P}$.

A program expression $\mathcal{E}$ can be queried by $\text{demo}(\mathcal{E}, G)$, where $G$ is an object level goal.
4.2. Semantics

Bottom-up semantics

To give a declarative semantics to program expressions, a higher-order semantics is adopted. In fact, the results reported in [27] show that the least Herbrand model semantics of logic programs does not lift smoothly to program expressions. Fundamental properties of semantics, like compositionality and full abstraction, are definitely lost. Intuitively, the least Herbrand model semantics is not compositional since it identifies programs which have different meanings in terms of program compositions. Actually, all the programs whose least Herbrand model is empty are identified with the empty program. For example, the programs

\[ r \leftarrow s \quad r \leftarrow q \]

are both denoted by the empty model, though they behave quite differently when composed with other programs (e.g., with \( q \)).

Mancarella and Pedreschi showed in [83] that if the meaning of a program \( P \) is denoted by the immediate consequence operator \( T_P \) itself (rather than by the least fixpoint of \( T_P \)), then such a meaning is a homomorphism for several interesting operations on programs.

Therefore the semantics of a program expression is assumed to be the immediate consequence operator associated to it, i.e., a function from Herbrand interpretations to Herbrand interpretations.

\[ T : Exp \rightarrow \wp(\mathcal{B}_L) \rightarrow \wp(\mathcal{B}_L) \]

Throughout the thesis, given a program expression \( \mathcal{E} \), we write \( f_{\mathcal{E}} \), to denote \( f(\mathcal{E}) \) where \( f \) is a function having a program expression as argument.

The semantics of the operations is given in a compositional way by expressing the meaning of a program expression in terms of the meanings of its sub-expressions.

**Definition 1 (Bottom-up semantics)** Let \( \mathcal{E} \) be a program expression, the function \( T_{\mathcal{E}} : Exp \rightarrow \wp(\mathcal{B}_L) \rightarrow \wp(\mathcal{B}_L) \) is defined as follows

1. (\( \mathcal{E} \) is a plain program \( P \))
   \[ T_P(I) = T_P(I) = \{ A \mid A \leftarrow B_1, \ldots, B_n \in \text{ground}_L(P) \land \{B_1, \ldots, B_n\} \subseteq I \} \]
2. (\( \mathcal{E} = \mathcal{P} \cup Q \))
   \[ T_{P \cup Q}(I) = T_P(I) \cup T_Q(I) \]
3. (\( \mathcal{E} = \mathcal{P} \cap Q \))
   \[ T_{P \cap Q}(I) = T_P(I) \cap T_Q(I) \]
4. (\( \mathcal{E} = \mathcal{P}^* \))
   \[ T_{P^*}(I) = T_{P}^\omega, \text{ where } T_{P}^\omega \text{ is the least fixpoint of } T_P. \]
Union and intersection of program expressions directly relate to the corresponding set-theoretic operations. Namely, given an interpretation \( I \), the set of immediate consequences of the union (resp. intersection) of two program expressions \( \mathcal{P} \cup \mathcal{Q} \) (resp. \( \mathcal{P} \cap \mathcal{Q} \)), from \( I \), is the set-theoretic union (resp. intersection) of the sets of immediate consequences of the component sub-expressions (\( \mathcal{P} \) and \( \mathcal{Q} \)), from \( I \). Furthermore, the encapsulation \( \mathcal{P}^* \) of a program expression \( \mathcal{P} \) is denoted by the least fixpoint of \( \mathcal{T}_\mathcal{P} \). Intuitively, for any interpretation \( I \), the set of immediate consequences of an encapsulated program expression \( \mathcal{P}^* \) is the set of formulae which may be derived from \( \mathcal{P} \) in an arbitrary (finite) number of steps, independently of \( I \).

Informally, union and intersection mirror two forms of cooperation among programs. In the case of union, either program may be used to derive immediate consequences. In the case of intersection, both programs must agree for deriving immediate consequences. Finally, encapsulation is a unary operation which substitutes a program with its least Herbrand model, thus affecting the way programs may cooperate, when composed together.

**Least fixpoint semantics**

The immediate consequence operator \( \mathcal{T}_\mathcal{E} \) is continuous on the lattice \( (\mathcal{E}(\mathcal{B}_L), \subseteq) \), hence the Fixpoint Theorem (Theorem 1) ensures that there exists the least fixpoint and it is \( \mathcal{T}_\mathcal{E}^\omega = \bigcup_{i \in \mathbb{N}} \mathcal{T}_\mathcal{E}^i \). Such a fixpoint is the least fixpoint semantics of a program expression \( \mathcal{E} \).

Finally, in [23] it is shown that the meta-logical and the least fixpoint semantics coincide with respect to object level goals, i.e., given a program expression \( \mathcal{E} \) and the meta-program \( V \) containing both the meta-level representation of the object level programs and the clauses (4.1)–(4.7), for any object level atomic formula \( A \), the following statement holds:

\[
\text{demo}(\mathcal{E}, A) \in T_V^\omega \iff A \in \mathcal{T}_\mathcal{E}^\omega
\]

### 4.3 Message passing

It is worth noting that the language of program expressions is employed as a meta-language for composing object logic programs. In [26, 97] a single language amalgamating the language in which programs are written (object program) and the language of program expressions (meta-language) is presented. Namely, plain programs are extended programs [26] which may contain meta-level calls to program expressions in clause bodies. More precisely, a program is a finite set of extended definite Horn clauses of the form

\[
A \leftarrow B_1, \ldots, B_n
\]

where \( A \) is an atom and each \( B_i \) is either an atom or a meta-level formula of the form \( \text{Bwr}t \mathcal{E} \) where \( B \) is an atom and \( \mathcal{E} \) is a program expression. A goal like \( \text{Bwr}t \mathcal{E} \)
introduces a form of message passing between object level programs. The idea is that the program containing the goal \( B \text{ wrt } \mathcal{E} \), sends the message \( B \) to the program expression \( \mathcal{E} \). As usual, logical variables act as input/output channels between programs.

The introduction of the \texttt{wrt} construct provides a computational model based on cooperation among programs through message passing. During the evaluation process a program may request the evaluation of a subquery to another program (or composition of programs) through a message call, in such a case the computation moves to another program (or composition of programs).

The meta-level representation of the \texttt{wrt} construct is obtained by adding the following clause to the meta-interpreter.

\[
demo(\mathcal{E}, A \text{ wrt } Q) \leftarrow demo(Q, A) \quad (4.8)
\]

This clause states that a goal of the form \( A \text{ wrt } Q \) is provable in \( \mathcal{E} \) if the goal \( A \) is provable in \( Q \). Therefore message passing is implemented as a switch of context.

**Example 8** Suppose that we want to represent information about a school: its students and its courses. Moreover in the school there is a library. Library information is maintained in a database which is different from the one containing information about students and courses. Finally, a third database stores information about teachers. The library would like to use some information about students and teachers in order to automatize loans. To accomplish that, the following programs are defined:

**School:**

\[
\begin{align*}
\text{exam(engl).} & \\
\text{student(mary).} & \\
\text{exam(math).} & \\
\text{student(john).} & \\
\end{align*}
\]

**Teach:**

\[
\begin{align*}
\text{prof(william).} & \\
\text{teaches(isaac,phys).} & \\
\text{prof(isaac).} & \\
\text{teaches(isaac,math).} & \\
\end{align*}
\]

**Lib:**

\[
\begin{align*}
\text{book(hamlet).} & \\
\text{book(principia).} & \\
\text{other_user(pat).} & \\
\text{user(X) \leftarrow student(X) \text{ wrt } School} & \\
\text{user(X) \leftarrow prof(X) \text{ wrt Teach,}} & \\
\text{teaches(X,E) \text{ wrt Teach,}} & \\
\text{exam(E) \text{ wrt School.}} & \\
\text{user(X) \leftarrow other_user(X)} & \\
\end{align*}
\]

By using the \texttt{wrt} construct, we define the users of the library. A user can be either a student of the School (first rule defining predicate user) or a professor teaching
a course of the School (second rule defining predicate user) or a person recorded as user inside the database of the library itself (third rule defining predicate user).

This language has been shown to be particularly suited for semantic integration purposes [97] and, in particular, it has been used to write mediators. In Chapter 7 we will deepen this peculiarity.
Chapter 5

Time in a Multi-Theory Framework

This chapter contains the description of our framework for temporal reasoning. The leading motivation of our work on time is the search for an extension of the multi-theory framework, presented in the previous chapter, capable of modeling and reasoning on temporal data. We believe that such a framework, adequately enriched, is a good candidate to fulfill the requirements of the thesis. Indeed it already supports semantic integration but it completely lacks of mechanisms to represent spatial and temporal information and to express spatio-temporal correlations.

We will consider a subset of the language of program expressions: only union $\cup$ and intersection $\cap$ are provided as composition operations. This choice is motivated by the fact that union and intersection are powerful enough to express the applications we are interested in. We leave as future work the investigation on the remaining operations. Moreover, the $\texttt{wrt}$ construct will be introduced only in Chapter 7 where we will define the formalism which allows us to support both interoperability and spatio-temporal reasoning.

We start presenting the approach proposed in [84, 85] which extends the language of program expressions by associating temporal information with clauses (Section 5.1). An object level program is still a named collection of clauses, but each clause is now annotated by a time interval representing the period of time in which the clause holds, i.e., a clause is of the form $A \leftarrow B_1, \ldots, B_n \square [a, b]$. The handling of time is hidden at the object level and it is managed at the meta-level by intersecting the intervals associated with clauses. However, this approach is not completely satisfactory. First, it does not offer any mechanism for modeling indefinite temporal information and periodic data. Moreover, some problems arise when at the object level we want to extract temporal information from the intervals.

To overcome these deficiencies, we enrich the language of program expressions by adopting Temporal Annotated Constraint Logic Programming (TACLP) to describe object level programs and we adequately define the composition operators
over annotated constraint logic programs. TACL, introduced in [48, 49, 50], supports qualitative and quantitative (metric) temporal reasoning involving both time points and time periods (time intervals) and their duration (Section 5.2). Moreover, it allows one to represent definite, indefinite and periodic temporal information.

Our resulting language [86], called MuTACL (Section 5.3), joins the advantages of TACL in handling temporal information with the ability to structure and compose programs coming from the language of program expressions. Moreover, the use of constraints enables efficient implementations and the representation of spatial data, which is the topic of the next chapter. In Section 5.4 we show the application of MuTACL to represent knowledge in the field of regulations and Section 5.5 studies the relationship between our two approaches for handling temporal information. Finally, in Section 5.6 we outline the main results of the chapter and we compare MuTACL with the logic-based proposals in literature dealing with time.

5.1 Time associated with clauses

There are several alternatives to introduce time into our multi-program setting. For instance we can associate a temporal entity with atoms or with clauses, we can use time intervals or time points, discrete or continuous time etc. However, whatever choice we will take, the objective is to obtain a model that can be embedded in a natural way into the (constraint) logic programming paradigm and programming style.

We are going to present two approaches: the first one [84, 85] annotates clauses with time intervals, the other one [86], which extends the former, annotates atoms with temporal information. Let us start with the approach described in [84, 85].

To begin, we characterize the structure of time we adopt in our framework. Time is discrete. Time points are totally ordered by the relation $\leq$. We call the set of time points $TP$. We assume that the time-line is left-bounded by the number 0 and open to the future, with the symbol $\infty$ used to denote a time point that is later than any other. A time period is an interval $[r, s]$ with $0 \leq r \leq s \leq \infty$ and $r \in TP$, $s \in TP$, representing the convex, non-empty set of time points $\{t \mid r \leq t \leq s\}$. Thus the interval $[0, \infty]$ denotes the whole time line. Time points and the order relation $\leq$ between them are axiomatized as follows:

\[
\begin{align*}
\text{timepoint}(0), & \quad X \leq X \leftarrow \text{timepoint}(X) \\
\text{timepoint}(s(X)) \leftarrow \text{timepoint}(X) & \quad X \leq Y \leftarrow X < Y \\
X < \infty & \leftarrow \text{timepoint}(X) \quad X > Y \leftarrow Y < X \\
0 < s(X) & \leftarrow \text{timepoint}(X) \quad X \geq Y \leftarrow Y \leq X \\
s(X) < s(Y) & \leftarrow X < Y
\end{align*}
\]

For the sake of clarity, in the following we will use more natural representations of time, i.e., dates, natural numbers etc. Their mapping into time points is straightforward.
Now we provide the set of time periods with a partial order relation, the subinterval relation, and with a binary operation, which, given two time periods, returns the interval denoting their intersection. Both the subinterval relation and the intersection operation will be used only at the meta-level in order to define the semantics of program expressions.

The subinterval relation between time intervals is denoted by $\subseteq$ and is defined as

$$[X, Y] \subseteq [W, Z] \iff nempty([X, Y]), nempty([W, Z]), W \leq X, Y \leq Z$$

where the predicate $nempty$ states that an interval is not empty and it is defined as

$$nempty([X, Y]) \iff X \leq Y.$$

Intersection of time intervals is denoted by $\cap$ and it is axiomatized as follows:

$$[X, Y] \cap [W, Z] = [W, Y] \iff nempty([X, Y]), nempty([W, Z]), X \leq W, W \leq Y, Y \leq Z$$

$$[X, Y] \cap [W, Z] = [W, Z] \iff nempty([X, Y]), nempty([W, Z]), X \leq W, Z < Y$$

$$[X, Y] \cap [W, Z] = [X, Z] \iff nempty([X, Y]), nempty([W, Z]), W \leq X, X \leq Z, Z \leq Y$$

$$[X, Y] \cap [W, Z] = [X, Y] \iff nempty([X, Y]), nempty([W, Z]), W \leq X, Y < Z$$

Notice that intersection is defined only on overlapping intervals. Therefore when we consider an interval obtained by intersecting different intervals, i.e., $\bigcap_{i=1}^n I_i$, we implicitly mean it is not empty.

Now we can describe how the multi-theory framework is extended to deal with temporal information.

As anticipated, we consider only two meta-level operations for composing logic programs, hence the language of program expressions is restricted to:

$$Exp ::= Pname \mid Exp \cup Exp \mid Exp \cap Exp$$

Moreover, object level programs are now collections of annotated clauses named by a constant symbol belonging to $Pname$. An annotated clause is a clause equipped with a time interval representing the period of time in which the clause holds. At the object level, an annotated clause looks like

$$A \leftarrow B_1, \ldots, B_n \sqcap [a, b].$$

where $A$, $B_1, \ldots, B_n$ are atoms and $[a, b]$ is a ground time period. Clauses without time periods are supposed to be annotated with $[0, \infty]$.

In a database perspective, such time intervals represent valid time and our temporal model can be classified as historical in the taxonomy of [103]. In this way we record the history of the domain and of the rules and we can query the database at a certain date in the past in order to obtain the information that held in that period according to rules in force at that time.
Example 9 We want to add some temporal information to Example 7. Suppose that a movie ticket is $5 for kids and $7 for adults in the first six months of 1996 and $7 for everybody in the rest of the year. The program BoxOffice becomes

BoxOffice:

\[
\begin{align*}
ticket-cost(5, X) & \leftarrow \\
age(X, Y), \text{lesseq}(Y, 16) & \square \\
[jan 1 1996, jun 30 1996] & \end{align*}
\]

\[
\begin{align*}
ticket-cost(7, X) & \leftarrow \\
age(X, Y), \text{greaterthan}(Y, 16) & \square \\
[jan 1 1996, jun 30 1996] & \end{align*}
\]

\[
\begin{align*}
ticket-cost(7, X) & \leftarrow \\
age(X, Y) & \square \\
[jul 1 1996, dec 31 1996] & \end{align*}
\]

\[
\begin{align*}
today(D1), \text{born}(X, D2), & \\
\text{year-diff}(D2, D1, Y) & \square \\
[0, \infty] & \end{align*}
\]

Every clause is annotated with the interval specifying when such rule holds. For instance the first two clauses are applicable in the first six months of the year, whereas the last clause for ticket-cost represents the fact that the ticket is $7 in the second part of the year, regardless of the customer’s age. Programs Today and Tom with the adequate time intervals are defined as follows:

**Today:**

\[
today(\text{may 28 1996}) \leftarrow \square [\text{may 28 1996}, \text{may 28 1996}] 
\]

**Tom:**

\[
\text{born}(\text{tom, may 7 1981}) \leftarrow \square [\text{may 7 1981}, \infty] 
\]

Notice that the valid time associated with the clause of the program Today is just the current day May 28 1996.

If we combine the previous theories by means of the union operator $\cup$, as we will see in the next section, we discover that Tom has to pay $5: only ticket-cost(5, \text{tom}) is derivable from the program expression

\[
\text{BoxOffice} \cup \text{Tom} \cup \text{Today}
\]

since Tom is fifteen years old on May 28 1996. The time period associated with the clause defining the predicate today is fundamental to select the right clause to solve the query ticket-cost(5, tom).

### 5.1.1 Semantics

In this section we extend the meta-interpreter and the bottom-up semantics of the language of program expressions presented in Section 4.2 in order to capture the temporal features we have just introduced.
Meta-interpreter

In order to take into account time intervals, both the predicates demo and clause have now an extra-argument denoting a time period: \( \text{demo} (\mathcal{E}, G, I) \) means that the goal \( G \) is provable in the program expression \( \mathcal{E} \) and holds within the time period \( I \), and \( \text{clause} (\mathcal{E}, A, G, I) \) means that the clause \( A \leftarrow G \) holds in \( I \) and is derivable from the program expression \( \mathcal{E} \). The meta-interpreter is defined by the following clauses.

\[
\text{demo} (\mathcal{E}, \text{empty}, [0, \infty]). \tag{5.2}
\]

\[
demo(\mathcal{E}, (G_1, G_2), I) \leftarrow \text{demo}(\mathcal{E}, G_1, K), \text{demo}(\mathcal{E}, G_2, J), \quad K \wedge J = H, I \in H \tag{5.3}
\]

\[
demo(\mathcal{E}, A, I) \leftarrow \text{clause}(\mathcal{E}, A, G, K), \text{demo}(\mathcal{E}, G, J), \quad K \wedge J = H, I \in H \tag{5.4}
\]

\[
\text{clause}(\mathcal{E}_1 \cup \mathcal{E}_2, A, G, I) \leftarrow \text{clause}(\mathcal{E}_1, A, G, I) \tag{5.5}
\]

\[
\text{clause}(\mathcal{E}_1 \cup \mathcal{E}_2, A, G, I) \leftarrow \text{clause}(\mathcal{E}_2, A, G, I) \tag{5.6}
\]

\[
\text{clause}(\mathcal{E}_1 \cap \mathcal{E}_2, A, (G_1, G_2), I) \leftarrow \text{clause}(\mathcal{E}_1, A, G_1, K), \text{clause}(\mathcal{E}_2, A, G_2, J), \quad K \wedge J = I \tag{5.7}
\]

A clause \( A \leftarrow B \parr K \) of a plain program \( P \) is now represented at the meta-level by

\[
\text{clause}(P, A, B, K) \leftarrow \text{empty}(K) \tag{5.8}
\]

At the object level, atoms and sequences of atoms are without explicit temporal references, at the meta-level, instead, the demo predicate provides them with a time interval expressing when they hold. Thus clause (5.2) states that the constant empty is solvable in any program expression \( \mathcal{E} \) and it holds throughout the whole time line. Clause (5.3) states that a conjunction of atoms is provable in a program expression \( \mathcal{E} \) and holds in an interval \( I \) if the conjuncts are provable in \( \mathcal{E} \) and hold in intervals \( K \) and \( J \) respectively which are overlapping and their intersection contains \( I \). Similarly, an atom \( A \), provable in \( \mathcal{E} \), holds within an interval \( I \) if there exists a clause in \( \mathcal{E} \), holding in a certain interval \( K \), and the clause body is solvable in \( \mathcal{E} \) and holds in an interval \( J \) which overlaps \( K \). The interval \( I \) is a subinterval of the non-empty intersection between \( K \) and \( J \).

On the other hand, time is explicit in clauses from object level programs but it has to be computed if we consider clauses from composition of programs, i.e., from program expressions. Clauses (5.5) and (5.6) state that a clause holding in
$I$ belongs to the union of two program expressions if it belongs, holding in $I$, to either $\mathcal{E}_1$ or $\mathcal{E}_2$. This is basically how union is computed in the original language of program expressions (see clauses (4.4) and (4.5)). Finally, clause (5.7) builds the clause belonging to the intersection of two program expressions as the corresponding clause (4.6) does, but there is a further condition: the intervals, in which the clauses from $\mathcal{E}_1$ and $\mathcal{E}_2$ hold, have to overlap, and their non-empty intersection is the time period in which the clause of the intersection holds.

**Bottom-up semantics**

Meta-logic provides a semantics to program expressions extended with time intervals in the sense that its axioms tell us how to compute goals of the form

$$\text{demo}(\mathcal{E}, G, I)$$

The meta-logical axioms define a new SLD procedure, capable of handling program expressions and time intervals, in terms of the basic SLD of logic programming.

On the other hand, in order to give a bottom-up semantics, we follow the idea of Mancarella and Pedreschi [83] thus, rather than characterizing programs in terms of interpretations (first order), programs are denoted by functions (the immediate consequence operator) from interpretations into interpretations (higher order). Now interpretations are made up of atom-interval pairs. Hence the immediate consequence operator $T^{\text{int}}$ is a function:

$$T^{\text{int}} : Exp \rightarrow \wp(B_L \times Int) \rightarrow \wp(B_L \times Int),$$

where $Int$ is the set of closed or right unlimited intervals of natural numbers, i.e.,

$$\text{Int} = \{ A \in \wp(\mathbb{N}) \mid \exists a, b \in \mathbb{N}, a \leq b \land \forall x. x \in A \iff a \leq x \leq b \} \lor \exists a \in \mathbb{N}, \forall x. x \in A \iff a \leq x \}.$$

In the following, elements of $Int$ will be simply called *intervals*. It is worth noting that an interval is never empty and that the intersection of intervals is an interval if and only if it is not empty.

To be rigorous, we should use different notations for terms denoting intervals in the language (syntactic intervals), and sets in the model (semantic intervals), but this would make the notation heavy and cumbersome. Actually, it is always clear from the context whether we are referring to syntax or to semantics.

Moreover, notice that the set $Int$ with the operations $\subseteq$ and $\cap$ is a model of the theory defining the axiomatization of the time points and of the order relation between points, the subinterval relation and the intersection operation between syntactic intervals. In other words, as one can trivially prove, for any intervals $I, J, K$

$$I \subseteq J \text{ is provable } \iff I \subseteq J \quad (5.9)$$

$$J \cap K = I \text{ is provable } \iff J \cap K = I. \quad (5.10)$$
It is worth recalling that in both statements above \( I \neq \emptyset \) since \( I \) is an interval.

Given a program expression \( \mathcal{E} \), its bottom-up semantics is the immediate consequence operator \( T^\text{int}_\mathcal{E} \), which is compositionally defined in terms of its sub-expressions.

**Definition 2 (Bottom-up semantics)** Let \( \mathcal{E} \) be a program expression, the function \( T^\text{int}_\mathcal{E} : \wp(\mathcal{B}_L \times \text{Int}) \rightarrow \wp(\mathcal{B}_L \times \text{Int}) \) is defined as follows.

- \( \mathcal{E} \) is a plain program \( P \)
  
  \[
  T^\text{int}_P(I) = \left\{ (A, I) \mid \begin{array}{l}
  A \leftarrow B_1, \ldots, B_n \sqcap I_0 \in \text{ground}_L(P),
  
  I = \bigcap_{i=0}^n I_i, \ I \neq \emptyset
  \end{array}\right\}
  \]

- \( \mathcal{E} = \mathcal{E}_1 \cup \mathcal{E}_2 \)
  
  \[
  T^\text{int}_{\mathcal{E}_1 \cup \mathcal{E}_2}(I) = T^\text{int}_{\mathcal{E}_1}(I) \cup T^\text{int}_{\mathcal{E}_2}(I)
  \]

- \( \mathcal{E} = \mathcal{E}_1 \cap \mathcal{E}_2 \)
  
  \[
  T^\text{int}_{\mathcal{E}_1 \cap \mathcal{E}_2}(I) = T^\text{int}_{\mathcal{E}_1}(I) \cap T^\text{int}_{\mathcal{E}_2}(I)
  \]

where \( I_1 \cap I_2 = \{(A, I) \mid (A, I_1) \in I_1, (A, I_2) \in I_2 : I = I_1 \land I_2, \ I \neq \emptyset\} \)

The definition for plain programs differs from the standard logic programming operator just for the conditions on intervals: an atom \( A \) holds in an interval \( I \), if there exists a clause, holding in a certain interval \( I_0 \), whose head is \( A \), the atoms of the body are solvable and hold in intervals whose intersection with \( I_0 \) is \( I \). The set of immediate consequences of a union of program expressions is the set-theoretic union of the immediate consequences of each program expression. Finally, the set of immediate consequences for the intersection of program expressions consist of atoms, which are consequences of both program expressions and hold in the (non-empty) interval which is the intersection of the intervals associated to the consequences.

It is easy to prove that \( T^\text{int}_\mathcal{E} \) is continuous on the lattice \( \wp(\mathcal{B}_L \times \text{Int}) \). Hence we can define the least fixpoint semantics of a program expression \( \mathcal{E} \) as the least fixpoint of \( T^\text{int}_\mathcal{E} \) which is \( (T^\text{int}_\mathcal{E})^\omega = \bigcup_{i \in \mathbb{N}} (T^\text{int}_\mathcal{E})^i \).

To conclude, we can prove the following result on soundness and completeness of the meta-interpreter with respect to the least fixpoint semantics.

**Theorem 3 (Soundness and completeness)** Let \( \mathcal{E} \) be a program expression and \( V \) be the meta-program containing the meta-level representation of the object level programs occurring in \( \mathcal{E} \), the axiomatization of the order relation between time points, the subinterval relation \( \ll \) and the intersection operation \( \sqcap \) between intervals and the meta-program consisting of the clauses (5.2)–(5.7). For any object level atomic formula \( A \) and any ground interval \( I \), the following statements hold:

1. \( \text{demo}(\mathcal{E}, A, I) \in T^\omega_V \implies \exists H : (A, H) \in (T^\text{int}_\mathcal{E})^\omega \land I \subseteq H. \)

2. \( (A, I) \in (T^\text{int}_\mathcal{E})^\omega \implies \text{demo}(\mathcal{E}, A, I) \in T^\omega_V. \)
The proof is reported in Appendix A.1.

Theorem 3 shows that a program expression $\mathcal{E}$ and its corresponding metaprogram $V$ have a comparable deductive capability with respect to object level goals. However, the meta-interpreter allows one to prove atoms on smaller intervals than the associated program expression $\mathcal{E}$ does. For instance, consider the program $P$ consisting of the clause

\[ p(a) \leftarrow \square [0,10]. \]

The least fixpoint semantics is $\langle T_P^{int} \rangle^\omega = \{(p(a), \{0, 1, 2, \ldots, 10\})\}$. On the other hand $T_P^\omega \supseteq \{\text{demo}(P, p(a), I) \mid I \subset [0, 10]\}$.

**Remark**

We presented only a simplified version of the top-down and bottom-up semantics which does not take into account the possibility of reasoning over overlapping intervals. For instance, suppose that we have the following program $P$

\[ q \leftarrow \square [0,10]. \quad q \leftarrow \square [10,20]. \]

We can prove both $q$ holds in $[0,10]$ and $q$ holds in $[10,20]$ but we cannot prove that $q$ holds in $[0,20]$, i.e., in the union of two overlapping intervals.

In [85] we showed how it is possible to extend the previously described meta-interpreter and the bottom-up semantics to support this kind of reasoning. Since the language of annotated clauses is only a first attempt to handle temporal information inside the multi-theory framework and it is not our final proposal, we prefer not to introduce the extension in [85] in order to avoid useless complications.

### 5.1.2 An example

Throughout the thesis, for the sake of clarity, we write object programs as named collections of object clauses instead of using the clumsier meta-level representation.

**Buy & sell subsidized houses**

According to the law in force since January 10 1980, a person can buy a subsidized house if either this house has been assigned to him/her or he/she lives with a person who has this house and he/she agrees to sell it.

The program `LEGAL-BUYER` contains the features of the person who can buy a house.

**LEGAL-BUYER:**

\[
\begin{align*}
\text{can-buy}(X, \text{Home}) & \leftarrow \\
\text{assignee}(X, \text{Home}) & \square \\
\text{[jan 10 1980, } & \infty] \\
\text{can-buy}(X, \text{Home}) & \leftarrow \\
\text{assignee}(Y, \text{Home}), \\
\text{lives-with}(X, Y, \text{Home}), \\
\text{agrees}(X, Y, \text{Home}) & \square \\
\text{[jan 10 1980, } & \infty]
\end{align*}
\]
After July 7, 1993 to buy a subsidized house, it is additionally necessary that the buyer lives in the house for more than five years.

The program CONSTR contains the further constraint imposed after July 7, 1993.

\[
\text{CONSTR:}
\begin{align*}
\text{can-buy}(X,\text{Home}) & \leftarrow \\
\text{filed-request}(X,\text{Home},D1), \\
\text{lives-in-since}(X,\text{Home},D2), \\
\text{year-diff}(D2,D1,\Delta), \\
\text{greaterthan}(\Delta, 5) & \square \\
\text{[jul 7 1993, }\infty]\}
\end{align*}
\]

The program expression \(\text{LEGAL-BUYER} \cap \text{CONSTR}\) allows us to represent the regulation after July 7, 1993. In fact it behaves as a program having two clauses defining the predicate \text{can-buy}. Such clauses are obtained by adding to the \text{LEGAL-BUYER} clauses the body of the \text{CONSTR} clause. Moreover, the clauses hold in an interval which is the intersection between \([\text{jan 10 1980, }\infty]\) and \([\text{jul 7 1993, }\infty]\), thus since July 7, 1993 onwards.

Now, we consider a person who wants to buy a subsidized house.

\[
\text{LUIGI:}
\begin{align*}
\text{assignee}(\text{luigi,h26}) & \leftarrow \square [\text{jan 1 1987, }\infty] \\
\text{lives-in-since}(\text{luigi,h26,jan 8 1988}) & \leftarrow \square [\text{jan 8 1988, }\infty]
\end{align*}
\]

The program \text{APPL283} is formed by a single clause that records the request.

\[
\text{APPL283:}
\begin{align*}
\text{filed-request}(\text{luigi,h26, feb 2 1995}) & \leftarrow \square [\text{feb 2 1995, }\infty]
\end{align*}
\]

To know whether Luigi can buy the house \(h26\) according to the last regulation, we ask the query

\[
\text{demo}((\text{LEGAL-BUYER} \cap \text{CONSTR}) \cup \text{APPL283}) \cup \text{LUIGI, can-buy(luigi,h26),}\_)
\]

The answer is yes, Luigi can buy the house \(h26\) since on February 2 1995 he has been living in the house for more than five years.

### 5.1.3 Derived operators on time intervals

In order to make easier the selection of information holding in a certain interval we introduce the following operation.

**Definition 3** Let \(P\) be a plain program and \(I\) be a ground interval.

\[
P \downarrow I = P \cap 1^I_P,
\]

where \(1^I_P\) is a program defined as follows:
for all $p$ defined in $P$ with arity $n$

\[ p(X_1, \ldots, X_n) \leftarrow \Box I. \]

This operation exploits the features of the intersection operator: what we do is to select only the clauses belonging to $P$ that hold in $I$ or in a subinterval of $I$ and we restrict their validity time to such an interval. Therefore $\downarrow$ allows us to create temporal views of programs, for instance $P \downarrow [t, t]$ is the program $P$ at time point $t$. This operation acts as a valid-timeslice operator in the field of databases (see the glossary in [45]).

Consider the example in the previous section. We can represent the whole history of the regulation concerning “Buy & Sell subsidized houses” by using the following program expression

\[(\text{LEGAL-buyer} \downarrow [0, \text{Jul 6 1993}]) \cup (\text{LEGAL-buyer} \cap \text{Constr})\]

Thanks to the $\downarrow$ operation we restrict the validity of the clauses belonging to LEGAL-buyer from January 10, 1980 up to July 6, 1993, thus modeling the law before July 7, 1993. On the other hand, the program expression LEGAL-buyer $\cap$ Constr expresses the regulation in force since July 7, 1993, as we previously explained.

This example suggests that the $\downarrow$ operation is useful to model updates. Suppose that we want to represent the fact that Frank is a researcher in mathematics, then he is promoted and becomes an assistant professor. In our formalism we define a program Frank that records the information associated to Frank as researcher.

**Frank:**

\[\text{researcher(maths)} \leftarrow \Box [\text{Mar 8 1993}, \infty]\]

On March 1996 Frank became an assistant professor. In order to modify the information contained in the program Frank, we build this program expression:

\[(\text{Frank} \downarrow [0, \text{Feb 29 1996}]) \cup \{\text{assistant_professor(maths)} \leftarrow \Box [\text{Mar 1 1996}, \infty]\}\]

The second expression is an unnamed theory. Unnamed theories are represented by the following meta-level clause:

\[\text{clause}([X \leftarrow Y \Box I], X, Y, I) \leftarrow \text{nempty}(I)\]

This update is similar to add and delete a ground atom. For instance in $\mathcal{LDC}$ and [119] we can express such a change by solving the goal $-\text{researcher(maths)}, +\text{assistant_professor(maths)}$.

The advantage of our approach is that we do not change directly the clauses of the program Frank but we compose the old theory with a new one that represents the current situation. Therefore we preserve even the state of the database before March 1, 1996, keeping faith to the historical dealing of information.
\[ \text{demo}((\text{Frank} \downarrow [0, \text{feb 29 1996}]) \cup \\
\{\text{assistant, professor}(\text{maths}) \leftarrow \Box [\text{mar 1 1996, } \infty]\}, \\
\text{researcher}(X), [\text{feb 23 1994}, \text{feb 23 1994}]) \\
X = \text{maths} \]

\[ \text{demo}((\text{Frank} \downarrow [0, \text{feb 29 1996}]) \cup \\
\{\text{assistant, professor}(\text{maths}) \leftarrow \Box [\text{mar 1 1996, } \infty]\}, \\
\text{researcher}(X), [\text{mar 12 1996, mar 12 1996}]) \\
\text{no.} \]

The first query inquires the updated database before the advance in career of Frank, on the contrary, the second one shows how information in the database has been modified.

### 5.1.4 Discussion

The distinctive features of this approach are the context, that is a multi-theory framework, and the fact that temporal information is attached to clauses. Here we analyze only the second aspect and we postpone the discussion on the multi-theory setting to Section 5.6, because it is also a peculiarity of MuTACLP, our final proposal for dealing with time.

Among the approaches to handling time by logic, briefly described in Section 3.1.1, Hrycej's Temporal Prolog [62] is the closest language to our proposal. Temporal information are explicitly associated with clauses by using the construct \( P \in T \), meaning that \( P \), a (unit) clause, holds during the time interval \( T \). Moreover, a first-order “reified” logic has been taken as a basis for the implementation. The predicate \( HOLD \) is used as meta-level predicate: \( HOLD(A, T) \) means that the object-level atom \( A \) is true in the time interval \( T \). A new resolution strategy, which the author calls temporal modus ponens, is defined. It acts as our extended SLD resolution: an atom \( A \) holds within an interval \( T \) if there exists a clause, holding in a time interval \( R \), whose head is unifiable to \( A \), whose body is solvable and holds in \( S \), and the time interval \( T \) is a subinterval of both \( R \) and \( S \). Hence by using the meta-level representation one can deduce temporal information also for atoms as well as we do. However, there are several differences: first of all, the time-point-oriented model where time points are totally ordered, discrete and infinite, and time intervals are convex set of time points, which coincides with our structure of time, is only an intermediate step to obtain an interval-based model where time intervals are considered as primitives and the usual Allen’s relations can be defined between them. Indeed Hrycej in this approach is only interested in qualitative temporal relations. Finally, Temporal Prolog deals with negation whereas we do not tackle it but it does not support any mechanism to structure and combine programs as we do.

Summing up, the language we presented in Section 5.1 is particularly well-suited
to model time-varying set of rules (see [105]). In fact the time interval we attach to clauses is a means to change dynamically, i.e., in a time dependent way, the set of clauses used to solve a query, as we pointed out in Sections 5.1.2 and 5.1.3. This is an attractive feature in several application fields such as legal reasoning where regulations have a validity for a fixed period of time, thus time is fundamental to select the regulation in force in a certain date. Nevertheless, this approach is not completely satisfactory. First, it does not offer any mechanism for modeling indefinite temporal information and periodic data. Moreover, some problems arise when at the object level we want to extract temporal information from the intervals. For instance, to model the fact that John was born on February 10 1974 we can define a clause \( \text{born}(\text{john}) \leftarrow \square [\text{Feb } 10 \ 1974, \infty] \). But then there is no way to extract John’s date of birth at the object level. To do this we have to explicitly duplicate this temporal information as in \( \text{born}(\text{john, Feb } 10 \ 1974) \leftarrow \square [\text{Feb } 10 \ 1974, \infty] \). Notice however that in this way the interaction between object level and meta-level temporal information is somehow uncontrolled. For instance, one can specify incoherent information such as \( \text{born}(\text{john, Feb } 10 \ 1974) \leftarrow \square [\text{Jan } 1 \ 1900, \infty] \).

To overcome these deficiencies, we enrich the language of program expressions by adopting Temporal Annotated Constraint Logic Programming (TACL) [50] to describe object level programs and we show that the program composition operations, union and intersection, can be smoothly extended to cope with temporal annotated constraint theories, i.e., TACL programs, by exploiting the lattice structure of temporal annotations. The resulting language, called MuTACL, is an extension of the language for dealing with temporal information proposed in this section as it will be shown in Section 5.5.

## 5.2 Temporal Annotated Constraint Logic Programming

Temporal Annotated Constraint Logic Programming (TACL) proposed by Frühwirth [48, 49, 50] has been shown to be a natural and powerful way of formalizing temporal information and reasoning. In particular, the pieces of temporal information are given by \textit{temporal annotations} which say at what time(s) the formula to which they are attached is valid. The annotations of TACL make time explicit but avoid the proliferation of temporal variables and quantifiers of the first-order approach. In this way, TACL supports qualitative and quantitative temporal reasoning and allows one to represent definite, indefinite and periodic temporal information and to work both with time points and time periods (time intervals).

In [50] TACL is presented as an instance of annotated constraint logic (ACL) for reasoning about time. ACL is a generalization of generalized annotated programs [70, 80], and extends first-order languages with a distinguished class of predicates, called \textit{constraints}, and a distinguished class of terms, called \textit{annotations}, used to
label formulae. Moreover ACL provides inference rules for annotated formulae and a constraint theory for handling annotations. One advantage of a language in the ACL framework is that its clausal fragment can be efficiently implemented: given a logic in this framework, there is a systematic way to make a clausal fragment executable as a constraint logic program. Both an interpreter and a compiler can be generated and implemented in standard constraint logic programming languages.

The TACL language is given two different kinds of semantics, an operational one based on meta-logic (top-down semantics) using a meta-interpreter and a fixpoint one obtained by extending the definition of the immediate consequence operator of CLP to deal with annotated atoms (bottom-up semantics). The meta-interpreter is proved to be sound and complete with respect to the bottom-up semantics. While top-down semantics are known from the early work on TACL [48, 49, 50], the bottom-up semantics has been studied only recently by Raffætà and Frühwirth [96, 95] and, consequently, for the first time soundness and completeness results for TACL are provided.

5.2.1 The language

This subsection briefly reviews TACL. TACL is a constraint logic programming language where formulae can be annotated with temporal labels and where temporal constraints express relations between these labels. In TACL, the choice of the temporal ontology is free. In this thesis, we consider the subset of TACL, where time points are totally ordered, and sets of time points are convex and non-empty. Moreover only atomic formulae can be annotated and clauses are free of negation. For a more detailed treatment of TACL and for the general theory of ACL we refer the reader to [50]. With an abuse of notation, in the rest of the thesis we call TACL such a subset of the language.

Time can be discrete or dense. Time points are totally ordered by the relation $\leq$. We call the set of time points $D$ and we suppose that a set of operations (such as the binary operations $+$, $-$) to manage such points are associated with it. We assume that the time-line is left-bounded by the number 0 and open to the future, with the symbol $\infty$ used to denote a time point that is later than any other. A time period is an interval $[r, s]$ with $0 \leq r \leq s \leq \infty, r \in D, s \in D$ that represents the convex, non-empty set of time points $\{t \mid r \leq t \leq s\}$. Thus the interval $[0, \infty]$ denotes the whole time line.

An annotated formula is of the form $A \alpha$ where $A$ is an atomic formula and $\alpha$ an annotation. In TACL, there are three kinds of annotations based on time points and on time periods. Let $t$ be a time point and let $I = [r, s]$ be a time period.

(at) The annotated formula $A \text{at } t$ means that $A$ holds at time point $t$.

\[^1\text{The results we present naturally extend to time lines that are bounded or unbounded in other ways and to time periods that are open on one or both sides.}\]
(th) The annotated formula \( A \text{th} I \) means that \( A \) holds throughout, i.e., at every time point in the time period \( I \). The definition of a \( \text{th} \)-annotated formula in terms of \( \text{at} \) is:

\[
A \text{th} I \iff \forall t \in I \rightarrow A \text{at} t.
\]

(in) The annotated formula \( A \text{in} I \) means that \( A \) holds at some time point(s) - but we do not know exactly which - in the time period \( I \). The definition of an \( \text{in} \)-annotated formula in terms of \( \text{at} \) is:

\[
A \text{in} I \iff \exists t \in I \wedge A \text{at} t.
\]

The \( \text{in} \) temporal annotation accounts for indefinite temporal information.

The set of annotations is endowed with a partial order relation \( \sqsubseteq \) which turns it into a lattice. Given two annotations \( \alpha \) and \( \beta \), the intuition is that \( \alpha \sqsubseteq \beta \) if \( \alpha \) is “less informative” than \( \beta \) in the sense that for all formulae \( A \), \( A \beta \models A \alpha \). More precisely, being an instance of ACL, in addition to Modus Ponens, TACL has two further inference rules: the rule \((\sqsubseteq)\) and the rule \((\sqcup)\).

\[
\frac{A \alpha}{A \gamma} \quad \text{rule } (\sqsubseteq) \quad \frac{A \alpha}{A \gamma} = \alpha \sqcup \beta \quad \text{rule } (\sqcup)
\]

The rule \((\sqsubseteq)\) states that if a formula holds with some annotation, then it also holds with all annotations that are smaller according to the lattice ordering. The rule \((\sqcup)\) says that if a formula holds with some annotation and the same formula holds with another annotation then it holds with the least upper bound of the annotations.

Now, we define the constraint theory for temporal annotations. We recall that a constraint theory is a non-empty, consistent first order theory that axiomatizes the meaning of the constraints. First of all, our constraint theory includes an axiomatization of the total order relation \( \leq \) on time points \( D \). Then it contains the following axioms defining the partial order on temporal annotations.

\[
\begin{align*}
\text{(at th)} & \quad \text{at } t = \text{th} [t, t] \\
\text{(at in)} & \quad \text{at } t = \text{in} [t, t] \\
\text{(th } \sqsubseteq \text{)} & \quad \text{th} [s_1, s_2] \sqsubseteq \text{th} [r_1, r_2] \iff r_1 \leq s_1, s_1 \leq s_2, s_2 \leq r_2 \\
\text{(in } \sqsubseteq \text{)} & \quad \text{in} [r_1, r_2] \sqsubseteq \text{in} [s_1, s_2] \iff r_1 \leq s_1, s_1 \leq s_2, s_2 \leq r_2
\end{align*}
\]

The first two axioms state that \( \text{th} I \) and \( \text{in} I \) are equivalent to \( \text{at} t \) when the time period \( I \) consists of a single time point \( t \).\(^2\) Next, if a formula holds at every element of a time period, then it holds at every element in all sub-periods of that period \((\text{th } \sqsubseteq)\) axiom). On the other hand, if a formula holds at some points of a time period then it holds at some points in all periods that include this period \((\text{in } \sqsubseteq)\) axiom).

\(^2\)Especially in dense time, one may disallow singleton periods and drop the two axioms. This restriction has no effects on the results we are presenting.
A consequence of the above axioms is

\[(\text{in th } \subseteq) \quad \text{in}[s_1, s_2] \subseteq \text{th}[r_1, r_2] \iff s_1 \leq r_2, r_1 \leq s_2, s_1 \leq s_2, r_1 \leq r_2\]

i.e., an atom annotated by in holds in any time period that overlaps with a time period where the atom holds throughout.

To summarize the partial order relation on annotations, the axioms can be arranged in the following chain, assuming \(r_1 \leq s_1, s_1 \leq s_2, s_2 \leq r_2\):

\[
\text{in}[r_1, r_2] \subseteq \text{in}[s_1, s_2] \subseteq \text{in}[s_1, s_1] = \text{at } s_1 = \text{th } s_1, s_1 \subseteq \text{th } s_1, s_2 \subseteq \text{th } [r_1, r_2]
\]

Now we axiomatize the least upper bound \(\sqcup\) of temporal annotations over time points and time periods. As explained in [50], the least upper bound exists but sometimes may be “too large”. In fact, rule \((\sqcup)\) is correct only if the lattice order ensures that \(A \alpha \land A \beta \land (\gamma = \alpha \sqcup \beta) \implies A \gamma\) whereas, in general, this is not true in our case. For instance, according to the lattice, \(\text{th}[1, 2] \sqcup \text{th}[4, 5] = \text{th}[1, 5]\), but according to the definition of th-annotated formulae in terms of at, the conjunction \(A \text{th}[1, 2] \land A \text{th}[4, 5]\) does not imply \(A \text{th}[1, 5]\), since it does not express that \(A \text{at } 3\) holds. From a theoretical point of view, this problem can be overcome by enriching the lattice of annotations with expressions involving \(\sqcup\). In practice, it suffices to consider the least upper bound for time periods that produce another different meaningful time period. Concretely, we restrict ourselves to th annotations with overlapping time periods that do not include one another:

\[(\text{th } \sqcup) \quad \text{th } [s_1, s_2] \sqcup \text{th } [r_1, r_2] = \text{th } [s_1, r_2] \iff s_1 < r_1, r_1 \leq s_2, s_2 < r_2\]

Summarizing, we fix a constraint domain for time points where the signature includes suitable constants for time points, function symbols for operations on time points (e.g., +, −, . . .), and the predicate symbol \(\leq\), modeling the total order relation on time points. Then such constraint domain is extended to a constraint domain for handling annotations, denoted by \(\mathcal{A}\), by enriching the signature with function symbols \([\cdot], \text{at}, \text{th}, \text{in}, \sqcup, \sqcap\) and the predicate symbol \(\sqsubseteq\), axiomatized as described above and \(\sqcap\) is axiomatized in Section 5.3.1.

As for traditional constraint logic programming, a TACL language is determined by fixing a constraint domain \(\mathcal{C}\), which, as one would expect, is required to contain the constraint domain \(\mathcal{A}\) for annotations. We denote by TACL(\(\mathcal{C}\)) the TACL language based on \(\mathcal{C}\). To lighten the notation, in the following we omit \(\mathcal{C}\).

Our results are parametric with respect to a language \(L = (\Sigma_L, \Pi_L)\) in which all programs and queries under consideration are included. The signature \(\Sigma_L\) coincides with the set of function symbols in \(\Sigma_C\), while \(\Pi_L\) includes the predicate symbols of \(\Pi_C\).

We can now define the clausal fragment of TACL that can be used as an efficient temporal programming language.
Definition 4 A TACL P clause is of the form:
\[ A \alpha \leftarrow C_1, \ldots, C_n, B_1 \alpha_1, \ldots, B_m \alpha_m \quad (n, m \geq 0) \]
where \( A \) is an atom (not a constraint), \( \alpha \) and \( \alpha_i \) are (optional) temporal annotations, the \( C_j \)'s are the constraints and the \( B_i \)'s are atomic formulae. Constraints \( C_j \) cannot be annotated.

A TACL P program is a finite set of TACL P clauses.

We remark that constraints and (annotated) atoms may appear in any order inside the body of a clause. In the definition we write them into separate groups only for notational convenience.

We conclude the introduction of TACL P with some examples.

Example 10 In a company, there are managers and a secretary who has to manage their meetings. A manager is busy if he is in a meeting or if he is out.

\[
\begin{align*}
\text{busy}(X) \text{ th} [T_1, T_2] & \leftarrow \text{in-meeting}(X) \text{ th} [T_1, T_2] \\
\text{busy}(X) \text{ th} [T_1, T_2] & \leftarrow \text{out-of-office}(X) \text{ th} [T_1, T_2]
\end{align*}
\]

Suppose the schedule for today to be the following: Smith and Jones have a meeting at 9am and at 9:30am respectively, each lasting one hour. In the afternoon Smith goes out for lunch at 2pm and comes back at 3pm:

\[
\begin{align*}
\text{in-meeting}(\text{smith}) \text{ th} [9am, 10am]. \\
\text{in-meeting}(\text{jones}) \text{ th} [9.30am, 10.30am]. \\
\text{out-of-office}(\text{smith}) \text{ th} [2pm, 3pm].
\end{align*}
\]

If the secretary wants to know whether Smith is busy between 9:30am and 10:30am she can ask for \( \text{busy(smith) in [9:30am, 10:30am]} \). Since Smith is in a meeting from 9am till 10am, one can indeed derive that Smith is busy. Notice that this query exploits indefinite information: since Smith is busy at least in one instant of the period [9:30am, 10:30am], the secretary cannot schedule an appointment for him for that period.

On the other hand, \( \text{busy(smith) th [9:30am, 10:30am]} \) does not hold, because Smith is not busy between 10am and 10:30am. Moreover, also \( \text{busy(smith) in [10:30am, 1:30pm]} \) does not hold, because Smith is not busy at all in that time period.

The query \( \text{busy(smith) th [T_1, T_2], busy(jones) th [T_1, T_2]} \) reveals that both managers are busy throughout the time period [9:30am, 10am], because this is the largest interval that is included in the time periods where both managers are busy.

Now assume that we define

\[
\text{busy th [T_1, T_2] \leftarrow busy(X) th [T_1, T_2]}
\]
Then busy holds when either manager is busy, namely for the time periods [9am, 10:30am] (which is the least upper bound of the time periods for the two overlapping meetings of Smith and Jones) and [2pm, 3pm].

In [102] TACL is successfully applied to a system for calculating the liquid flow in a network of water tanks from some events specifying when the taps were switched on and off. The following example involving continuous change is also presented.

**Example 11** We model information about the growth of trees.

1. Tree 1 sprouts at time 3.5 (the middle of year 3).

   \[ \text{sprouts(tree1) at 3.5.} \]

2. Tree 1 is an oak tree.

   \[ \text{tree\_type(tree1, oak).} \]

3. The growth rate of oak trees is 3 meters per year.

   \[ \text{growth\_rate(oak, 3).} \]

4. If a tree is of a type that has a given growth rate \( r \), and the tree sprouts at time \( s \) then at time \( t \) it has a height \( h \), where \( h = (t - s) \times r \).

   \[
   \text{height(Tree, H) at } T \leftarrow \\
   \text{tree\_type(Tree, Type),} \\
   \text{growth\_rate(Type, R),} \\
   \text{sprouts(Tree) at } S, \\
   H = (T - S) \times R
   \]

5. If a tree has height \( h \) meters at time \( t \), where \( h \geq 6.75 \), then it is mature.

   \[ \text{mature(Tree) th } [T, \infty] \leftarrow \text{height(Tree, H) at } T, H \geq 6.75 \]

In the last clause, the maturity of the tree at an instant is implied by a constraint on the height of the tree at that instant. Height is the continuously changing quantity. The query

\[ \text{mature(tree1) th } [6, 7] \]

can be proved. This means that tree1 is mature throughout the time period which begins at year 6 and ends at year 7.

The query

\[ \text{mature(tree1) th } [T_1, T_2] \]

yields \( T_1 \geq 5.75, T_2 = \infty \).
5.2.2 Semantics of TACLp

In this section we define the operational (top-down) semantics of the language TACLp by presenting a meta-interpreter for it. Then we provide TACLp with a fixpoint (bottom-up) semantics, based on the definition of an immediate consequence operator, which has been given only recently by Raffaetà and Führwirth in [96, 95].

In the definition of the semantics, without loss of generality, we assume all atoms to be annotated with \( \text{th} \) or \( \text{in} \) labels. In fact at \( t \) annotations can be replaced with \( \text{th} [t, t] \) by exploiting the (at \( \text{th} \)) axiom. Moreover, each atom which is not annotated in the object level program is intended to be true throughout the whole temporal domain, and thus can be labelled with \( \text{th} [0, \infty] \). Constraints stay unchanged.

Operational Semantics via Meta-Interpreter

The meta-interpreter for TACLp is defined by the following clauses:

\[
demo(\text{empty}). \tag{5.11}
\]

\[
demo((G_1, G_2)) \leftarrow demo(G_1), demo(G_2) \tag{5.12}
\]

\[
demo(A \text{ th} [T_1, T_2]) \leftarrow S_1 \leq T_1, T_2 \leq S_2, T_1 \leq T_2, \newline
\text{clause}(A \text{ th} [S_1, S_2], G), demo(G) \tag{5.13}
\]

\[
demo(A \text{ th} [T_1, T_2]) \leftarrow S_1 \leq T_1, T_1 < S_2, S_2 < T_2, \newline
\text{clause}(A \text{ th} [S_1, S_2], G), demo(G), demo(A \text{ th} [S_2, T_2]) \tag{5.14}
\]

\[
demo(A \text{ in} [T_1, T_2]) \leftarrow T_1 \leq S_2, S_1 \leq T_2, T_1 \leq T_2, \newline
\text{clause}(A \text{ th} [S_1, S_2], G), demo(G) \tag{5.15}
\]

\[
demo(A \text{ in} [T_1, T_2]) \leftarrow T_1 \leq S_1, S_2 \leq T_2, \newline\text{clause}(A \text{ in} [S_1, S_2], G), demo(G) \tag{5.16}
\]

\[
demo(C) \leftarrow \text{constraint}(C), C \tag{5.17}
\]

A clause \( A \alpha \leftarrow B \) of a TACLp program \( P \) is represented at the meta-level by

\[
\text{clause}(A \alpha, B) \leftarrow T_1 \leq T_2. \tag{5.18}
\]

where \( \alpha = \text{th} [T_1, T_2] \) or \( \alpha = \text{in} [T_1, T_2] \).
This meta-interpreter can be written in any CLP language that provides a suitable constraint solver for temporal annotations (see Section 5.2.1 for the constraint theory). Hence the first difference with the vanilla meta-interpreter is that our meta-interpreter handles constraints, which can either occur explicitly in its clauses, e.g. \( S_1 \leq T_1, T_1 \leq T_2, T_2 \leq S_2 \) in clause (5.13), or come from the resolution steps. The latter kind of constraints is managed by clause (5.17) which passes directly to the constraint solver each constraint \( C \) to be solved.

The second difference is that our meta-interpreter implements not only Modus Ponens but also the rule (\( \sqsubseteq \)) and the rule (\( \sqcup \)). This is the reason why the third \( demo \) clause of the vanilla meta-interpreter is now split into four clauses. Clauses (5.13), (5.15) and (5.16) implement the inference rule (\( \sqsubseteq \)): the atomic goal to be solved is required to be labelled with an annotation which is smaller than the one labelling the head of the clause used in the resolution step. For instance, clause (5.13) states that given a clause \( \text{Ath} [S_1, S_2] \leftarrow B \) whose body \( B \) is solvable, we can derive the atom \( A \) annotated with any \( \text{th} [T_1, T_2] \) such that \( \text{th} [T_1, T_2] \sqsubseteq \text{th} [S_1, S_2] \), i.e., according to axiom \( (\text{th} \sqsubseteq ) \), \( [T_1, T_2] \subseteq [S_1, S_2] \), as expressed by the constraint \( S_1 \leq T_1, T_2 \leq S_2, T_1 \leq T_2 \). Clauses (5.15) and (5.16) are built in an analogous way by exploiting axioms \( (\text{in th} \sqsubseteq ) \) and \( (\text{in} \sqsubseteq ) \), respectively.

Rule (\( \sqcup \)) is implemented by clause (5.14). According to the discussion in Section 5.2.1, it is applicable only to \( \text{th} \) annotations with overlapping time periods which do not include one another. More precisely, clause (5.14) states that if we can find a clause \( \text{Ath} [S_1, S_2] \leftarrow B \) such that the body \( B \) is solvable, and if moreover the atom \( A \) can be proved throughout the time period \( [S_2, T_2] \) (i.e., \( demo(A \text{th} [S_2, T_2]) \) is solvable) then we can derive the atom \( A \) labelled with any annotation \( \text{th} [T_1, T_2] \sqsubseteq \text{th} [S_1, T_2] \). The constraints on temporal variables ensure that the time period \( [T_1, T_2] \) is a new time period different from \( [S_1, S_2] \) and \( [S_2, T_2] \) and their subintervals.

Finally, in the meta-level representation of object clauses, clause (5.18), we have to add the constraint \( T_1 \leq T_2 \) to ensure that the head of the object clause has a well-formed, namely non-empty, annotation.

**Bottom-up semantics**

There are several ways of defining a bottom-up semantics of TACLp, related to the different possible choices of the semantic domain where the immediate consequence operator is defined. In this thesis we consider the powerset \( \mathcal{C}\text{-base}_L \times \text{Ann} \) with set-theoretic inclusion\(^3\), disregarding the partial order structure of the set of annotations \( \text{Ann} \). Other proposals are discussed in the concluding section of the chapter (see Section 5.6).

In this section we will largely use the terminology introduced in Chapter 2 for constraint logic programming.

---

\(^3\)The formal definition of \( \mathcal{C}\text{-base}_L \) is in Chapter 2. Briefly, it is the natural generalization of the notion of Herbrand Base in constraint logic programming.
To generalize the immediate consequence operator to deal with temporal annotations we consider a kind of extended interpretations, basically consisting of sets of annotated elements of $C$-base$_L$. More precisely, we define the set of (semantical) annotations
\[
Ann = \{ \text{th} [t_1, t_2] \cup \text{in} [t_1, t_2] \mid t_1, t_2 \text{ time points} \land D_C \models t_1 \leq t_2 \}
\]
where $D_C$ is the $S_C$-structure that is the intended interpretation of the constraints.

Then the lattice of interpretations is defined as $(C$-base$_L \times Ann), \sqsubseteq$ where $\sqsubseteq$ is the usual relation of set-theoretic inclusion.

**Definition 5** Let $P$ be a $TACL\bar{P}$ program, the function $T_P^C : \varphi(C$-base$_L \times Ann) \rightarrow \varphi(C$-base$_L \times Ann)$ is defined as follows.

\[
T_P^C(I) = \begin{cases}
\begin{aligned}
&\text{th}[s_1, s_2] \vee \text{in}[s_1, s_2], \\
&\text{A} \alpha \leftarrow C_1, \ldots, C_k, B_1 \alpha_1, \ldots, B_n \alpha_n \in ground_C(P), \\
&\{((B_1, \beta_1), \ldots, (B_n, \beta_n)) \subseteq I, \\
&D_C \models C_1, \ldots, C_k, \alpha_1 \sqsubseteq \beta_1, \ldots, \alpha_n \sqsubseteq \beta_n, s_1 \leq s_2
\end{aligned}
&\begin{aligned}
&\text{A} \alpha \leftarrow C_1, \ldots, C_k, B_1 \alpha_1, \ldots, B_n \alpha_n \in ground_C(P), \\
&\{((B_1, \beta_1), \ldots, (B_n, \beta_n)) \subseteq I, \\
&D_C \models C_1, \ldots, C_k, \alpha_1 \sqsubseteq \beta_1, \ldots, \alpha_n \sqsubseteq \beta_n, \\
&s_1 < r_1, r_1 \leq s_2, s_2 < r_2
\end{aligned}
\end{cases}
\]

This definition properly extends the standard definition of the immediate consequence operator. In fact, in a sense, it captures not only the Modus Ponens rule, as the standard operator does, but also rule $(\sqcup)$ (second set in the above definition). In addition, rule $(\sqsubseteq)$ is used to prove that an annotated atom holds in an interpretation: to derive the head $A \alpha$ of a clause it is not necessary to find in the interpretation exactly the atoms $B_1 \alpha_1, \ldots, B_n \alpha_n$ occurring in the body of the clause, but it suffices to find atoms $B_i \beta_i$ which imply $B_i \alpha_i$, i.e., such that each $\beta_i$ is an annotation stronger than $\alpha_i$ ($D_C \models \alpha_i \sqsubseteq \beta_i$). Finally, notice that $T_P^C(I)$ is not downward closed, namely, it is not true that if $(A, \alpha) \in T_P^C(I)$ then for all $(A, \gamma)$ such that $D_C \models \gamma \sqsubseteq \alpha$, we have $(A, \gamma) \in T_P^C(I)$. However such a closure is done at the end of the computation of the fixpoint of $T_P^C$. In this way the rule $(\sqsubseteq)$ is completely captured.

An important property of the $T_P^C$ operator, which is at the core of the definition of the bottom-up semantics, is continuity over the lattice of interpretations.

**Theorem 4 (Continuity)** For any $TACL\bar{P}$ program $P$ the function $T_P^C$ is continuous (on $(\varphi(C$-base$_L \times Ann), \sqsubseteq)$).

**Proof** The result is a direct consequence of the definition of $T_P^C$ and of the partial order $\sqsubseteq$ on the interpretations.
Let \( \{I_i\}_{i \in \mathbb{N}} \) be a chain in \( \varphi(C_{\text{base}} \times \text{Ann}) \), i.e., \( I_0 \subseteq I_1 \subseteq \ldots \subseteq I_i \). Then we have to prove

\[
(A, \alpha) \in \mathcal{T}_P^C \left( \bigcup_{i \in \mathbb{N}} I_i \right) \iff (A, \alpha) \in \bigcup_{i \in \mathbb{N}} \mathcal{T}_P^C (I_i).
\]

\[
(A, \alpha) \in \mathcal{T}_P^C (\bigcup_{i \in \mathbb{N}} I_i) \iff \text{definition of } \mathcal{T}_P^C
\]

\[
((\alpha = \text{th} [s_1, s_2] \lor \alpha = \text{in} [s_1, s_2]) \land
A \alpha \leftarrow C_1, \ldots, C_k, B_1 \alpha_1, \ldots, B_n \alpha_n \in \text{ground}_C (P) \land
\{ (B_1, \beta_1), \ldots, (B_n, \beta_n) \} \subseteq \bigcup_{i \in \mathbb{N}} I_i \land
D_C \models C_1, \ldots, C_k, \alpha_1 \subseteq \beta_1, \ldots, \alpha_n \subseteq \beta_n, s_1 \leq s_2) \lor
\]

\[
(A = \text{th} [s_1, r_2] \land A \text{th} [s_1, s_2] \leftarrow C_1, \ldots, C_k, B_1 \alpha_1, \ldots, B_n \alpha_n \in \text{ground}_C (P) \land
\{ (B_1, \beta_1), \ldots, (B_n, \beta_n) \} \subseteq \bigcup_{i \in \mathbb{N}} I_i \land
(A, \text{th} [r_1, r_2]) \in \bigcup_{i \in \mathbb{N}} I_i \land
D_C \models C_1, \ldots, C_k, \alpha_1 \subseteq \beta_1, \ldots, \alpha_n \subseteq \beta_n, s_1 < r_1, r_1 \leq s_2, s_2 < r_2)
\]

\[
\iff \text{property of set-theoretic union and } \{I_i\}_{i \in \mathbb{N}} \text{ is a chain. Notice that for }
\]

\[
(\implies) j \text{ can be any element of the set } \{ k \mid (B_i, \beta_i) \in I_k, i = 1, \ldots, n \}
\]

\[
\text{which is clearly not empty}
\]

\[
((\alpha = \text{th} [s_1, s_2] \lor \text{in} [s_1, s_2]) \land
A \alpha \leftarrow C_1, \ldots, C_k, B_1 \alpha_1, \ldots, B_n \alpha_n \in \text{ground}_C (P) \land
\{ (B_1, \beta_1), \ldots, (B_n, \beta_n) \} \subseteq I_j \land
D_C \models C_1, \ldots, C_k, \alpha_1 \subseteq \beta_1, \ldots, \alpha_n \subseteq \beta_n, s_1 \leq s_2) \lor
\]

\[
(A = \text{th} [s_1, r_2] \land A \text{th} [s_1, s_2] \leftarrow C_1, \ldots, C_k, B_1 \alpha_1, \ldots, B_n \alpha_n \in \text{ground}_C (P) \land
\{ (B_1, \beta_1), \ldots, (B_n, \beta_n) \} \subseteq I_j \land
(A, \text{th} [r_1, r_2]) \in I_j \land
D_C \models C_1, \ldots, C_k, \alpha_1 \subseteq \beta_1, \ldots, \alpha_n \subseteq \beta_n, s_1 < r_1, r_1 \leq s_2, s_2 < r_2)
\]

\[
\iff \text{definition of } \mathcal{T}_P^C
\]

\[
(A, \alpha) \in \mathcal{T}_P^C (I_j)
\]

\[
\iff \text{set-theoretic union}
\]

\[
(A, \alpha) \in \bigcup_{i \in \mathbb{N}} \mathcal{T}_P^C (I_i)
\]

We define an operator \( \downarrow \) that provides the downward closure of an interpretation.

**Definition 6 (Downward closure)** Given an interpretation \( I, I \subseteq (C_{\text{base}} \times \text{Ann}) \), its downward closure is defined as:

\[
\downarrow I = \{ (A, \alpha) \mid (A, \beta) \in I, D_C \models \alpha \subseteq \beta \}
\]

The bottom-up semantics for a program \( P \) is defined as the downward closure of the least fixpoint of \( \mathcal{T}_P^C \), which, by continuity and the Fixpoint Theorem (Theorem 1), is the least upper bound of the chain \( \{ (\mathcal{T}_P^C)^j \}_{j \in \mathbb{N}} \).

**Definition 7 (Bottom-up semantics)** Let \( P \) be a TACL program. Then the bottom-up semantics of \( P \) is defined as

\[
\mathcal{F}^C (P) = \downarrow (\mathcal{T}_P^C)^{\omega}
\]

where \( (\mathcal{T}_P^C)^{\omega} = \bigcup_{i \in \mathbb{N}} (\mathcal{T}_P^C)^i \).
Soundness and Completeness

In the spirit of [28, 87] we define the semantics of the extended vanilla meta-interpreter by relating the semantics of an object program to the semantics of the corresponding vanilla meta-program (i.e., including the meta-level representation of the object program), as we have already done in Section 5.1.1. When stating the correspondence between the object program and the meta-program we consider only formulae of interest, i.e., elements of $C \text{-} base_L$ annotated with labels from $Ann$, which are the semantic counterpart of object level annotated atoms. We show that given a TACLP program $P$ (object program) for any $A \in C \text{-} base_L$ and any $\alpha \in Ann$, $demo(A, \alpha)$ is provable at the meta-level if and only if $(A, \alpha)$ is provable in the object program. Formally,

$$demo(A, \alpha) \in (T^M_v)^\omega \iff (A, \alpha) \in \mathcal{F}^C(P) \tag{5.19}$$

where $V$ is the meta-program containing the meta-level representation of the object program $P$ according to (5.18) and the clauses (5.11)–(5.17), and $T^M_v$ is the standard immediate consequence operator of CLP. It is worth noting that $V$ is a CLP$(\mathcal{M})$ program where the constraint domain $\mathcal{M}$ is multi-sorted and composed by the constraint domain $Term$, presented in Example 2, and the constraint domain $\mathcal{C}$. Therefore if $C$ is a $\mathcal{C}$-ground instance of a constraint then $\mathcal{D}_M \models C \iff \mathcal{D}_C \models C$.

An explicit proof of Statement (5.19) can be found in [96]. However, as we will point out later, such a result is a corollary of Theorem 6, by considering as program expression $\mathcal{E}$, the TACLP program $P$ (see Corollary 1).

Since the given meta-logical definition axiomatizes a top-down operational semantics for TACLP programs, the proof of the statement (5.19) corresponds to showing the equivalence of computing top-down and bottom-up.

5.2.3 Related Work

In [70], Templog [1] and an interval based temporal logic are translated into generalized annotated logic programs. The annotations used there correspond to the $\mathfrak{th}$ annotations of TACL. To implement the annotated logic language, the paper proposed to use “reductants”, additional clauses which are derived from existing clauses to express all possible least upper bounds. The problem was that a finite program may generate infinitely many such reductants. Then, “ca-resolution” for annotated logic programs was proposed [80]. The idea is to compute dynamically and incrementally the least upper bounds by collecting partial answers. Operationally this is similar to the meta-interpreter presented in Section 5.2.2 which relies on recursion to collect the partial answers. However, in [80] the intermediate stages of the computation are not sound with respect to the standard CLP semantics.

Moreover, in [70] two fixpoint semantics, defined in terms of two different operators, are presented for generalized annotated programs (GAP). The first operator,
5.2. Temporal Annotated Constraint Logic Programming

called $T_P$, is based on interpretations which associate with each element of the Herbrand Base of the program $P$ a set of annotations which is an ideal, i.e., a set downward closed and closed with respect to finite least upper bounds. The computed ideal is the least one containing the annotations $\alpha$ of annotated atoms $A\alpha$ which are heads of (instances of) clauses whose body holds in the interpretation. The other operator, $R_P$, is based on interpretations which associate with each atom of the Herbrand Base a single annotation which is the least upper bound of the set of annotations computed as in the previous case. Our fixpoint operator for TACLP works similarly to the $T_P$ operator: at each step we close with respect to (representable) finite least upper bounds, and, although we perform the downward closure only at the end of the computation, this does not reduce the set of derivable consequences. The main difference resides in the language: TACLP is an extension of CLP, which focuses on temporal aspects, taking from GAP the basic ideas for handling annotations, whereas GAP is a general language with negation and arbitrary annotations but without constraints.

Our temporal annotations correspond to some of the predicates proposed by Galton in [55], which is a critical examination of Allen’s classical work on a theory of action and time [5]. Galton provides for both time points and time periods in dense linear time. Assuming that the intervals $I$ are not singletons, Galton’s predicate holds-in($A,I$) can be mapped into TACLP’s $A\text{in}I$, holds-on($A,I$) into $A\text{th}I$, and holds-at($A,t$) into $A\text{at}t$, where $A$ is an atomic formula. From this mapping it becomes clear that TACLP can be seen as reified FOL where annotated formulae, for example born($John$)$\text{at}t$, correspond to binary meta-relations between predicates and temporal information, for example at(born($John$),$t$). But also, TACLP can be regarded as a modal logic, where the annotations are seen as parameterized modal operators, e.g. born($John$) (at $t$).

Our temporal annotations also correspond to some temporal characteristics in the ChronoBase data model [106]. Such a model allows for the representation of a wide variety of temporal phenomena in a temporal database which cannot be expressed by using only th and in annotations. An interesting line of research is to investigate the possibility of enriching the set of annotations in order to capture some other temporal characteristics still maintaining a simple and clear semantics.

In [30], a powerful temporal logic named MTL (tense logic extended by parameterized temporal operators) is translated into first order constraint logic. The resulting language subsumes Templog, as does TACLP. The parameterized temporal operators of MTL correspond to the temporal annotations of TACLP. The constraint theory of MTL is rather complex as it involves quantified variables and implication, whose treatment goes beyond standard CLP implementations. On the other hand, TACLP inherits an efficient standard constraint-based implementation of annotations from the ACL framework.
5.3 Multi-theory Temporal Annotated Constraint Logic Programming

Multi-theory Temporal Annotated Constraint Logic Programming (MuTACLCP) is our resulting framework where temporal information can be naturally represented and handled, and, at the same time, knowledge can be separated and combined by means of meta-level composition operations.

On one side we can think of MuTACLCP as a language which enriches the formalism of Frühwirth with high-level mechanisms for structuring programs and for combining separate knowledge bases. On the other side MuTACLCP extends the language of program expressions by following the basic ideas of Frühwirth of annotating atoms with temporal information and of managing such annotations via a constraint theory.

5.3.1 MuTACLCP language

The definition of the restricted language of program expressions (5.1), consisting of union and intersection operations, changes only for the kind of clauses belonging to a plain program. Now a plain program is a TACLCP program as described in Section 5.2.1, named by a constant symbol belonging to Pname.

We assume to have the same structure of time, the same constraint theory presented in Section 5.2.1, and the further inference rules (⊔) and (⊔). However, in order to be able to compose programs we add to the constraint theory the axiomatization of the greatest lower bound \( \exists \) of two annotations:

\[
\begin{align*}
(\text{th} \cap) & \quad \text{th}[s_1, s_2] \cap \text{th}[r_1, r_2] = \text{th}[t_1, t_2] \iff \begin{array}{l}
  s_1 \leq s_2, r_1 \leq r_2, t_1 = \max\{s_1, r_1\}, \\
  t_2 = \min\{s_2, r_2\}, t_1 \leq t_2
\end{array} \\
(\text{th} \cap') & \quad \text{th}[s_1, s_2] \cap \text{th}[r_1, r_2] = \text{in}[t_2, t_1] \iff \begin{array}{l}
  s_1 \leq s_2, r_1 \leq r_2, t_1 = \max\{s_1, r_1\}, \\
  t_2 = \min\{s_2, r_2\}, t_2 < t_1
\end{array} \\
(\text{th in} \cap) & \quad \text{th}[s_1, s_2] \cap \text{in}[r_1, r_2] = \text{in}[r_1, r_2] \iff \begin{array}{l}
  s_1 \leq r_2, r_1 \leq s_2, s_1 \leq s_2, r_1 \leq r_2
\end{array} \\
(\text{th in} \cap') & \quad \text{th}[s_1, s_2] \cap \text{in}[r_1, r_2] = \text{in}[s_2, r_2] \iff s_1 \leq s_2, s_2 < r_1, r_1 \leq r_2 \\
(\text{th in} \cap") & \quad \text{th}[s_1, s_2] \cap \text{in}[r_1, r_2] = \text{in}[s_1, r_1] \iff r_1 \leq r_2, r_2 < s_1, s_1 \leq s_2 \\
(\text{in} \cap) & \quad \text{in}[s_1, s_2] \cap \text{in}[r_1, r_2] = \text{in}[t_1, t_2] \iff \begin{array}{l}
  s_1 \leq s_2, r_1 \leq r_2, t_1 = \min\{s_1, r_1\}, \\
  t_2 = \max\{s_2, r_2\}
\end{array}
\end{align*}
\]

It is straightforward to check that these axioms define the greatest lower bound with respect to the partial order relation \( \subseteq \), recalling that annotations regard time periods, i.e., convex, non-empty sets of time points.

The greatest lower bound of two \( \text{th} \) annotations, \( \text{th}[r_1, r_2] \) and \( \text{th}[s_1, s_2] \), can be:
• a th annotation if \([r_1, r_2]\) and \([s_1, s_2]\) are overlapping intervals, hence their intersection, \([t_1, t_2]\), is not empty and the greatest lower bound is just \(\text{th}\ [t_1, t_2]\) (axiom \((\text{th} \sqcap)\)).

• an in annotation, otherwise. In this case the associated interval is built as follows

\[
\begin{array}{c}
\text{th} \\
\hline
s_1 \\
\hline
s_2 \\
\hline
r_1 \\
\hline
r_2 \\
\hline
\text{in} \\
\hline
\hline
\end{array}
\]

The interval \([t_1, t_2]\) is the least convex set which intersects both \([s_1, s_2]\) and \([r_1, r_2]\). Hence \(\text{in}\ [t_1, t_2]\) is the greatest lower bound of \(\text{th}\ [s_1, s_2]\) and \(\text{th}\ [r_1, r_2]\) (axiom \((\text{th} \sqcap')\)).

In all other cases the greatest lower bound is always an in annotation whose interval contains at least the interval(s) associated with the in argument(s). For instance, the greatest lower bound of a th annotation and an in annotation with disjoint intervals is defined as follows

\[
\begin{array}{c}
\text{th} \\
\hline
s_1 \\
\hline
s_2 \\
\hline
r_1 \\
\hline
r_2 \\
\hline
\text{in} \\
\hline
\hline
\end{array}
\]

The interval \([s_2, r_2]\) is the least convex set containing \([r_1, r_2]\) and at least a point of \([s_1, s_2]\) (axiom \((\text{th in} \sqcap')\)).

The greatest lower bound will be fundamental in the definition of the intersection operation over program expressions. Notice that in TACL it is not needed since the problem of combining programs is not dealt with.

**Example 12** We can distribute information in Example 10 among different programs. In fact we can separate stable information, such as the rules establishing when a manager is busy, from what changes daily. For this reason we define two distinct programs: MANAGERS and TODAY-SCHED.

**MANAGERS:**

\[
\begin{align*}
\text{busy}(X) & \rightarrow \text{in-meeting}(X) \text{ th } [T_1, T_2] \\
\text{busy}(X) & \rightarrow \text{out-of-office}(X) \text{ th } [T_1, T_2]
\end{align*}
\]

In this program the predicates in-meeting and out-of-office are undefined since the schedule of managers varies daily. Therefore, to know whether a manager is busy or not we collect the schedules in a separate program.
TODAY-SCHED:
    in-meeting(smith) th [9am, 10am].
    in-meeting(jones) th [9.30am, 10.30am].
    out-of-office(smith) th [2pm, 3pm].

By using the union operation we can combine Managers with TODAY-SCHED and
MANAGERS ∪ TODAY-SCHED behaves exactly as the TACL program in Example 10
with the advantage that we can reuse the program Managers for other days.

Example 13 At 10pm Tom was found dead in his house. The only hint is that the
answering machine recorded some messages from 7pm up to 8pm. At a first glance,
the doctor said Tom was dead for one to two hours. The detective made a further
assumption: Tom did not answer to the telephone so he could be already dead.

We collect all these hints and assumptions into three programs, HINTS, DOCTOR
and DETECTIVE, in order not to mix facts with simple hypotheses that might change
during the investigations.

HINTS:  found at 10pm.
         ans-machine th [7pm, 8pm].

DOCTOR: dead in [T − 2:00, T − 1:00] ← found at T

DETECTIVE: dead in [T_1, T_2] ← ans-machine th [T_1, T_2]

If we combine the hypotheses of the doctor and those of the detective we can ex-
tend the period of time in which Tom possibly died. The program expression DO-
CTOR ∩ DETECTIVE behaves as

\[
    \text{dead in } [S_1, S_2] \leftarrow \text{in } [T − 2:00, T − 1:00] \cap \text{in } [T_1, T_2] = \text{in } [S_1, S_2], \\
    \text{found at } T, \\
    \text{ans-machine th } [T_1, T_2]
\]

The constraint in [T − 2:00, T − 1:00] ∩ in [T_1, T_2] = in [S_1, S_2] builds the annotation
in [S_1, S_2] in which Tom possibly died, and by using axiom (in ∩) we know that the
resulting interval is $S_1 = \min\{T − 2:00, T_1\}$ and $S_2 = \max\{T − 1:00, T_2\}$. In fact,
according to the semantics we will see in the next section, a consequence of the
program expression

HINTS ∪ (DOCTOR ∩ DETECTIVE)

is just dead in [7pm, 9pm] since the annotation in [7pm, 9pm] is the greatest lower
bound of in [8pm, 9pm] and in [7pm, 8pm].
5.3.2 Semantics of MuTACL

To give MuTACL a semantics we can go through the path followed by Brogi et al. [25] in defining the semantics for the language of program expressions (see Section 4.2). Indeed we replace logic programs with TACL programs and then we have to properly define the meaning of the composition operations over TACL programs. Thus the operational semantics is provided by adding new clauses to the meta-interpreter for TACL programs; the bottom-up semantics is an extension of the TACL immediate consequence operator, and finally, the least fixpoint semantics is the downward closure of the least fixpoint of the bottom-up semantics. To conclude this section, we prove soundness and completeness results for the meta-interpreter with respect to the least fixpoint semantics.

As in Section 5.2.2, we assume all atoms to be annotated with \texttt{th} or \texttt{in} annotations.

\textbf{Meta-interpreter}

The extended meta-interpreter is defined by the following clauses.

\begin{equation}
\text{demo} (\mathcal{E}, \text{empty}).
\end{equation}

\begin{equation}
\text{demo}(\mathcal{E}, (B_1, B_2)) \leftarrow \text{demo}(\mathcal{E}, B_1), \text{demo}(\mathcal{E}, B_2)
\end{equation}

\begin{equation}
\text{demo}(\mathcal{E}, \text{th}[T_1, T_2]) \leftarrow S_1 \leq T_1, T_1 \leq T_2, T_2 \leq S_2,
\text{clause}(\mathcal{E}, \text{th}[S_1, S_2], B), \text{demo}(\mathcal{E}, B)
\end{equation}

\begin{equation}
\text{demo}(\mathcal{E}, \text{th}[T_1, T_2]) \leftarrow S_1 \leq T_1, T_1 < S_2, S_2 < T_2,
\text{clause}(\mathcal{E}, \text{th}[S_1, S_2], B), \text{demo}(\mathcal{E}, B),
\text{demo}(\mathcal{E}, \text{th}[S_2, T_2])
\end{equation}

\begin{equation}
\text{demo}(\mathcal{E}, \text{in}[T_1, T_2]) \leftarrow T_1 \leq S_2, S_1 \leq T_2, T_1 \leq T_2,
\text{clause}(\mathcal{E}, \text{in}[S_1, S_2], B), \text{demo}(\mathcal{E}, B)
\end{equation}

\begin{equation}
\text{demo}(\mathcal{E}, \text{in}[T_1, T_2]) \leftarrow T_1 \leq S_1, S_2 \leq T_2,
\text{clause}(\mathcal{E}, \text{in}[S_1, S_2], B), \text{demo}(\mathcal{E}, B)
\end{equation}

\begin{equation}
\text{demo}(\mathcal{E}, C) \leftarrow \text{constraint}(C), C
\end{equation}

\begin{equation}
\text{clause}(\mathcal{E}_1 \cup \mathcal{E}_2, A \alpha, B) \leftarrow \text{clause}(\mathcal{E}_1, A \alpha, B)
\end{equation}
\[ \text{clause}(E_1 \cup E_2, A \alpha, B) \leftarrow \text{clause}(E_2, A \alpha, B) \] (5.28)

\[ \text{clause}(E_1 \cap E_2, A \gamma, (B_1, B_2)) \leftarrow \text{clause}(E_1, A \alpha, B_1), \text{clause}(E_2, A \beta, B_2), \alpha \cap \beta = \gamma \] (5.29)

A clause \( A \alpha \leftarrow B \) of a plain program \( P \) is represented at the meta-level by

\[ \text{clause}(P, A \alpha, B) \leftarrow S_1 \leq S_2 \] (5.30)

where \( \alpha = \text{th}[S_1, S_2] \) or \( \alpha = \text{in}[S_1, S_2] \).

The only difference between clauses (5.20)–(5.26), (5.30) and the meta-interpreter for TACL P programs consists of the use of a two-argument predicate \text{demo} and a three-argument predicate \text{clause}. Now we have to specify where to solve a goal because there is no longer one underlying program but a collection of programs that can be combined together. Hence the first argument of both predicates is a program expression. A consequence of this remark is that if \( P \) is a TACL P program then

\[ \text{demo}(A \alpha) \text{ is provable } \iff \text{demo}(P, A \alpha) \text{ is provable} \] (5.31) through TACL P meta-interpreter through MuTACL P meta-interpreter

As far as the meta-level definition of the union and intersection operations is concerned, clauses implementing the union operation are not changed with respect to the original definition given in Section 4.2, whereas in the clause implementing the intersection operation a constraint which expresses the kind of annotation for the derived atom is added. Informally, a clause \( A \alpha \leftarrow B \), belonging to the intersection of two program expressions \( E_1 \) and \( E_2 \), is built by taking one instance of clause in each program expression \( E_1 \) and \( E_2 \), such that the head atoms of the two clauses are unifiable. Let such instances of clauses be \( cl_1 \) and \( cl_2 \). Then \( B \) is the conjunction of the bodies of \( cl_1 \) and \( cl_2 \) and \( A \) is the unified atom labelled with the greatest lower bound of the annotations of the heads of \( cl_1 \) and \( cl_2 \).

The following example shows the usefulness of clause (5.23) to derive new temporal information according to the inference rule (\( \cup \)).

**Example 14** Consider two distinct library databases DB1 and DB2 containing information about loans. Mary first borrowed the book *Hamlet* from May 12, 1995 to June 12, 1995 from DB1 and then on June 12, 1995 she took the book from the other library up to August 1, 1995.

**DB1:**

\[ \text{borrow(mary, hamlet)} \text{th}[\text{may 12 1995, jun 12 1995}]. \]

**DB2:**

\[ \text{borrow(mary, hamlet)} \text{th}[\text{jun 12 1995, aug 1 1995}]. \]
The period of time in which Mary borrowed Hamlet can be obtained by querying the union of the above theories as follows:

\[
demo(DB1 \cup DB2, \text{borrow}(\text{mary, hamlet}) \text{th} [T_1, T_2]).
\]

By using clause (5.23), we can derive the interval \([\text{may 12 1995, aug 1 1995}] \) (more precisely, the constraints \(\text{may 12 1995} \leq T_1, T_1 < \text{jun 12 1995}, \text{jun 12 1995} < T_2, T_2 \leq \text{aug 1 1995} \) are derived) that otherwise would be never generated. In fact, by applying clause (5.22) alone, we can prove only that Mary borrowed Hamlet in the intervals \([\text{may 12 1995, jun 12 1995}] \) and \([\text{jun 12 1995, aug 1 1995}] \) (or in subintervals of them) separately.

**Bottom-up semantics**

The bottom-up semantics of a program expression \(\mathcal{E} \) is the immediate consequence operator \(T^\mathcal{E}_\mathcal{P} \) which extends the one for TACL programs (Definition 5) to deal with program expressions.

The lattice of interpretations is unchanged, i.e., it is \((\wp(\text{C-base}_L \times \text{Ann}), \subseteq)\). The immediate consequence operator of a program expression is compositionally defined in terms of the immediate consequence operator of the sub-expressions.

**Definition 8 (Bottom-up semantics)** Let \(\mathcal{E} \) be a program expression, the function \(T^\mathcal{E}_\mathcal{P} : \wp(\text{C-base}_L \times \text{Ann}) \rightarrow \wp(\text{C-base}_L \times \text{Ann}) \) is defined as follows.

- (\(\mathcal{E} \) is a plain program \(P\))
  \[
  T^\mathcal{E}_P(I) = T^P_I(I)
  \]

- (\(\mathcal{E} = \mathcal{E}_1 \cup \mathcal{E}_2\))
  \[
  T^\mathcal{E}_{\mathcal{E}_1 \cup \mathcal{E}_2}(I) = T^\mathcal{E}_{\mathcal{E}_1}(I) \cup T^\mathcal{E}_{\mathcal{E}_2}(I)
  \]

- (\(\mathcal{E} = \mathcal{E}_1 \cap \mathcal{E}_2\))
  \[
  T^\mathcal{E}_{\mathcal{E}_1 \cap \mathcal{E}_2}(I) = T^\mathcal{E}_{\mathcal{E}_1}(I) \cap T^\mathcal{E}_{\mathcal{E}_2}(I)
  \]

where \(I_1 \cap I_2 = \{(A, \gamma) \mid (A, \alpha) \in I_1, (A, \beta) \in I_2, \mathcal{D}_\mathcal{C} \models \alpha \cap \beta = \gamma\}\).

The set of immediate consequences of a plain program is the set computed by \(T^P_P\), since a plain program is a TACL program. The set of immediate consequences of a union of program expressions is the set-theoretic union of the immediate consequences of each program expression, whereas such a set for the intersection of program expressions consists of atoms, which are consequences of both program expressions, and the associated label is the greatest lower bound of the annotations associated to such consequences. It is worth pointing out a property of the definition of the intersection operation.
Proposition 1  Let $I_1$ and $I_2$ be two interpretations. Then
\[ \downarrow (I_1 \cap I_2) = \downarrow I_1 \cap \downarrow I_2 \]

**Proof** Assume $(A, \alpha) \in \downarrow (I_1 \cap I_2)$. By definition of downward closure there exists $\gamma$ such that $(A, \gamma) \in I_1 \cap I_2$ and $\mathcal{D}_C \models \alpha \subseteq \gamma$. By definition of $\cap$ there exists $\beta$ and $\beta'$ such that $(A, \beta) \in I_1$ and $(A, \beta') \in I_2$ and $\mathcal{D}_C \models \beta \cap \beta' = \gamma$. Therefore $\mathcal{D}_C \models \alpha \subseteq \beta, \alpha \subseteq \beta'$, by definition of downward closure we conclude $(A, \alpha) \in \downarrow I_1$ and $(A, \alpha) \in \downarrow I_2$, i.e., $(A, \alpha) \in \downarrow (I_1 \cap I_2)$.

Vice versa assume $(A, \alpha) \in \downarrow (I_1 \cap I_2)$. By definition of set-theoretic intersection and downward closure there exists $\beta$ and $\beta'$ such that $\mathcal{D}_C \models \alpha \subseteq \beta, \alpha \subseteq \beta'$ and $(A, \beta) \in I_1$ and $(A, \beta') \in I_2$. By definition of $\cap$, $(A, \gamma) \in I_1 \cap I_2$ and $\mathcal{D}_C \models \beta \cap \beta' = \gamma$. By property of the greatest lower bound $\mathcal{D}_C \models \alpha \subseteq \beta \cap \beta'$, hence $(A, \alpha) \in \downarrow (I_1 \cap I_2)$.

This property ensures that if we consider the downward closure, the meaning of the intersection of two program expressions is the set-theoretic intersection of their immediate consequences recovering the intuition that the program expressions have to agree at each computation step (see Section 4.2).

The $\mathcal{T}_E$ operator is continuous over the lattice of interpretations.

**Theorem 5 (Continuity)** Let $E$ be a program expression. The function $\mathcal{T}_E$ is continuous (on $(\wp(\text{Base}_L \times \text{Ann}), \subseteq)$).

**Proof** Let $\{I_i\}_{i \in \mathbb{N}}$ be a chain in $(\wp(\text{Base}_L \times \text{Ann}), \subseteq)$, i.e., $I_0 \subseteq I_1 \subseteq \ldots \subseteq I_i, \ldots$. Then we have to prove

\[ (A, \alpha) \in \mathcal{T}_E \left( \bigcup_{i \in \mathbb{N}} I_i \right) \iff (A, \alpha) \in \bigcup_{i \in \mathbb{N}} \mathcal{T}_E (I_i). \]

The proof is by structural induction of $E$.

($E$ is a plain program $P$). Straightforward, since $\mathcal{T}_P(I) = \mathcal{T}_P(I)$ and $\mathcal{T}_E$ is continuous on $(\wp(\text{Base}_L \times \text{Ann}), \subseteq)$.

($E = Q \cup R$).

\[ (A, \alpha) \in \mathcal{T}_{\text{QUR}} \left( \bigcup_{i \in \mathbb{N}} I_i \right) \iff \{\text{definition of } \mathcal{T}_{\text{QUR}}\} \]

\[ (A, \alpha) \in \mathcal{T}_Q \left( \bigcup_{i \in \mathbb{N}} I_i \right) \cup \mathcal{T}_R \left( \bigcup_{i \in \mathbb{N}} I_i \right) \iff \{\text{inductive hypothesis}\} \]

\[ (A, \alpha) \in \left( \bigcup_{i \in \mathbb{N}} \mathcal{T}_Q (I_i) \right) \cup \left( \bigcup_{i \in \mathbb{N}} \mathcal{T}_R (I_i) \right) \iff \{\text{properties of union}\} \]

\[ (A, \alpha) \in \bigcup_{i \in \mathbb{N}} \left( \mathcal{T}_Q (I_i) \cup \mathcal{T}_R (I_i) \right) \iff \{\text{definition of } \mathcal{T}_{\text{QUR}}\} \]

\[ (A, \alpha) \in \bigcup_{i \in \mathbb{N}} \mathcal{T}_{\text{QUR}} (I_i) \]
(\mathcal{E} = \mathcal{Q} \cap \mathcal{R}).

\( (A, \alpha) \in T^c_{\mathcal{Q} \cap \mathcal{R}} (\bigcup_{i \in \mathbb{N}} I_i) \)

\[ \iff \quad \text{definition of } T^c_{\mathcal{Q} \cap \mathcal{R}} \}

\( (A, \alpha) \in T^c_{\mathcal{Q}} (\bigcup_{i \in \mathbb{N}} I_i) \cap T^c_{\mathcal{R}} (\bigcup_{i \in \mathbb{N}} I_i) \)

\[ \iff \quad \text{inductive hypothesis} \}

\( (A, \alpha) \in (\bigcup_{i \in \mathbb{N}} T^c_{\mathcal{Q}} (I_i)) \cap (\bigcup_{i \in \mathbb{N}} T^c_{\mathcal{R}} (I_i)) \)

\[ \iff \quad \text{definition of } \cap \text{ and monotonicity of } T^c \}

\( (A, \alpha) \in \bigcup_{i \in \mathbb{N}} T^c_{\mathcal{Q} \cap \mathcal{R}} (I_i) \)

\( (A, \alpha) \in \bigcup_{i \in \mathbb{N}} T^c_{\mathcal{Q} \cap \mathcal{R}} (I_i) \)

\[ \blacksquare \]

Now we define the least fixpoint semantics for a program expression as the downward closure of the least fixpoint of \( T^c_\mathcal{E} \) which is \( \bigcup_{i \in \mathbb{N}} (T^c_\mathcal{E})^i \) because of the continuity of the function \( T^c_\mathcal{E} \).

**Definition 9 (Least fixpoint semantics)** Let \( \mathcal{E} \) be a program expression. Then the least fixpoint semantics of \( \mathcal{E} \) is defined as

\[ \mathbb{F}^c(\mathcal{E}) = \downarrow (T^c_\mathcal{E})^\omega \]

We chose to do the closure once at the end of the computation of the fixpoint of \( T^c_\mathcal{E} \). However, since the downward closure is a continuous function on the lattice of interpretations, we have

\[ \downarrow \left( \bigcup_{i \in \mathbb{N}} (T^c_\mathcal{E})^i \right) = \bigcup_{i \in \mathbb{N}} \downarrow (T^c_\mathcal{E})^i \]

showing that the set of consequences we obtain is equal to the one computed by doing the closure at each step. Hence we completely capture rule (\( \sqsubseteq \)).

Finally, assuming that the formulae of interest are elements of \( \mathcal{C}\text{-base}_L \) annotated with labels from \( \text{Ann} \), we prove the soundness and completeness of the meta-interpreter with respect to the least fixpoint semantics.

**Theorem 6 (Soundness and completeness)** Let \( \mathcal{E} \) be a program expression and let \( V \) be the meta-program containing the meta-level representation of the object level programs occurring in \( \mathcal{E} \) and the clauses (5.20)-(5.29). For any \( A \in \mathcal{C}\text{-base}_L \) and \( \alpha \in \text{Ann} \), the following statement holds:

\[ \text{demo}(\mathcal{E}, A \alpha) \in (T^v_\mathcal{M})^\omega \iff (A, \alpha) \in \mathbb{F}^c(\mathcal{E}) \]

where \( T^v_\mathcal{M} \) is the standard immediate consequence operator for CLP programs and \( \mathcal{M} \) is the constraint domain defined in Section 5.2.2.
The proof of this theorem is presented in Appendix A.2. An immediate corollary of this theorem states the soundness and the completeness of TACL meta-interpreter with respect to the bottom-up semantics for TACL programs.

**Corollary 1** Let \( P \) be a TACL program and let \( V \) be the meta-program containing the meta-level representation of \( P \) and the clauses (5.11)-(5.17). For any \( A \in \mathcal{C}_{base_L} \) and \( \alpha \in Ann \), the following statement holds:

\[
demo(A \alpha) \in (T^M_V)^\omega \iff (A, \alpha) \in \mathcal{F}^C(P)
\]

**Proof** By statement (5.31) we have

\[
demo(A \alpha) \in (T^M_V)^\omega \iff demo(P, A \alpha) \in (T^M_m)^\omega
\]

where \( mV \) is the meta-program containing the meta-level representation of \( P \) and the clauses (5.20)-(5.29). Now by using Theorem 6 we obtain \((A, \alpha) \in \mathcal{F}^C(P)\). But \( \mathcal{F}^C(P) \) is the downward closure of \((T^C_P)^\omega \) which is, by definition of \( T^C_P \), just \((T^C_P)^\omega \). Since by definition \( \mathcal{F}^C(P) = \downarrow (T^C_P)^\omega \) hence we can conclude that \((A, \alpha) \in \mathcal{F}^C(P)\).

**5.4 Application to legal reasoning**

In this section we show the usefulness of our framework by showing its application to legal reasoning.

The basic idea is that laws and rules are naturally represented in separate theories and that they can be combined in ways that are necessarily more complex than plain merging. Time is another crucial ingredient in the definition of laws and rules. Quite often, rules have to refer to instants of time and, furthermore, they have a validity for a fixed period of time. This is especially true for laws and rules which concern taxation and government budget related regulations in general.

**British Nationality Act**

We start with a classical example in the field of legal reasoning [100], a small part of the British Nationality Act, where the simple partitioning of the knowledge in separate programs, and the use of the basic union operation allow one to use the temporal information in an orderly way.

The statement

\[x \text{ obtains the British Nationality at time } t\]

if \( x \) is born in U.K. at time \( t \) and

\( t \) is after commencement and

\( y \) is parent of \( x \) and

\( y \) is a British citizen at time \( t \)

or \( y \) is a British resident at time \( t \)
is modeled by the following program. Assume that Jan 1 1955 is the commencement date of the law.

BNA:
\[
get\text{-citizenship}(X) \text{at } T \leftarrow T \geq \text{Jan 1 1955}, \text{born}(X,uk) \text{at } T,
\]
\[
parent(Y,X) \text{at } T, \text{british\text{-citizen}}(Y) \text{at } T
\]
\[
get\text{-citizenship}(X) \text{at } T \leftarrow T \geq \text{Jan 1 1955}, \text{born}(X,uk) \text{at } T,
\]
\[
parent(Y,X) \text{at } T, \text{british\text{-resident}}(Y) \text{at } T
\]

Now, we can encode in a separate program the data of a person, say John.

JOHN:
\[
\text{born}(john,uk) \text{at aug 10 1969}.
\]
\[
\text{parent}(bob,john) \text{th}[T,\infty] \leftarrow \text{born}(john,\_\_\_\_) \text{at } T
\]
\[
\text{british\text{-citizen}}(bob) \text{th}[\text{sept 6 1940},\infty].
\]

We can now use the union operation to inquire about the citizenship of John,

\[
\text{demo}(\text{BNA} \cup \text{JOHN}, \text{get\text{-citizenship}}(john) \text{at } T)
\]

obtaining as result \( T = \text{aug 10 1969} \).

Cinema tickets

Since 1997, an Italian regulation for encouraging people to go to the cinema, states that on Wednesdays the ticket price is 8000 liras, whereas in the rest of the week is 12000 liras. The situation can be modeled by the following theory BOXOFF.

BOXOFF:
\[
\text{ticket}(8000, X) \text{at } T \leftarrow T \geq \text{Jan 1 1997}, \text{wed} \text{at } T
\]
\[
\text{ticket}(12000, X) \text{at } T \leftarrow T \geq \text{Jan 1 1997}, \text{non\_wed} \text{at } T
\]

The constraint \( T \geq \text{Jan 1 1997} \) represents the validity of the clause, i.e, from January 1, 1997 onwards.

The predicates \text{wed} and \text{non\_wed} are defined in a separate theory DAYS, where \( w \) is assumed to be the last Wednesday of 1996.

DAYS:
\[
\text{wed} \text{at } w.
\]
\[
\text{wed} \text{at } T + 7 \leftarrow \text{wed} \text{at } T
\]
\[
\text{non\_wed} \text{th}[w + 1, w + 6],
\]
\[
\text{non\_wed} \text{at } T + 7 \leftarrow \text{non\_wed} \text{at } T
\]
It is worth noting that we can easily express periodic temporal information by a recursive predicate: a day is Wednesday if it is a date which is known to be Wednesday or it is a day coming seven days after a day proved to be Wednesday. The predicate non-wed is defined in an analogous way, in this case the unit clause states that all six consecutive days following a Wednesday are not Wednesdays.

Now, let us suppose that the owner of a cinema wants to increase the discount for young people on Wednesdays establishing that the ticket price for people who are eighteen years old or younger is 6000 liras. By using the intersection operation we can build a program expression that represents exactly the desired policy. We define three new programs: \text{Cons}, \text{Disc} and \text{Age}.

\text{Cons: } \quad \text{ticket}(8000, X) \at T \leftarrow \text{age}(X, Y) \at T, Y > 18 \\
\quad \text{ticket}(12000, X) \at T.

In the above theory we specify how the predicate definitions in BoxOff have to change with respect to the new policy. In fact to get a 8000 liras ticket now there is the further constraint that the customer has to be older than eighteen years old. On the other hand, no further requirement is asked to buy a 12000 liras ticket.

\text{Disc: } \quad \text{ticket}(6000, X) \at T \leftarrow \text{wed} \at T, \text{age}(X, Y) \at T, Y \leq 18

The clause states that to buy a 6000 liras ticket a person has to go to the cinema on Wednesdays and she/he has to be eighteen years old or younger.

The programs \text{Cons} and \text{Disc} are parametric with respect to the predicate \text{age} whose definition can be given in the theory \text{Age}.

\text{Age: } \quad \text{age}(X, Y) \at T \leftarrow \text{born}(X) \at T_1, \text{year-diff}(T_1, T, Y)

At this point we can compose the above programs to obtain the desired knowledge representing the new policy.

\((\text{BoxOff} \cap \text{Cons}) \cup \text{Disc} \cup \text{Days} \cup \text{Age})

In order to know how much is a ticket for a given person, we have to join the above program expression with a separate program containing the date of the birth of this person. For instance,

\text{TOM: } \quad \text{born(tom)} \at may \ 7 \ 1982.

Then the answer to the query

\[ \text{demo}(((\text{BoxOff} \cap \text{Cons}) \cup \text{Disc} \cup \text{Days} \cup \text{TOM}), \text{ticket}(X, \text{tom}) \at may \ 20 \ 1998) \]

is \( X = 6000 \) since \textit{May 20 1998} is Wednesday and Tom is sixteen years old.
Invim

Invim was an Italian law dealing with paying taxes on real estate transactions. The original regulation, in force since January 1, 1950, depends on time calculations, since the amount of taxes depends on the period of ownership of the real estate property. Furthermore, the law was abolished in 1992, which means that the rules still apply but only for the period antecedent to 1992.

Our approach allows us to have a program containing the original regulation and to have two other programs, one containing the constraints due to the decisions taken in 1992, and the other containing the new policy. It is important to notice that the design of the constraining theory can be done without taking care of the details (which may be quite complicated) embodied in the original law.

The following program - INVIM - contains a sketch of the original body of regulations.

INVIM:
\[ \text{due}(\text{Amount}, X, \text{Prop}) \text{ th } [T_2, \infty] \leftarrow T_2 \geq \text{jan 1 1950} , \text{ buys}(X, \text{Prop}) \text{ at } T_1, \]
\[ \text{sells}(X, \text{Prop}) \text{ at } T_2, \]
\[ \text{compute}(\text{Amount}, X, \text{Prop}, T_1, T_2) \]

\[ \text{compute}(\text{Amount}, X, \text{Prop}, T_1, T_2) \leftarrow \ldots \]

In order to adapt the above body of regulations to the new situation imposed by the 1992 decisions, we construct two new theories as in the previous example. The first one is designed as a set of constraints on the applicability of the original rules, while the second one is designed to embody new rules capable of handling the new situation.

CONSTRAINTS:
\[ \text{due}(\text{Amount}, X, \text{Prop}) \text{ th } [\text{jan 1 1993}, \infty] \leftarrow \]
\[ \text{sells}(X, \text{Prop}) \text{ in } [\text{jan 1 1950, dec 31 1992}] \]

\[ \text{compute}(\text{Amount}, X, \text{Prop}, T_1, T_2). \]

The first rule specifies that the relation \text{due} is computed, i.e., its original body is computed, provided that the selling date is antecedent to December 31, 1992. The second rule specifies that the rules for \text{compute}, whatever number they are, and whatever complexity they have, carry on unconstrained to the new version of the regulation.

ADDITIONS:
\[ \text{due}(\text{Amount}, X, \text{Prop}) \text{ th } [T_2, \infty] \leftarrow T_2 \geq \text{jan 1 1993} , \text{ buys}(X, \text{Prop}) \text{ at } T_1, \]
\[ \text{sells}(X, \text{Prop}) \text{ at } T_2, \]
\[ \text{compute}(\text{Amount}, X, \text{Prop}, T_1, \text{dec 31 1992}) \]
This rule handles the case of selling a property, bought before December 31 1992, after the first of January, 1993.

Now, we consider a separate theory representing the transactions regarding Mary who bought an apartment on March 8, 1965 and sold it on July 2, 1997.

**Trans1:**

\[
\text{buys}(\text{mary,apt8}) \text{ at mar 8 1965.}\]
\[
\text{sells}(\text{mary,apt8}) \text{ at jul 2 1997.}\]

The query

\[
\text{demo}(\text{INVIM } \cup \text{ Trans1}, \text{ due(Amount,mary,apt8) th } [\ldots])
\]

yields the amount, say 32.1, that Mary has to pay when selling the apartment according to the old regulations. On the other hand, the query

\[
\text{demo}(((\text{INVIM } \cap \text{ CONSTRAINTS}) \cup \text{ ADDITIONS}) \cup \text{ Trans1},
\]
\[
\text{due(Amount,mary,apt8) th } [\ldots])
\]

yields the amount, say 27.8, according to the new regulations. It is a smaller amount of taxes because taxes are computed only for the period from March 8 1965 to December 31 1992, by using the clause in the program ADDITIONS. We cannot apply the clause in \text{INVIM } \cap \text{ CONSTRAINTS} because the condition on the selling date (\text{sells(X,Prop) in [jan 1 1950, dec 31 1992]}) does not hold.

In the following transaction Paul buys the flat on January 1, 1995.

**Trans2:**

\[
\text{buys}(\text{paul,apt9}) \text{ at jan 1 1995.}\]
\[
\text{sells}(\text{paul,apt9}) \text{ at sep 12 1998.}\]

\[
\text{demo}(\text{INVIM } \cup \text{ Trans2}, \text{ due(Amount,paul,apt9) th } [\ldots])
\]

\[
\text{Amount} = 1.7
\]

\[
\text{demo}(((\text{INVIM } \cap \text{ CONSTRAINTS}) \cup \text{ ADDITIONS}) \cup \text{ Trans2},
\]
\[
\text{due(Amount,paul,apt9) th } [\ldots])
\]

\[
\text{no}
\]

If we query the theory \text{INVIM } \cup \text{ Trans2} Paul must pay a certain amount of tax, say 1.7, but if we consider the updated regulation he must not pay the Invim tax because he bought and sold the flat after December 31, 1992. This is why the answer to the query with respect to the theory \text{INVIM } \cap \text{ CONSTRAINTS} \cup \text{ ADDITIONS} \cup \text{ Trans2} is \text{no}, i.e., no taxe is due.

Summing up, the union operation is used to obtain a larger set of clauses. We can join a program with another to provide it with definitions of its undefined
predicates (e.g., AGE provides a definition for the predicate age not defined in DISC and CONS) or simply to add new definitions for a predicate (e.g., in DISC one can find a new definition for the predicate ticket). On the other hand, the use of the intersection operator provides a natural way of imposing constraints on existing programs (e.g., the program CONS constrains the BoxOFF definition of ticket). Now such constraints have impact not only on how to compute a particular property, as the intersection operation defined by Brogi et al. [25] does, but also on the temporal information in which the property holds. Moreover, the use of TACLP programs allows us to represent and reason on temporal information in a natural way. We avoid the problems of our first approach to dealing with time. Since time is explicit, at the object level we can directly access to the temporal information associated with atoms. For instance, now the date of birth of John is an annotation, \textit{born(john,uk) at aug 10 1969}, and it can be used to build new temporal information as in clause \textit{parent(bob,john) th [T, ∞] ← born(john, -) at T}, or to establish temporal relations with other predicates, e.g., the conjunction \textit{born(john,uk) at T, british-citizen(bob) at T} establishes that both atoms have to hold in the same time point T.

Periodic information can be easily expressed by recursive predicates (see the predicates \textit{wed and non-wed}) and indefinite temporal information is represented by \textit{in} annotation. In the program ADDITIONS the \textit{in} annotation is used to specify that a certain date is within a time period \textit{(sell(X,Prop) in [jan 1 1950, dec 31 1992])}. This is a case in which it is not important to know the precise date but it is sufficient to have an information which delimits the time period in which it can occur.

5.5 Translation of programs with annotated clauses into MuTACLP programs

In this section we show that the language presented in Section 5.1 is a proper subset of MuTACLP. First of all we consider the structure of time. In our first approach time can be only discrete and this is the only difference with MuTACLP since in both cases the time line is left-bounded by 0 and open to the future with the symbol ∞ denoting a time point later than any other. Both approaches use time periods which are non-empty, convex set of time points.

The kind of temporal information we represent with annotated clauses can be modeled by \textit{th} annotations: a property holds in each time point of the interval, i.e., \textit{throughout} the time period. Hence an annotated clause \textit{A ⊆ B_1,...,B_n \sqcap [s_0 ^ 1, s_0 ^ 2]} is translated into the following MuTACLP clause.

\[ A \text{ th } [t_1, t_2] ← \left( \bigcap_{i=0,n} \text{ th } [s_i ^ 1, s_i ^ 2] = \text{ th } [t_1, t_2] \right), B_1 \text{ th } [s_1 ^ 1, s_1 ^ 2],..., B_n \text{ th } [s_n ^ 1, s_n ^ 2] \] (5.32)

The translation of an annotated clause reflects the way the immediate consequence
operator $T^{int}$ (Definition 2) computes on plain programs: an atom $A$ holds in an interval $J$ if there exists a clause in the plain program such that the atoms occurring in the body hold in intervals $I_i$ and the intersection of such intervals with the interval of validity of the clause is just $J$. Indeed the constraint $\text{th} [t_1, t_2] = \left( \prod_{i=0}^{n} \text{th} [s^1_i, s^2_i] \right)$ ensures that $[t_1, t_2]$ is the non-empty intersection of the time periods $[s^1_i, s^2_i]$ with $i = 0, n$.

We identify with the name $\mu(P)$ the MuTACLP program obtained by translating each annotated clause from a plain program $P$. Then, we denote with $\mu(\mathcal{E})$ the translation of a program expression $\mathcal{E}$ defined as follows:

* $\mu(\mathcal{Q} \cup \mathcal{R}) = \mu(\mathcal{Q}) \cup \mu(\mathcal{R})$
* $\mu(\mathcal{Q} \cap \mathcal{R}) = \mu(\mathcal{Q}) \cap \mu(\mathcal{R})$

Since a program $P$ with annotated clauses is basically a logic program, the only constraints present in the MuTACLP program, $\mu(P)$, are those introduced during the translation and are equalities over terms from $P$, used to replace the unification mechanism. Therefore we can fix as constraint domain $\mathcal{C}$, the multi-sorted domain composed by the constraint domain $\text{Term}$, presented in Example 2 (Chapter 2), and the constraint domain for handling annotations $\mathcal{A}$, where time points are interpreted into natural numbers and $\infty$ into itself. We want that a ground atom $A \in B_L$ is interpreted into itself in the $\mathcal{C}$-base $L$. As a consequence an element $(A, J) \in B_L \times \text{Int}$ is translated into a pair denoted by $\mu((A, J))$ and defined in this way

* $(A, \text{th} [r_1, r_2])$ if $r_1, r_2$ are natural numbers and $x \in J \iff r_1 \leq x \leq r_2$,
* $(A, \text{th} [r_1, \infty])$ if $r_1$ is a natural number and $x \in J \iff x \geq r_1$.

Given an interpretation $\mathcal{I} \subseteq B_L \times \text{Int}$, we denote with $\mu(\mathcal{I})$ the set

$$\{ \mu((A, J)) \mid (A, J) \in \mathcal{I} \}$$

Finally, we observe that the axiomatization of the operation $\overline{\cap}$ between time periods coincides with the greatest lower bound between two $\text{th}$ annotations with overlapping time periods and the subinterval relation $\subseteq$ between time periods is exactly axiom (th $\subseteq$). It is immediate to prove that given three ground time periods $[s_1, s_2], [r_1, r_2], [t_1, t_2]$

$$[s_1, s_2] \overline{\cap} [r_1, r_2] = [t_1, t_2] \text{ is provable iff } D_{\mathcal{C}} \models \text{th} [s_1, s_2] \cap \text{th} [r_1, r_2] = \text{th} [t_1, t_2] \quad (5.33)$$

$$[s_1, s_2] \subseteq [r_1, r_2] \text{ is provable iff } D_{\mathcal{C}} \models \text{th} [s_1, s_2] \subseteq \text{th} [r_1, r_2] \quad (5.34)$$

where in the right hand side, $s_1, s_2, r_1, r_2, t_1, t_2$ are the interpreted time points in $D_{\mathcal{C}}$, denoted, with a little abuse of notation, by the same symbols of the syntactical time points in the left hand side.
We are now ready to prove that given a program expression $\mathcal{E}$, if an atom $A$ which holds in $J$ is a consequence of $\mathcal{E}$ by using $T^{{\text{int}}}$, then $\mu((A, J))$ is a consequence of $\mu(\mathcal{E})$ by using $T^K_{\mu(\mathcal{E})}$.

**Lemma 1** Let $\mathcal{E}$ be a program expression and $\mathcal{I} \subseteq B_L \times \text{Int}$. For any object level atomic formula $A$ and any ground interval $J$, we have

$$(A, J) \in T^{{\text{int}}}_P(\mathcal{I}) \iff \mu((A, J)) \in T^K_{\mu(P)}(\mu(\mathcal{I}))$$

**Proof** The proof is by structural induction on $\mathcal{E}$.

$$(\mathcal{E} \text{ is a plain program } P).$$

$$(A, J) \in T^{{\text{int}}}_P(\mathcal{I})$$

$\iff \{\text{definition of } T^{{\text{int}}} \text{ for a plain program}\}$$

$A \leftarrow B_1, \ldots, B_n \sqsubseteq I_0 \in \text{ground}_P(\mathcal{I}) \land$

$\{(B_1, I_1), \ldots, (B_n, I_n)\} \subseteq \mathcal{I} \land J = \bigcap_{i=1}^n I_i \land J \neq \emptyset$

$\iff \{\text{translation rules and remarks (5.10) and (5.33) and}\}$

$\mu((A, J)) = (A, \text{th}[t_1, t_2])$

$A \text{ th } [t_1, t_2] \leftarrow \left(\prod_{i=1}^n \text{th } I_i = \text{th } [t_1, t_2]\right), B_1 \text{ th } I_1, \ldots, B_n \text{ th } I_n \in \text{ground}_K(\mu(P))$

$\land \{(B_1, \text{th } I_1), \ldots, (B_n, \text{th } I_n)\} \subseteq \mu(\mathcal{I}) \land \mu((A, J)) = (A, \text{th}[t_1, t_2])$\land

$D \models \left(\prod_{i=1}^n \text{th } I_i = \text{th } [t_1, t_2]\right), t_1 \leq t_2$

$\iff \{\text{definition of } T^K_{\mu(P)} \text{ and reflexivity of } \models\}$

$\mu((A, J)) \in T^K_{\mu(P)}(\mu(\mathcal{I}))$

$$(\mathcal{E} = \mathcal{Q} \cup \mathcal{R}).$$

$$(A, J) \in T^{{\text{int}}}_{\mathcal{Q} \cup \mathcal{R}}(\mathcal{I})$$

$\iff \{\text{definition of } T^{{\text{int}}} \text{ for union of program expressions}\}$

$$(A, J) \in T^{{\text{int}}}_\mathcal{Q}(\mathcal{I}) \cup T^{{\text{int}}}_\mathcal{R}(\mathcal{I})$$

$\iff \{\text{inductive hypothesis}\}$

$\mu((A, J)) \in T^K_{\mu(\mathcal{Q})}(\mu(\mathcal{I})) \lor \mu((A, J)) \in T^K_{\mu(\mathcal{R})}(\mu(\mathcal{I}))$

$\iff \{\text{definition of } T^K \text{ for union of program expressions}\}$

$\mu((A, J)) \in T^K_{\mu(\mathcal{Q}) \cup \mu(\mathcal{R})}(\mu(\mathcal{I})) = T^K_{\mu(\mathcal{Q} \cup \mathcal{R})}(\mu(\mathcal{I}))$

$$(\mathcal{E} = \mathcal{Q} \cap \mathcal{R}).$$

$$(A, J) \in T^{{\text{int}}}_{\mathcal{Q} \cap \mathcal{R}}(\mathcal{I})$$

$\iff \{\text{definition of } T^{{\text{int}}} \text{ for intersection of program expressions}\}$

$$(A, J) \in T^{{\text{int}}}_\mathcal{Q}(\mathcal{I}) \cap T^{{\text{int}}}_\mathcal{R}(\mathcal{I})$$

$\iff \{\text{definition of } \cap\}$

$$(A, K) \in T^{{\text{int}}}_\mathcal{Q}(\mathcal{I}) \land (A, K') \in T^{{\text{int}}}_\mathcal{R}(\mathcal{I}) \land J = K \cap K' \land J \neq \emptyset$

$\iff \{\text{inductive hypothesis, translation rules, and remarks (5.10) and (5.33)}\}$

$\mu((A, K)) \in T^K_{\mu(\mathcal{Q})}(\mu(\mathcal{I})) \land \mu((A, K')) \in T^K_{\mu(\mathcal{R})}(\mu(\mathcal{I}))$
\[
\mu((A, K)) = (A, \text{th}[r_1, r_2]) \land \mu((A, K')) = (A, \text{th}[s_1, s_2]) \land \\
\mu((A, J)) = (A, \text{th}[t_1, t_2]) \land D_c \models \text{th}[r_1, r_2] \land \text{th}[s_1, s_2] = \text{th}[t_1, t_2]
\]
implies \{\text{definition of } \cap \text{ and definition of } T_{\mu(\mathcal{E})}^{\mathcal{K}}\}
\[
\mu((A, J)) \in T_{\mu(\mathcal{Q})}^{\mathcal{K}}(\mu(\mathcal{I})) \land T_{\mu(\mathcal{R})}^{\mathcal{K}}(\mu(\mathcal{I})) = T_{\mu(\mathcal{Q}) \cap \mu(\mathcal{R})}^{\mathcal{K}}(\mu(\mathcal{I})) = T_{\mu(\mathcal{Q}) \cap \mu(\mathcal{R})}^{\mathcal{K}}(\mu(\mathcal{I}))
\]

The lemma proves the desired property only for the immediate consequences of a program expression, the following theorem generalizes the result for all consequences belonging to the least fixpoint semantics of a program expression.

**Theorem 7** Let \( \mathcal{E} \) be a program expression and \( \mathcal{I} \subseteq \mathcal{B}_L \times \text{Int} \). For any object level atomic formula \( A \) and any ground interval \( J \), the following statement holds
\[
(A, J) \in (T_{\mathcal{E}}^{\text{int}})^{\omega} \implies \mu((A, J)) \in (T_{\mu(\mathcal{E})}^{\mathcal{K}})^{\omega}.
\]

**Proof** We first show that for all \( h \)
\[
(A, J) \in (T_{\mathcal{E}}^{\text{int}})^h \implies \mu((A, J)) \in (T_{\mu(\mathcal{E})}^{\mathcal{K}})^h \tag{5.35}
\]

The proof is by induction on \( h \).

**(Base case).** Trivial since \((T_{\mathcal{E}}^{\text{int}})^0 = \emptyset\).

**(Inductive case).** Assume that
\[
(A, J) \in (T_{\mathcal{E}}^{\text{int}})^h \implies \mu((A, J)) \in (T_{\mu(\mathcal{E})}^{\mathcal{K}})^h
\]

Suppose \((A, J) \in (T_{\mathcal{E}}^{\text{int}})^{h+1}\). By definition of the powers of the function \( T_{\mathcal{E}}^{\text{int}} \), we have \((A, J) \in T_{\mathcal{E}}^{\text{int}}((T_{\mathcal{E}}^{\text{int}})^h)\). Now by using Lemma 1, \( \mu((A, J)) \in T_{\mu(\mathcal{E})}^{\mathcal{K}}(\mu((T_{\mathcal{E}}^{\text{int}})^h)) \). By inductive hypothesis for all \((B, K) \in (T_{\mathcal{E}}^{\text{int}})^h\), we have \( \mu((B, K)) \in (T_{\mu(\mathcal{E})}^{\mathcal{K}})^h \), hence \( \mu((T_{\mathcal{E}}^{\text{int}})^h) \subseteq (T_{\mu(\mathcal{E})}^{\mathcal{K}})^h \). Since the function \( T_{\mu(\mathcal{E})}^{\mathcal{K}} \) is monotonic we can conclude that \( \mu((A, J)) \in T_{\mu(\mathcal{E})}^{\mathcal{K}}((T_{\mu(\mathcal{E})}^{\mathcal{K}})^h) \).

Now we can prove the theorem. Suppose \((A, J) \in (T_{\mathcal{E}}^{\text{int}})^{\omega}\), then there exists \( h \) such that \((A, J) \in (T_{\mathcal{E}}^{\text{int}})^h\). By the previous statement (5.35) we have \( \mu((A, J)) \in (T_{\mu(\mathcal{E})}^{\mathcal{K}})^h \) and this allows us to conclude \( \mu((A, J)) \in (T_{\mu(\mathcal{E})}^{\mathcal{K}})^{\omega} \) because of the definition of the least fixpoint of \( T_{\mu(\mathcal{E})}^{\mathcal{K}} \).

It is worth noting that the least fixpoint semantics of the translated program expression \( \mu(\mathcal{E}) \) properly extends the corresponding semantics of \( \mathcal{E} \). In fact, consider the following program \( P \) and its translation \( \mu(P) \)
\[
P : q \Box [1, 2], \quad \mu(P) : q \text{th}[1, 2], \\
q \Box [2, 3], \quad \quad \quad q \text{th}[2, 3].
\]

\(^4\) In case of unit annotated clauses, \( A \leftarrow \Box [s_1, s_2] \), the translation can be simplified to a MuTACLP fact \( A \text{th}[s_1, s_2] \).
The least fixpoint semantics of $P$ is $(T^\text{int}_P)^\omega = \{(q, \{1, 2\}), (q, \{2, 3\})\}$ whereas the least fixpoint semantics of $\mu(P)$ contains also the annotated atom $(q, \text{th}[1, 3])$, i.e., $(T^\text{\mu}_P)^\omega \supseteq \{(q, \text{th}[1, 2]), (q, \text{th}[2, 3]), (q, \text{th}[1, 3])\}$, because $T^\text{\mu}_P$ implements also the rule (L) which allows one to join overlapping $\text{th}$ annotations. Furthermore, Theorem 3 ensures that even top-down we cannot compute such an annotated atom, because the meta-interpreter can deduce atoms annotated with intervals $[1, 2]$, $[2, 3]$ or with non-empty subintervals of them.

### 5.6 Conclusion

Our research on time led to the definition of MuTACL. This language joins the advantages of TACL in handling temporal information with the ability to structure and compose programs. Thus we can describe discrete or continuous time, work with time points and time periods, and model definite, indefinite and periodic temporal information, which can be distributed among different theories. Representing knowledge in separate programs allow us to use knowledge from different sources. Thus information can be stored at different sites and combined in a modular way by employing the meta-level operations. Moreover, this modular approach favors the reuse of the knowledge encoded in the programs for future applications.

The search for a bottom-up semantics for MuTACL stimulated the definition of a bottom-up semantics for TACL. In fact, while top-down semantics were known from the early work on TACL [48, 49, 50], the bottom-up semantics has been studied only recently by Raffaetà and Frühwirth in [96, 95] and, consequently, for the first time soundness and completeness results for TACL have been provided.

Still concerning semantic issues, it would be interesting to investigate on different definitions of the immediate consequence operator, for instance by considering an operator similar to the function $R_P$ for generalized annotated programs [70]. Alternative solutions, based on a different choice of the semantic domain where the immediate consequence operator is defined, thus on a different kind of interpretation, may consider a more abstract domain. This domain is obtained by endowing $C\text{-base}_L \times \text{Ann}$ with the product order (induced by the identity relation on $C\text{-base}_L$ and the order on Ann) and then by taking as elements of powerdomain only those subsets of annotated atoms which satisfy some closure properties with respect to such an order. For instance, one can require “downward-closedness”, which amounts to including subsumption in the $T_P$ operator. Another possible property is “limit-closedness”, namely the presence of the least upper bound of all directed sets which, from a computational point of view, amounts to consider computations which possibly require more than $\omega$ steps.

It is worth noting that TACL is presented as a particular instance of annotated constraint logic (ACL) [48, 50] for reasoning about time, then, similarly, we could consider MuTACL as an instance of MuACL, i.e., Multi-theory Annotated Constraint Logic. In fact, we can reformulate our construction starting from the more
general paradigm of ACL where atoms can be labelled by a general class of annotations. In defining MuACL we have to further require that the class of annotations forms a lattice, in order to have both upper bounds and lower bounds. We recall that lower bounds are necessary for the definition of the intersection operation. Under the assumptions that only atoms can be annotated and clauses are free of negation, we can smoothly generalize both the meta-interpreter (Subsection 5.3.2) and the immediate consequence operator (Definition 8) to deal with general annotations. In this way we extend ACL with high-level mechanisms for structuring programs and for combining them.

A critical point is the handling of negation. In the line of Frühwirth, we can embody the “negation by default” of logic programming into MuTACL by exploiting the logical equalities proved in [50]:

\[(\neg A \text{ th } I) \iff \neg (A \text{ in } I), \quad (\neg A \text{ in } I) \iff \neg (A \text{ th } I)\]

Consequently, the meta-interpreter is extended with two clauses which use such equalities:

\[
demo(\mathcal{E}, (-A \text{ th } I)) \leftarrow \neg \text{ demo}(\mathcal{E}, A \text{ in } I)
\]

\[
demo(\mathcal{E}, (-A \text{ in } I)) \leftarrow \neg \text{ demo}(\mathcal{E}, A \text{ th } I)
\]

However the interactions between negation by default and program composition operations are still to be fully understood. Some results on the semantic interactions between operations and negation by default are presented in [22], where, nevertheless, the handling of time is not considered. The investigation on negation is left as future work.

In Section 5.2.3 we have already shown the main features of handling time in our approach, in respect to how it is dealt with in the literature. Here we focus on the multi-theory setting, i.e., on the possibility, offered by MuTACL, to structure and compose (temporal) knowledge. There are few logic-based approaches that provide the user with these tools. For instance, most logical languages discussed in Section 3.1.1 lack of these powerful means to represent and reuse knowledge. An exception is Temporal Datalog [88], an extension of Datalog based on a simple temporal logic with two temporal operators, namely first and next. It introduces a notion of module but this mechanism seems not to be used as a knowledge representation tool but to define new non-standard algebraic operators. In fact, to query a temporal Datalog program, Orgun proposes a “point-wise extension” of the relational algebra upon the set of natural numbers, called TRA-algebra. Then he provides a mechanism for specifying generic modules. A generic module, called temporal module, is a parametric Temporal Datalog program, with a number of input predicates (parameters) and an output predicate, so that the same definitions inside the module can be used for various relations. This module can be regarded as an operator which, given a temporal relation, returns a temporal relation. Thus temporal modules are indeed used as operators of TRA, through which one has access to the use of recursion, arithmetic predicates and temporal operators.
A multi-theory framework, based on *annotated logics*, in which temporal information can be handled, is proposed by Subrahmanian [110]. This is a very general framework aimed at amalgamating multiple knowledge bases which can also contain temporal information. These knowledge bases are generalized annotated programs (GAP) [70] and in this context temporal information is modeled by using an appropriate lattice of annotations. In order to integrate these programs, a so called *Mediatory Database* [3] is given which is a GAP having clauses of the form

$$A_0 : [m, \mu] \leftarrow A_1 : [D_1, \mu_1], \ldots, A_n : [D_n, \mu_n]$$

where $D_i$ is a set of database names, i.e., the names of the knowledge bases. Intuitively, ground instances of clauses in the mediator say: “If the databases in set $D_i$, $1 \leq i \leq n$, (jointly) imply that the truth value of $A_i$ is at least $\mu_i$, then the mediator will conclude that the truth value of $A_0$ is at least $\mu$”. In practice the fundamental mechanism provided to combine knowledge bases is a kind of message passing. Roughly speaking, the resolution of an atom $A_i : [D_i, \mu_i]$ is delegated to different databases, $D_i$, and the annotation $\mu_i$ is obtained by considering the least upper bounds of the annotations of $A_i$ computed in the distinct databases. Our approach is quite different because the meta-level composition operations offer not only the access to the relation defined by a predicate but also to the definitions of the predicate. For instance $P \cup Q$ is equivalent to a program whose clauses are the union of the clauses of $P$ and $Q$. Thus the information which can be derived from $P \cup Q$ is greater than the union of what we can derive from $P$ and $Q$ separately.
Chapter 6

Modeling spatial data in MuTACLPL

This chapter analyses the problem of representing and reasoning on spatial data in MuTACLPL. A basic observation is that, as already pointed out in the field of constraint databases [16, 17, 56, 57, 93, 36, 38], constraints are a natural and powerful means for modeling spatial information. At the data level, constraints are able to finitely represent possibly infinite sets, while as a data model they offer a paradigm for the representation of different sorts of data in a unified framework. In this way, a spatial object, seen as an infinite set of points, can be dealt with as a first class citizen with an explicit representation. For instance a convex polygon, which is the intersection of a set of half-planes, is defined by the conjunction of the inequalities defining each half-plane, while a non convex polygon is modeled as the union (logical disjunction) of a set of convex polygons. The additional features offered by MuTACLPL like the deductive capabilities of the language, the possibility of structuring the knowledge in different theories which can be variously combined, and the explicit representation and treatment of temporal information, definitively increase the expressive power of the constraint-based model.

In this chapter we first present some simple examples aimed at giving evidence to the adequacy of MuTACLPL for modeling spatial data. We suggest how it is possible to represent and reason about spatial information and we point out the ability supplied by MuTACLPL of establishing spatio-temporal correlations. In such examples we assume that spatial data are represented in the proposed language itself. In practice, and more commonly, data must be recovered from already existing sources where the information is stored by using a different, non-logical encoding. A very important example is given by Geographical Information Systems (GIS) which are computer-based information systems that enable to capture, model, manipulate, retrieve, analyze and visualize geographically referenced data [117]. A well-known problem of GIS is the fact that these systems provide a large set of analysis functions, whose syntax and semantics are not intuitive for a non-expert end-user [117, 39, 94]. We show how MuTACLPL can be exploited to integrate GIS technology and
constraint logic programming, in order to supply the user with a declarative language which supports and improves GIS analysis at least at the specification level. After reviewing a related approach proposed by Aquilino et al. [11, 9, 97], which enriches logic programming with built-in atoms mapped into GIS functions, we illustrate our proposal [82] which, relying on MuTACLp, gives an explicit logical representation to the spatial data stored in GIS. In this way, all the previously discussed capabilities of MuTACLp can be used to reason on the data contained in a GIS. Remarkably, spatial data can be associated with temporal information, a great advantage with respect to the current GIS technology since time is almost completely ignored in current GIS, although it is recognized to be an essential component of geographical information [78, 117]. Furthermore, having a uniform representation of data in a common model favors the interoperability among different GIS which is nowadays very difficult and a hot research area [117, 37, 38]. Observe that our approach can be considered as an extension of the approach in [9, 97] since we can think of maintaining the built-in atoms in the language, and, according to the kind of analysis which is required, to choose whether to invoke directly GIS functions or to manipulate spatial data at the logical level.

The chapter is organized as follows. Section 6.1 illustrates how spatial data can be naturally modeled and combined with temporal information in our framework by means of constraints. Inspired by a former approach [11, 9], Section 6.2 proposes an integration of GIS technology with (constraint) logic programming where GIS functions can be directly invoked inside a program. Section 6.3 provides spatial objects stored in a GIS with a constraint-based representation, which can be expressed and handled inside MuTACLp, and it presents a set of facilities to cope with the complexity of spatial data. In Section 6.4 some examples are given to focus on how MuTACLp can support GIS analysis. Finally, Section 6.5 draws some conclusions.

6.1 Introducing spatial information: the simple way

We start by considering an example which illustrates how spatial data can be naturally modeled and integrated with temporal information in our framework by means of constraints. The example is taken from [57], where moving objects with time-varying activities are modeled by using the system DEDALE which relies on a logical model based on linear constraints, generalizing the constraint database model of [68]. We will discuss how the same kind of information can be expressed in MuTACLp, and we will compare the two approaches.

In [57] time is associated with attributes of relations. For instance, the relation schema People is defined as
$$People = \{\{\text{name} : \text{string},
    \text{category} : \text{string},
    \text{activity} : \{[\text{alpha} : \text{string}, \text{time} : \mathbb{Q}]\},
    \text{trajectory} : \{[\text{space} : \mathbb{Q}^2, \text{time} : \mathbb{Q}]\}\}$$

where the trajectory is described by partitioning the day in time slices and associating with each slice the position(s) where the person is likely to be found during such time slice. In our approach this information can be modeled by associating time with tuples. Thus we describe a person by his/her name, his/her category, his/her activity and his/her position(s) in a certain time interval. For instance assume that John is a tourist and from 1am to 10am he sleeps, from 11am to 12am he has breakfast and then in the afternoon he goes skiing up to 4pm, while Monica is a journalist and from noon to 4pm she skis. This can be expressed by means of the following clauses.

\begin{align*}
\text{person}(john, \text{tourist, sleep, X, Y}) & \text{ th } [1am, 10am] \leftarrow X = 1230, Y = 134 \\
\text{person}(john, \text{tourist, eat, X, Y}) & \text{ th } [11am, 12am] \leftarrow X = 1230, Y = 120 \\
\text{person}(john, \text{tourist, ski, X, Y}) & \text{ th } [12am, 4pm] \leftarrow 3X - 2Y \leq 49, 23X + 2Y \geq 134, \ldots \\
\text{person}(monica, \text{journalist, ski, X, Y}) & \text{ th } [1pm, 4pm] \leftarrow \ldots
\end{align*}

The temporal information is represented by a th annotation because the property holds throughout the time period, whereas the position is expressed by a constraint on the variables \(X, Y\).

Furthermore, a resort can be described by its name and its geometry (shape and position).

\begin{align*}
\text{resort}(\text{terrace, X, Y}) & \leftarrow X \leq 5, X \geq 3, Y \leq 10, Y \geq 8 \\
\text{resort}(\text{ski, X, Y}) & \leftarrow \ldots
\end{align*}

Now we present some queries from [57] involving the spatial and/or temporal knowledge and we show how they can be formulated in our language.

1. Where is John between 12am and 2pm?

\begin{align*}
\text{person}(john, \_ X, Y) & \text{ in } [12am, 2pm] \\
\text{The answer to this query consists of all movings of John occurred in that time period. We use the in annotation because we want to capture all the different positions that John assumes between 12am to 2pm. If we asked for person}(john, \_ X, Y) & \text{ th } [12am, 2pm] \text{ then we would have constrained John to stay in only one place for the whole time period.}
\end{align*}

2. When does Monica stay at the bar’s terrace?

\begin{align*}
\text{person}(monica, \_ X, Y) & \text{ th } [T_1, T_2], \text{resort}(\text{terrace, X, Y})
\end{align*}
This is a spatial join obtained by imposing that \(X, Y\) satisfies both the constraints related to person\((\ldots)\) and to resort\((\ldots)\). The result is the time interval \([t_1, t_2]\) in which Monica’s position intersects the terrace area. Notice that since the position generally represents an area where a person is likely to be found, this query, which corresponds to the one in [57] returning the same answer, more precisely asks in which time period(s) Monica is likely at the terrace.

3. Where is John while Monica is at the bar’s terrace?

\[
person(john, \_X, Y) \text{th} \ [T_1, T_2], \ person(monica, \_V, W) \text{th} \ [T_1, T_2], \ resort(terrace, V, W)
\]

This query is a composition of a spatial join and a temporal join.

4. In which places does Monica sleep?

\[
person(monica, \_sleep, X, Y) \text{th} \ [\_]
\]

5. Where did Monica and John meet?

\[
person(john, \_X, Y) \text{th} \ [T_1, T_2], \ person(monica, \_X, Y) \text{th} \ [T_1, T_2]
\]

Two people meet if they are in the same place in the same time interval.

6. Who ate in the skiing area, and when?

\[
person(P, \_cat, X, Y) \text{th} \ [T_1, T_2], \ resort(ski, X, Y)
\]

The above examples suggest how the approach by Grumbach et al. can be simulated in MuTACLP. A difference concerns the kind of temporal information which can be modeled. Grumbach et al. succeed in representing only definite temporal data, corresponding to our \text{th} annotations, while by using MuTACLP programs we have the possibility to express also periodic and indefinite temporal information. For instance, the fact that on Wednesdays Frank works in Florence is simply stated by the clause

\[
person(frank, employee, work, X, Y) \text{at} \ T \leftarrow \ \text{wed at} \ T, \ resort(florence, X, Y)
\]

The predicate \text{wed}, which appears in the body, has been previously defined (in Section 5.4) and checks if \(T\) is Wednesday, while \text{resort}(florence, X, Y)\) is defined by

\[
\text{resort}(florence, X, Y) \leftarrow c(X, Y)
\]

where \(c(X, Y)\) is a constraint specifying the position of the town.

Furthermore, to represent the fact that Frank has a break between 5pm and 6pm (indefinite temporal information) at the bar’s terrace we use the clause

\[
person(frank, employee, break, X, Y) \text{in} \ [5pm, 6pm] \leftarrow \ \text{resort}(terrace, X, Y)
\]
It is worth noting that we can also describe time-varying areas. For instance the area flooded by the water depicted in Figure 3.7, where the front end of the tide water is a linear function of time, is modeled in MuTACL by the following clause

\[
\text{floodedarea}(X, Y) \at T \leftarrow 1 \leq Y, Y \leq 10, 3 \leq X, X \leq 10, Y \geq X + 8 - T
\]

In Grumbach et al. the temporal and the spatial dimensions are independent, thus it is not possible to express spatial relations which are parametric with respect to time as in this case. Their concept of moving objects captures only discrete changes.

Moreover, MuTACL offers a framework to work with and compose different programs. The next example points out just the usefulness of having multiple theories.

**Example 15** Suppose that two distinct databases maintain information about the building areas of a country (specifying if they enjoy this property, the name of the owner and the location), and the density of the population of the country subdivided in zones, respectively.

**CADASTRE:**

\[
\text{area(ok, john, } X, Y) \leftarrow \ldots
\]

\[
\text{area(no, mary, } X, Y) \leftarrow \ldots
\]

**DENSITY:**

\[
\text{density(10, } X, Y) \leftarrow \ldots
\]

\[
\text{density(79, } X, Y) \leftarrow \ldots
\]

where \ldots stands for constraints on the variables \(X, Y\) expressing the position of each area.

Then if we join the theories modeling such databases we can ask: “Which are the building zones with a density less or equal to 10%?” through the query

\[
\text{demo(CADASTRE } \cup \text{ DENSITY, (area(ok, } X, Y), \text{density(D, } X, Y), D \leq 10))
\]

This query makes a spatial join between the areas in the cadastral database and the ones in the density database, returning a constraint in \(X, Y\) which represents a zone with the desired features.

An advantage of [57] is that it allows for a more compact representation of the information by means of disjunctive constraints. On the other hand, here we use only conjunctive constraints, and we model disjunction by defining different clauses, one for each disjunct. Thus we need backtracking to compute all the solutions. Moreover, for the absence of negation, MuTACL does not allow us to express those database operations requiring negation, like difference between relations. While in principle we can employ disjunctive constraints in MuTACL (only an adequate constraint solver is required), the treatment of negation is not straightforward and, as already mentioned, represents an interesting topic of future research.
6.2 Declarative GIS analysis: a preliminary investigation

The examples of spatio-temporal reasoning in MuTACL, described in the previous section, rely on the assumption that spatial data have already a constraint-based representation. However, as stated before, in order to deal with real applications it is important to make our framework able to manipulate spatial data stored in existing sources of knowledge, like Geographical Information Systems (GIS). One of our goals is to improve analysis in GIS by using MuTACL. In fact, one of the frequently stated problems for GIS is that these systems have a complex functionality which is not accessible to non expert end-users. Today GIS user interfaces are not easy to use and require much time to get used to them [117, 39, 94]. Thus a user often knows what she/he wants, but does not know how to obtain it from the GIS. Moreover, the current GIS analysis approaches are not general and reusable. In fact analysis is typically based on complex procedural algorithms, and data are built and structured for the specific application. We are convinced that a declarative approach is a better solution for solving this kind of problems at least at the specification level.

In the literature we can find several attempts to exploit the deductive capabilities of logics to reason on geographic data [117, 2, 32, 9]. Some approaches go towards the use of Artificial Intelligence techniques (expert systems) to reason about geographical data. Other approaches express spatial data in an object-oriented paradigm and build a deductive part to infer knowledge from the spatial objects [2]. Here we aim at exploiting the capability of combining logic theories in order to capture constraints and rules of the application. Furthermore, we look at the logical layer as the programming language for the analysis of data actually held in GIS systems, rather than as a way of specifying a complete GIS application.

In this section we first describe an approach, appeared in [11, 9, 97], which represents a first step for providing the user with a declarative language for GIS analysis. It basically relies on (an enriched) language of program expressions and on the introduction of special atoms which can be used to invoke GIS functions. We see that such approach can be smoothly recast in MuTACL, where the specific features of the language immediately give a greater expressive power (e.g., fine spatio-temporal analysis). The approaches in this section still present some limits such as the impossibility of directly manipulating the representation of spatial data and the absence of a uniform representation of data. A possible different solution which overcome these problems will be discussed in the next section.

6.2.1 Integration with GIS by invoking GIS functions

This approach was proposed by Aquilino et al. in [11, 97, 9] and it is based on the language of program expressions enriched with the message passing mechanism wrt (Section 4.3). The resulting language is called MedLan. The key point of the
work relies on defining the interface between the logical theories that represent the analysis rules and the GIS analysis functions. As we will see, some logical atoms are mapped into functions provided by the GIS. We describe the approach through an example taken from [11].

**Example 16** A logging company has been given a license to cut down trees in the Oakwood area. However the license has placed restrictions on where the company can cut down trees. The company will use a GIS system to devise the best place to start logging.

**License restrictions**

1. No tree may be cut down within 1km of the sea, the lake or any river, in order to prevent land erosion.

2. The logging sites must be within 5km of existing roads for easy access by heavy logging equipment since conservation laws will not allow any new roads to be built in this area.

On top of these restrictions the company also has its own selection rules.

**Company site selection rules**

1. The logging sites must be below the snow line (1000m) as heavy logging equipment cannot operate in snowbound areas.

2. Heavy logging equipment cannot operate in any area with a slope greater than 5 degrees.

Each set of rules corresponds to a logic theory representing the criteria of the analysis.

**License**

allow_cut_trees(X) ← water(W),

   distance(X,W,K2), K2>1

logging_site(X) ← road(Y), distance(X,Y,K), K<5

**Company-rules**

logging_site(X) ← contour_line(X,Y), Y < 1000,

slope(X,k), K≤5

The language provides a set of operations to combine knowledge from logic theories. In this example the "message passing" mechanism is employed to evaluate a query in a given theory and the intersection operation to restrict the company rules by means of the environmental rules.
Top-rules

\[
\text{best\_place}(X) \leftarrow \text{logging\_site}(X) \text{ wrt Company-rules } \cap \text{License},
\]

\[
\text{allow\_cut\_trees}(X) \text{ wrt License}
\]

The program Top-rules models the criteria which establish the best place to start logging. The best place is an area \( X \) which respects both the license restrictions and the company selection rules. The atom \( \text{logging\_site}(X) \) is evaluated in the program expression \( \text{Company-rules } \cap \text{License} \) since it should satisfy the company general rules and the license restrictions that are the constraining rules to the general case. We ask the system to find out the best place to start logging through the query

\[
\text{best\_place}(X) \text{ wrt Top-rules}
\]

The crucial point of this approach is that some predicates, the ones in typewriter font, are directly implemented as GIS analysis functions. For example, the evaluation of the predicates \( \text{water, road} \) corresponds to the combination of GIS overlay and extract operations which select the water, road layer respectively and it is assumed that the layers returned by the GIS are conceptually equivalent to set of answers in the deductive database world. GIS use buffering to represent distances of areas, so the evaluation of the goals \( \text{distance}(X, Y, K), K < d \) is assumed to correspond to a buffering operation. The evaluation of the \( \text{slope} \) predicate requires the access to the GIS functions that return data about the slopes of the area. For more details about the implementation, we refer the reader to [97, 9].

Summing up, as shown in Figure 6.1, the system is composed by three parts: the analysis rules, the interface with the GIS and the analysis functions provided by the GIS itself.

Replacing MedLan with MuTACLp

The approach just described puts in evidence that adding a declarative programming layer on top of Geographical Information Systems can allow a better use of them, essentially because a declarative approach is much closer to the natural ways of expressing analysis rules than a procedural approach. First of all, since (constraint) logic programming is essentially a rule-based system, it is possible to build GIS applications with an expert system flavor. Even the small example in the previous subsection shows how naturally company procedures and business rules can be represented in the language. Furthermore, the multi-theory framework, that provide tools for combining different knowledge components, coded in different programs, seems particularly suited to handle the naturally fragmented knowledge used in GIS analysis.

An analogous architecture, keeping the mentioned advantages can be adopted in our framework simply by replacing MedLan with MuTACLp. In fact all the features of MedLan are present in MuTACLp, with the only exception of the message passing
construct which, however, can be smoothly introduced, as discussed in the next chapter (see Section 7.1.2).

The greater expressiveness of MuTACL can be fruitfully exploited. The built-in atoms are still mapped into GIS functions, but now we have the possibility to represent and reason on temporal information and to express spatio-temporal correlations. For instance, we can model that during the winter an animal lives in a certain area whereas in the summer it moves in another zone, by using clauses of the form

\[
\text{animal}(X,A)_\text{th}[\text{jan},\text{mar}] \leftarrow \text{criteria to compute } A
\]

\[
\text{animal}(X,A)_\text{th}[\text{jun},\text{sep}] \leftarrow \text{criteria to compute } A
\]

The criteria invoke GIS functions, such as computation of layers that have a particular kind of vegetation, or buffering operations to compute the distance from sources of noise, all elements which can be relevant to determine whether or not an area is a favourable habitat for the animal, and which can change along the year.

This is a great advantage and an added functionality with respect to current GIS technology. In fact these systems are often based on a static two-dimensional view of the world, whereas in many applications it is required to work with dynamic information modeling phenomena as they evolve through the time [78, 117]. As reported by Langran [78], many researchers depict the elements of a GIS as time, location, and attribute, although today’s technology treats only location and attribute. Therefore our declarative programming layer on top of GIS can be a way to
relate time to the data stored in GIS, and, in any case, a natural means to represent time-varying analysis rules, as largely pointed out in the previous chapter (e.g., see Section 5.4).

6.3 Declarative GIS analysis in MuTACLPL

In this section we illustrate a different solution for using MuTACLPL in GIS analysis [82]. It consists of replacing not only MedLan with MuTACLPL, but also the built-in predicates with an explicit representation of the GIS spatial data in MuTACLPL. In other words, instead of having a block Mapping to GIS functions, now the system has a block Translation process (shown in Figure 6.2) which provides a spatial object stored in a GIS with a constraint-based representation, which can be expressed and handled inside MuTACLPL. The main advantage is that we have a declarative language where spatio-temporal and thematic information can be represented in a uniform way and we can exploit the features of constraint logic programming such as recursion and constraint handling to perform sophisticated spatio-temporal reasoning. Moreover, this unifying language is also promising to support the interoperability among different GIS. For a general discussion on interoperability the reader is referred to the next chapter.

Notice that, although not explicitly discussed in this section, the new approach can be integrated with the previous one to offer the possibility of directly invoking GIS functions. This may be relevant if one is interested in efficiency issues. In fact the logical representation and manipulation of spatial data is computationally expensive, and therefore one can think of employing it only when it is strictly necessary for the analysis, resorting to the GIS functions in the other cases.

In Section 6.3.1 we discuss the logical representation of GIS data, then in Section 6.3.2 we show how the classical set-theoretic operations on spatial objects (union, intersection, etc.) can be defined in MuTACLPL, and in Section 6.3.3 we discuss a logical reconstruction of GIS layers. Finally in Section 6.3.4 we present a module which collects all the features given in the section for handling spatial data.

We focus our attention on 2-dimensional spatial objects, i.e., on objects in the plane.

6.3.1 A logical representation of GIS spatial data

In current GIS there are two basic modes for representing objects in the plane, namely the raster and vector mode. Briefly we recall that in the Raster Model [99, 58] an object is given by a finite number of raster points. Raster points are uniformly distributed following an easy geometric pattern, which is normally a square. In the Vector Model or Spaghetti Model [79] an object is intensionally deduced from its contour, which is approximated by segments and points, and it is usually represented by a list of points (see Section 3.2 for more details).
Figure 6.2: The translation process from GIS to MuTACLp.

We propose an automatic translation of the spatial data stored in a GIS into MuTACLp programs, assuming that spatial data are represented according to the Spaghetti Model. Observe that this assumption is reasonable because such a model is very popular and there are also GIS functions that convert data from the Raster Model into the Spaghetti Model and vice versa. Under this hypothesis the translation process consists of two steps:

1. a spatial object is triangulated, and each triangle is described by an identifier and by its three vertexes via a unit clause.

2. By using the implicit representation of a triangle, as determined in Step 1, i.e., its vertexes, we build a constraint which explicitly defines the set of points belonging to it. Finally, a spatial object is denoted by an expression which represents the union of the triangles which compose the object.

Figure 6.2 summarizes the steps of the translation process.

First step

In the Spaghetti model, a GIS object is uniquely determined by an identifier and it has some thematic attributes and a spatial component consisting of a list of points modeling its contour. In the following, we focus on the translation of the spatial component of the object because thematic attributes do not require any particular treatment, and they can be represented by themselves. By using an algorithm to triangulate the object we obtain a set of triangles which approximate it. As a result each object is divided into triangles, each represented by an identifier and by its three vertexes. Triangles can be degenerate, i.e., points and segments. We assume that
• points are represented by triangles with three identical vertexes;

• segments are represented by triangles with the first two vertexes distinct (the end-points of the segment) and the third one equal to the second one, and

• proper triangles are represented by three distinct vertexes.

We suppose that if the algorithm returns a triangle with three distinct vertexes, then such points are not all collinear.¹

Now we can define a set of clauses providing the logical representation of the object in the 2-Spaghetti Model.

**Definition 10 (2-Spaghetti Model)** An object identified by objId and decomposed into n triangles, with identifier trId, and vertexes \((x_1^i, y_1^i), (x_2^i, y_2^i), (x_3^i, y_3^i)\), for \(i = 1, \ldots, n\), is represented by the following unit clauses:

- \(n_{\tri}(\text{objId}, n)\).

- \(\tri(\text{objId}, i, \text{trId}_i, x_1^i, y_1^i, x_2^i, y_2^i, x_3^i, y_3^i)\). for \(i = 1, \ldots, n\).

The first clause states that the object objId is composed by \(n\) triangles while the predicate \(\tri\) defines the \(n\) triangles composing the object. A \(\tri\) clause states that the triangle \(\text{trId}_i\) is the \(i^{th}\) triangle of the object objId and its vertexes are \(x_1^i, y_1^i, x_2^i, y_2^i, x_3^i, y_3^i\). The identifier \(\text{trId}_i\) is a global (unique) identifier.

**Example 17** The objects depicted in Figure 6.3 are represented according to the above 2-Spaghetti Model as follows:

\[
\begin{align*}
n_{\tri}(\text{point}, 1). & \quad \tri(\text{point}, 0, p1, 2, 6, 2, 6, 2, 6). \\
n_{\tri}(\text{square}, 2). & \quad \tri(\text{square}, 0, \text{tr1}, 1, 3, 1, 1, 3, 1). \quad \tri(\text{square}, 1, \text{tr2}, 3, 1, 3, 1, 3, 1). \\
n_{\tri}(\text{polyline}, 3). & \quad \tri(\text{polyline}, 0, \text{pl1}, 4, 5, 5, 3, 5, 3). \quad \tri(\text{polyline}, 1, \text{pl2}, 5, 3, 6, 3, 6, 3). \\
& \quad \tri(\text{polyline}, 2, \text{pl3}, 6, 3, 7, 2, 7, 2).
\end{align*}
\]

The first clause states that the object point consists of a single triangle. Indeed it is degenerate since it is a point and it is represented by the unit clause \(\tri\) where all the vertexes coincide with the coordinates of the point. The object square is triangulated into two non-degenerate triangles with identifiers \(\text{tr1}\) and \(\text{tr2}\) having vertexes \((1, 3), (1, 1), (3, 1)\) and \((3, 1), (3, 3), (1, 3)\) respectively. Finally, the object polyline is composed by three segments. Each segment has a unique identifier \((\text{pl1}, \text{pl2}, \text{pl3}\) respectively) and it is represented by the unit clause \(\tri\) where the first two vertexes are the end-points of the segment and the third one is equal to the second one.

¹In the implementation we used the algorithm for triangulating objects that can be downloaded from [http://www.cs.unc.edu/~dm/CODE/GEM/chapter.html](http://www.cs.unc.edu/~dm/CODE/GEM/chapter.html).
Indeed this representation of spatial objects is an intermediate step to obtain a representation based on linear constraints. The second kind of representation has the advantage of giving an explicit characterization of the points which belong to the considered object (e.g. a polygon is explicitly the infinite set of points it contains versus the implicit definition by means of the sequence of border points). Therefore, it allows us to manipulate spatial objects through standard set operations, such as union, intersection, difference etc.

Second step

As objects are composed by triangles, first we describe how we can obtain a constraint from the 2-spaghetti representation of a triangle. We consider a non-degenerate triangle; the translation of points and line segments (degenerate triangles) is shown in Section 6.3.4. The idea, also exploited by Chomicz and Revesz [38] and by Grumbach et al. in [56], is that a triangle, which is the intersection of three half-planes, can be defined as the conjunction of the inequalities defining each half-plane. The predicate \textit{findSide} expresses the constraint for a half-plane.

\[
\text{findSide}(X, Y, X_{1}, Y_{1}, X_{2}, Y_{2}, X_{3}, Y_{3}) \leftarrow (Y_{3} - Y_{1})(X_{2} - X_{1}) \geq (Y_{2} - Y_{1})(X_{3} - X_{1}),
\]
\[
(Y - Y_{1})(X_{2} - X_{1}) \geq (Y_{2} - Y_{1})(X - X_{1}).
\]

\[
\text{findSide}(X, Y, X_{1}, Y_{1}, X_{2}, Y_{2}, X_{3}, Y_{3}) \leftarrow (Y_{3} - Y_{1})(X_{2} - X_{1}) \leq (Y_{2} - Y_{1})(X_{3} - X_{1}),
\]
\[
(Y - Y_{1})(X_{2} - X_{1}) \leq (Y_{2} - Y_{1})(X - X_{1}).
\]

The constraints in the clause body express that all the points of the triangle are above (or below) the line segment defined by the points \((X_{1}, Y_{1})\) and \((X_{2}, Y_{2})\) and
they are in the same half-plane of \((X_3, Y_3)\) (Figure 6.4).

In order to find the points belonging to a non-degenerate triangle we simply intersect the three half-planes built from the lines crossing each couple of the triangle vertexes.

\[
\text{tri}_\text{con}(\text{TrId}, X, Y) \leftarrow \text{tri}(\text{TrId}, X_1, Y_1, X_2, Y_2, X_3, Y_3),
\]
\[
\text{distinct}(X_1, Y_1, X_2, Y_2, X_3, Y_3),
\]
\[
\text{findSide}(X, Y, X_1, Y_1, X_2, Y_2, X_3, Y_3),
\]
\[
\text{findSide}(X, Y, X_2, Y_2, X_3, Y_3, X_1, Y_1),
\]
\[
\text{findSide}(X, Y, X_3, Y_3, X_1, Y_1, X_2, Y_2)
\]

where \text{distinct} is a predicate that checks that the vertexes are all distinct

\[
\text{distinct}(X_1, Y_1, X_2, Y_2, X_3, Y_3) \leftarrow \text{non-equal}(X_1, Y_1, X_2, Y_2),
\]
\[
\text{non-equal}(X_2, Y_2, X_3, Y_3),
\]
\[
\text{non-equal}(X_3, Y_3, X_1, Y_1)
\]

\[
\text{non-equal}(X, Y, X', Y') \leftarrow X \neq X'
\]
\[
\text{non-equal}(X, Y, X', Y') \leftarrow Y \neq Y'
\]

The resolution of the atom \text{tri}_\text{con}(\text{TrId}, X, Y) provides the constraint representing all the points of the triangle \text{TrId}.

Now we can model a spatial object inside our logical framework: an object can be seen as the union of its triangles.

\[
\text{obj}(\text{ObjId}, \text{SpExp}) \leftarrow n_{\text{tri}}(\text{ObjId}, N),
\]
\[
\text{join}(\text{ObjId}, N - 1, \text{SpExp})
\]
\begin{align*}
    \text{join}(\text{ObjId}, 0, \text{TrId}) & \leftarrow \text{tri}(\text{ObjId}, 0, \text{TrId}, \cdots).
    \\
    \text{join}(\text{ObjId}, J, \text{SpExp} \oplus \text{TrId}) & \leftarrow \text{tri}(\text{ObjId}, J, \text{TrId}, \cdots).
    \\
    \text{join}(\text{ObjId}, J - 1, \text{SpExp})
\end{align*}

The expression \text{SpExp} (\text{SpExp} stands for \textit{Spatial Expression}) is a symbolic representation of the object and a means to recover the constraints associated with each triangle composing the object. The intended meaning of \(\oplus\) is set-theoretic union and it will be defined formally in the next section.

**Example 18** We want to represent the region in Figure 6.5. It is composed by six spatial objects, respectively identified by \(\text{ob1}, \ldots, \text{ob6}\), which are squares or rectangles. Each object is triangulated into two triangles, each with its own identifier. The logical representation of the region in the 2-spaghetti model is

\[
\begin{align*}
    n_{\text{tri}}(\text{ob1,2}). & \quad \text{tri}(\text{ob1,0,}tr1,2,7,2,5,7,5). & \quad \text{tri}(\text{ob1,1,}tr2,2,7,7,5,7,7). \\
    n_{\text{tri}}(\text{ob2,2}). & \quad \text{tri}(\text{ob2,0,}tr3,3,6,5,1,5,6). & \quad \text{tri}(\text{ob2,1,}tr4,3,6,3,1,5,1). \\
    n_{\text{tri}}(\text{ob3,2}). & \quad \text{tri}(\text{ob3,0,}tr5,1,4,4,1,4,4). & \quad \text{tri}(\text{ob3,1,}tr6,1,4,1,4,1). \\
    n_{\text{tri}}(\text{ob4,2}). & \quad \text{tri}(\text{ob4,0,}tr7,2,3,2,-5,6,-5). & \quad \text{tri}(\text{ob4,1,}tr8,2,3,6,-5,6,3). \\
    n_{\text{tri}}(\text{ob5,2}). & \quad \text{tri}(\text{ob5,0,}tr9,4,-2,4,-4,7,-4). & \quad \text{tri}(\text{ob5,1,}tr10,4,-2,7,-4,7,-2). \\
    n_{\text{tri}}(\text{ob6,2}). & \quad \text{tri}(\text{ob6,0,}t11,1,-3,3,-6,3,-3). & \quad \text{tri}(\text{ob6,1,}tr12,1,-3,1,-6,3,-6).
\end{align*}
\]

For instance the object \text{ob1} is composed by two triangles, identified by \(\text{tr1}\) and \(\text{tr2}\) having vertexes \((2,7), (2,5), (7,5)\) and \((2,7), (7,5), (7,7)\), respectively. If we compute the query

\[
\text{tri\_con(tr1, X, Y)}
\]

we obtain as answer the constraint \(5Y + 2X \leq 39, X \geq 5, X \geq 2\) which explicitly defines the set of points belonging to the triangle \(\text{tr1}\). Finally, if we ask the system

\[
\text{obj(ob1, SpE)}
\]

the answer is \(\text{SpE} = \text{tr1} \oplus \text{tr2}\), which expresses that the object \text{ob1} is the union of two triangles with identifiers \(\text{tr1}\) and \(\text{tr2}\).

### 6.3.2 Operators on spatial objects

In order to manipulate spatial objects we provide the ordinary set-theoretic operations: union \(\oplus\), intersection \(\odot\), difference \(\setminus\) and complement\(^*\). Through these operations we build expressions whose basic components are the triangle identifiers. Formally the set of spatial expressions we will use in our framework is defined as follows:

\[
\text{SpExp ::= TrId | } \emptyset | \text{SpExp} \oplus \text{SpExp} | \text{SpExp} \odot \text{SpExp} | \text{SpExp} \setminus \text{SpExp} | \text{SpExp}^-
\]
Figure 6.5: Our region of interest.

where \( TrId \) is the syntactic category of triangle identifiers and \( \emptyset \) denotes an empty area.

The meaning of such operations is defined by a set of clauses defining the predicate \( belong \) stating when a point belongs to a spatial expression.

\[
\begin{align*}
\text{\textit{belong}}(X,Y, \text{\textit{TrId}}) & \leftarrow \text{\textit{tri\_con}}(\text{\textit{TrId}}, X, Y) \\
\text{\textit{belong}}(X,Y, \text{SpExpr}_1 \oplus \text{SpExpr}_2) & \leftarrow \text{\textit{belong}}(X,Y, \text{SpExpr}_1) \\
\text{\textit{belong}}(X,Y, \text{SpExpr}_1 \oplus \text{SpExpr}_2) & \leftarrow \text{\textit{belong}}(X,Y, \text{SpExpr}_2) \\
\text{\textit{belong}}(X,Y, \text{SpExpr}_1 \ominus \text{SpExpr}_2) & \leftarrow \text{\textit{belong}}(X,Y, \text{SpExpr}_1), \\
& \text{\textit{belong}}(X,Y, \text{SpExpr}_2) \\
\text{\textit{belong}}(X,Y, \text{SpExpr}_1 \setminus \text{SpExpr}_2) & \leftarrow \text{\textit{belong}}(X,Y, \text{SpExpr}_1), \\
& \text{\textit{belong}}(X,Y, \text{SpExpr}_2) \\
\text{\textit{belong}}(X,Y, \text{SpExpr}^{-}) & \leftarrow \text{\textit{complement}}(X,Y, \text{SpExpr})
\end{align*}
\]
The first clause states that a point \((x, y)\) belongs to a triangle \(\text{triId}\) if it satisfies the constraint representing the triangle (i.e., if \(\text{tri\_con(triId, x, y)}\) is provable). The definitions of union, intersection and difference are straightforward since they exactly reflect their mathematical definition. The definition of the complement operation is based on the predicate \(\text{complement}\). To define it, first we provide the representation of the complement of a triangle and then, by exploiting De Morgan laws, we easily obtain a set of clauses for the predicate \(\text{complement}\). It is worth noting that there is no clause for \(\emptyset\) because no point belongs to it.

**Complement of a triangle**

The complement of a non-degenerate triangle is the union of three half-planes which are the complements of the half-planes determined by the predicate \(\text{findSide}\). First we define the predicate \(\text{findOpp}\) which provides the constraint expressing such half-planes.

\[
\text{findOpp}(X, Y, X_1, Y_1, X_2, Y_2, X_3, Y_3) \leftarrow (Y_3 - Y_1)(X_2 - X_1) \geq (Y_2 - Y_1)(X_3 - X_1), \\
(Y_2 - Y_1)(X_2 - X_1) < (Y_2 - Y_1)(X - X_1).
\]

\[
\text{findOpp}(X, Y, X_1, Y_1, X_2, Y_2, X_3, Y_3) \leftarrow (Y_3 - Y_1)(X_2 - X_1) \leq (Y_2 - Y_1)(X_3 - X_1), \\
(Y_2 - Y_1)(X_2 - X_1) > (Y_2 - Y_1)(X - X_1).
\]

Notice that \(\text{findOpp}\) selects all the points in the half-plane opposite to the one containing \((X_3, Y_3)\), in contrast with \(\text{findSide}\) which selects all the points in the same half-plane of \((X_3, Y_3)\).

We define next the predicate \(\text{compl\_tri}\) which models the complement of a triangle. A point \((X, Y)\) belongs to the complement of a triangle if it belongs to one of the three half-planes determined by the predicate \(\text{findOpp}\).

\[
\text{compl\_tri}(\text{triId}, X, Y) \leftarrow \text{tri}(\approx \text{triId}, X_1, Y_1, X_2, Y_2, X_3, Y_3), \\
\text{distinct}(X_1, Y_1, X_2, Y_2, X_3, Y_3), \\
\text{findOpp}(X, Y, X_1, Y_1, X_2, Y_2, X_3, Y_3)
\]

\[
\text{compl\_tri}(\text{triId}, X, Y) \leftarrow \text{tri}(\approx \text{triId}, X_1, Y_1, X_2, Y_2, X_3, Y_3), \\
\text{distinct}(X_1, Y_1, X_2, Y_2, X_3, Y_3), \\
\text{findOpp}(X, Y, X_2, Y_2, X_3, Y_3, X_1, Y_1)
\]

\[
\text{compl\_tri}(\text{triId}, X, Y) \leftarrow \text{tri}(\approx \text{triId}, X_1, Y_1, X_2, Y_2, X_3, Y_3), \\
\text{distinct}(X_1, Y_1, X_2, Y_2, X_3, Y_3), \\
\text{findOpp}(X, Y, X_3, Y_3, X_1, Y_1, X_2, Y_2)
\]

We finally give the definition of the predicate \(\text{complement}\) in terms of the predicates \(\text{compl\_tri, belong}\) and by using the De Morgan laws.
\[ \text{complement}(X, Y, \text{TrId}) \leftarrow \text{compl}\_\text{tri}(\text{TrId}, X, Y) \]

\[ \text{complement}(X, Y, \text{SpExp}_1 \oplus \text{SpExp}_2) \leftarrow \text{complement}(X, Y, \text{SpExp}_1), \]
\[ \quad \text{complement}(X, Y, \text{SpExp}_2) \]

\[ \text{complement}(X, Y, \text{SpExp}_1 \ominus \text{SpExp}_2) \leftarrow \text{complement}(X, Y, \text{SpExp}_1) \]
\[ \quad \text{complement}(X, Y, \text{SpExp}_2) \]

\[ \text{complement}(X, Y, \text{SpExp}_1 \setminus \text{SpExp}_2) \leftarrow \text{complement}(X, Y, \text{SpExp}_1) \]
\[ \quad \text{complement}(X, Y, \text{SpExp}_2) \]

\[ \text{complement}(X, Y, \text{SpExp}^-) \leftarrow \text{belong}(X, Y, \text{SpExp}) \]

### 6.3.3 Creation of GIS layers

In GIS the objects of interest are often a set of a quite large number of disjoint areas characterized by a common property, such as zones with water or with a particular kind of tree or ground etc. Such areas form what in GIS is called a layer. To allow a user of our system to obtain directly these kinds of information, we provide the system with a mechanism (layer) that, given a certain property, returns a spatial expression representing the corresponding layer.

To specify that an object, identified by \(\text{ObId}\), enjoys a property \(\text{Prop}\), we use a unit clause of the form

\[ \text{hasProperty}(\text{ObId}, \text{Prop}). \]

In order to collect the objects with the same property we define the clause

\[ \text{objWithProp}(\text{Prop}, \text{ListObId}) \leftarrow \text{set}\_\text{of}(\text{ObId}, \text{hasProperty}(\text{ObId}, \text{Prop}), \text{ListObId}) \]

where \(\text{set}\_\text{of}\) is the Prolog meta-predicate provided to work on sets. In this case, it is used to compute the list of distinct object identifiers which satisfy the goal \(\text{hasProperty}(\text{ObId}, \text{Prop})\), that is the list of identifiers of objects which enjoy property \(\text{Prop}\). Notice that \(\text{set}\_\text{of}\) is a predicate which allows one to answer questions at the second-order and it is not included in the traditional computational model of (constraint) logic programming. However, our use of the \(\text{set}\_\text{of}\) predicate can be simulated by computing, for each property \(\text{Prop}\), the list of \(\text{ObId}\) satisfying \(\text{Prop}\) during the translation phase from GIS to our logical representation. Hence this computation is made only once for all and no actual use of \(\text{set}\_\text{of}\) is then required further.

Finally, a layer is represented by a spatial expression, obtained by solving the predicate \(\text{layer}\).
6.3. Declarative GIS analysis in MuTACLP

\[
\begin{align*}
\text{layer} & \quad (\text{Prop}, \text{SpExp}) \leftarrow \text{objWithProp} (\text{Prop}, L), \\
& \text{extract} (L, \text{SpExp}) \\
\text{extract} & \quad ([], \emptyset).
\end{align*}
\]

\[
\begin{align*}
\text{extract} & \quad ([\text{ObId}], \text{SpExp}) \leftarrow \text{obj} (\text{ObId}, \text{SpExp}) \\
\text{extract} & \quad ([\text{ObId} \mid L], \text{SpExp} \oplus \text{SpE}) \leftarrow \text{obj} (\text{ObId}, \text{SpE}), \\
& \text{extract} (L, \text{SpExp})
\end{align*}
\]

A layer of the property Prop is denoted by a spatial expression which is the union of the spatial expressions associated with the objects satisfying Prop. If no object enjoys the property then the empty area, \(\emptyset\), is returned.

6.3.4 A module for handling spatial data

We can exploit our multi-theory setting to organize knowledge in distinct programs, in order to favor their reuse in other applications. In particular, we observe that many definitions, given in the previous subsections, are not domain dependent: only \(n_{\text{tri}}, \text{tri}\) and \(\text{hasProperty}\) are used to represent specific spatial data. Thus we can collect most definitions, provided to handle spatial objects, into a program, called \text{SPACE}Module, which will be combined with the specific theories in all the spatial analyses. We next list the predicates contained in such a module, omitting the definitions that can be found in Section 6.3 and in its subsections.

\text{SPACE}Module:

\% predicates used to find the constraint representing a triangle

\[
\begin{align*}
\text{findSide} & \quad (X, Y, X_1, Y_1, X_2, Y_2, X_3, Y_3) \leftarrow \ldots \\
\text{distinct} & \quad (X_1, Y_1, X_2, Y_2, X_3, Y_3) \leftarrow \ldots \\
\text{non-equal} & \quad (X_1, Y_1, X_2, Y_2) \leftarrow \ldots \\
\text{tri_con} & \quad (\text{TrId}, X, Y) \leftarrow \ldots
\end{align*}
\]

\% predicates used to model an object

\[
\begin{align*}
\text{obj} & \quad (\text{ObjId}, \text{SpExp}) \leftarrow \ldots \\
\text{join} & \quad (\text{ObjId}, J, \text{SpExp}) \leftarrow \ldots
\end{align*}
\]

\% predicate used to obtain the set of points in a spatial expression

\[
\begin{align*}
\text{belong} & \quad (X, Y, \text{SpExp}) \leftarrow \ldots
\end{align*}
\]

\% predicates used to find the constraint representing the complement of a triangle

\[
\begin{align*}
\text{findOpp} & \quad (X, Y, X_1, Y_1, X_2, Y_2, X_3, Y_3) \leftarrow \ldots \\
\text{compLtri} & \quad (\text{TrId}, X, Y) \leftarrow \ldots
\end{align*}
\]
\% predicate used to obtain the set of points in the complement of a
\% spatial expression
\textit{complement}(X, Y, SpExp) \leftarrow \ldots

\% predicates used to create a layer
\textit{layer}(\textit{Prop}, \textit{SpExp}) \leftarrow \ldots
\textit{extract}((\textit{ListObId}, \textit{SpExp}) \leftarrow \ldots

\% some auxiliary predicates
\textit{nonEmpty}(\textit{SpExp}) \leftarrow \textit{belong}(X, Y, \textit{SpExp})

We can also extend this module in order to cope with the case of degenerate triangles, i.e., points and segments.

**Degenerate triangles: points and segments**

We recall that a point is represented by a triangle having three identical vertexes, and a segment by a triangle with the first two vertexes distinct (the end-points of the segment) and the third one equal to the second one.

If the triangle is degenerate and it is a point then the constraint is immediate.

\textit{tri\_con}(\textit{TrId}, X, Y) \leftarrow \textit{tri}(\rightarrow \textit{TrId}, X_1, Y_1, X_1, Y_1, X_1, Y_1),
X = X_1, Y = Y_1

In the case of a segment we obtain the desired constraint by using the equation of the line crossing the end-points of the segment. In detail, first we define a predicate \textit{between} which returns the constraint representing the points between two given points.

\begin{align*}
\textit{between}(X, X_1, X_2) & \leftarrow X_1 > X_2, X_1 \geq X, X \geq X_2 \\
\textit{between}(X, X_1, X_2) & \leftarrow X_1 < X_2, X_1 \leq X, X \leq X_2
\end{align*}

Then we define the predicate \textit{findPointsIn} which returns the constraint representing the points that belong to a segment.

\begin{align*}
\textit{findPointsIn}(X, Y, X_1, Y_1, X_2, Y_2) & \leftarrow (Y - Y_1)(X_2 - X_1) = (Y_2 - Y_1)(X - X_1), \\
& \quad X_1 \neq X_2, \\
& \quad \textit{between}(X, X_1, X_2) \\
\textit{findPointsIn}(X, Y, X_1, Y_1, X_2, Y_2) & \leftarrow X_1 = X_2, X = X_1, \\
& \quad \textit{between}(Y, Y_1, Y_2)
\end{align*}

Finally, recalling that in the case of a segment, the end-points are the first two vertexes in the triangle predicate \textit{tri}, the definition of the predicate \textit{tri\_con} is
\[ tri_{\con}(TrId, X, Y) \leftarrow tri(\neg TrId, X_1, X_2, Y_1, Y_2),
\]
\[ \text{non-equal}(X_1, X_2, Y_2),
\]
\[ \text{findPointsIn}(X, Y, X_1, X_2, Y_2)
\]

**Complement of a degenerate triangle**

A similar strategy has to be followed to define the complement of a degenerate triangle. If the triangle is degenerate and it is a point then its complement is straightforward.

\[ \text{compl}_\tri(TrId, X, Y) \leftarrow tri(\neg TrId, X_1, X_1, X_1, X_1, X_1),
\]
\[ X \neq X_1, Y \neq Y_1
\]

Otherwise if it is a segment, its complement is defined as follows. First we define a predicate \( \text{nonBetween} \) which returns a constraint representing the points that are not between two points.

\[ \text{nonBetween}(X, X_1, X_2) \leftarrow X_1 > X_2, X_1 < X
\]
\[ \text{nonBetween}(X, X_1, X_2) \leftarrow X_1 > X_2, X < X_2
\]
\[ \text{nonBetween}(X, X_1, X_2) \leftarrow X_1 < X_2, X_1 > X
\]
\[ \text{nonBetween}(X, X_1, X_2) \leftarrow X_1 < X_2, X > X_2
\]

Then we define the predicate \( \text{findPointsOut} \) which returns the constraint representing the points which do not belong to a segment.

\[ \text{findPointsOut}(X, Y, X_1, Y_1, X_2, Y_2) \leftarrow (Y - Y_1)(X_2 - X_1) > (Y_2 - Y_1)(X - X_1)
\]
\[ \text{findPointsOut}(X, Y, X_1, Y_1, X_2, Y_2) \leftarrow (Y - Y_1)(X_2 - X_1) < (Y_2 - Y_1)(X - X_1)
\]
\[ \text{findPointsOut}(X, Y, X_1, Y_1, X_2, Y_2) \leftarrow (Y - Y_1)(X_2 - X_1) = (Y_2 - Y_1)(X - X_1),
\]
\[ X_1 \neq X_2,
\]
\[ \text{findPointsOut}(X, Y, X_1, Y_1, X_2, Y_2) \leftarrow \text{nonBetween}(X, X_1, X_2)
\]
\[ \text{findPointsOut}(X, Y, X_1, Y_1, X_2, Y_2) \leftarrow X_1 = X_2, X = X_1,
\]
\[ \text{findPointsOut}(X, Y, X_1, Y_1, X_2, Y_2) \leftarrow \text{nonBetween}(Y, Y_1, Y_2)
\]

Now we can give the definition of the predicate \( \text{compl}_\tri \) in the case \( TrId \) is a segment.

\[ \text{compl}_\tri(TrId, X, Y) \leftarrow tri(\neg TrId, X_1, X_2, Y_1, X_2, Y_2),
\]
\[ \text{non-equal}(X_1, X_2, Y_2),
\]
\[ \text{findPointsOut}(X, Y, X_1, Y_1, X_2, Y_2)
\]
Temporal views

We conclude this section by observing that the operation $\downarrow$, defined in Section 5.1.3 for programs with annotated clauses, can be easily redefined into the MuTACL framework. This operation acts as a valid-timeslice operator in the field of databases and it will be useful in the next examples to create temporal views.

**Definition 11** Let $P$ be a TACL program and $I$ be a ground interval.

$$P \downarrow I = P \cap 1^I_p,$$

where $1^I_p$ is a program defined as follows:

for all $p$ defined in $P$ with arity $n$

$$p(X_1, \ldots, X_n) \text{ th } I.$$

Notice that this program is exactly the translation of the one in Definition 3 according to the rules established in Section 5.5.

### 6.4 Some examples

In this section we present some applications that highlight how analysis criteria can be naturally described in MuTACL. In particular, we illustrate a way to combine spatial and temporal information and how the features offered by constraint logic programming improves the ability of reasoning.

**Favorable Habitat**

The first application problem we address consists in the analysis of a geographic area, which can be formulated as follows.

Find all the zones in a given region that provide a favorable habitat for hares. These animals live in woods, near sources of water, and they eat vegetables, such as lettuce, wild cabbage, turnip. The main predators of hares are foxes, wolves and raptorial birds like eagles. Thus a favorable habitat will be an area rich of water where it is easy to find vegetables and possibly without predators.

The above description is very general and it refers to the usual behavior of a hare during the year. We can describe the favorable habitat for a hare as follows

**Hares-Habitat:**

$$\text{habitat(hares, } \text{SpE}_1 \otimes \text{SpE}_2) \text{ th } [\text{jan, dec}] \leftarrow \text{layer(water, } \text{SpE}_1),$$

$$\text{layer(vegetable, } \text{SpE}_2)$$

$$\text{predator(hares, } (\text{SpE}_1 \oplus \text{SpE}_2) \oplus \text{SpE}_3) \text{ th } [\text{jan, dec}] \leftarrow \text{layer(fox, } \text{SpE}_1),$$

$$\text{layer(wolf, } \text{SpE}_2),$$

$$\text{layer(eagle, } \text{SpE}_3)$$
Figure 6.6: Favorable habitat

\[
\text{favorableArea}(\text{hares, } X, Y) \leftarrow \text{habitat}(\text{hares, } \text{SpE}_1) \leftarrow \text{th}[T_1, T_2], \\
\text{predator}(\text{hares, } \text{SpE}_2) \leftarrow \text{th}[T_1, T_2], \\
\text{belong}(X, Y, \text{SpE}_1 \setminus \text{SpE}_2)
\]

The clause for \text{habitat} states that a zone where hares can live is the intersection of the layer of water and of the layer of vegetables and this holds throughout the year. Since foxes, wolves and eagles are predators of the hare during the entire year we annotate the head of the clause defining \text{predator} with \text{th}[\text{jan, dec}]. Now we can compute a favorable habitat for hares in a certain period \([T_1, T_2]\) by removing from the habitat areas in \([T_1, T_2]\), pieces of land where the predators of hares can be found during that time period.

In Figure 6.6 we depict where the water is, where vegetables grow and where foxes, wolves and eagles live in the region described in Figure 6.5. The program \text{REGION} gives the logical description of the region. It consists of the unit clauses defined in Example 18 together with the property associated with each object.
Region:
\[ n_{\text{tri}}(ob1, 2). \quad \text{tri}(ob1, 0, \text{tr}1, 2, 7, 2, 5, 7, 5). \quad \text{tri}(ob1, 1, \text{tr}2, 2, 7, 7, 5, 7, 7). \]
\[ \vdots \]
\[ n_{\text{tri}}(ob6, 2). \quad \text{tri}(ob6, 0, \text{tr}11, 1, -3, 3, -6, 3, -3). \quad \text{tri}(ob6, 1, \text{tr}12, 1, -3, 1, -6, 3, -6). \]
\[ \text{hasProperty}(ob1, \text{eagle}). \]
\[ \text{hasProperty}(ob3, \text{water}). \]
\[ \text{hasProperty}(ob5, \text{water}). \]
\[ \text{hasProperty}(ob2, \text{vegetable}). \]
\[ \text{hasProperty}(ob4, \text{fox}). \]
\[ \text{hasProperty}(ob6, \text{wolf}). \]

In our region there are two areas rich of water whereas the layers for vegetables and for the different kinds of predators consist of a single area. To compute the water layer we ask the system

\[ \text{demo}(\text{Region} \cup \text{SpaceModule}, \text{layer(water, SpE)}) \]

and the answer is \( \text{SpE} = ob3 \oplus ob5 \). Now if we want to know where is the favorable habitat in March, we ask the following query

\[ \text{demo}(\text{Hares-Habitat} \cup (\text{Region} \cup \text{SpaceModule}), \text{favorableArea(hares, X, Y) th [mar, mar]}) \]

The grey area is the favorable habitat and the system describes it by returning two solutions to the previous query:

\[ X \leq 4, Y \leq 4, X + \frac{2}{5} * Y \geq \frac{22}{5} \]
\[ X \leq 4, Y \leq 4, X + \frac{2}{5} * Y \leq \frac{22}{5}, Y > 3, X \geq 3 \]

Indeed we can improve the analysis on the behavior of the hare taking into account how it varies during the year.

During Spring hares like eating corn and in Autumn they feed on also with oilseeds rich in moisture.

As a consequence their favorable areas in the cited seasons are more specific: the areas where hares prefer staying should contain respectively corn or oilseeds, too.

To reflect this more complex behavior of hares we can restrict the program Hares-Habitat by using the intersection operation. We define two programs Spring and Autumn which refine the definition of the favorableArea predicate.

Spring:
\[ \text{habitat(hares, SpE) th [apr, jun]}. \]
\[ \text{predator(hares, SpE) th [apr, jun]}. \]
\[ \text{favorableArea(hares, X, Y) th [apr, jun] \leftarrow layer(corn, SpE)}, \]
\[ \text{belong}(X, Y, \text{SpE}) \]
Autumn:
\[ \text{habitat}(hares, SpE) \, \text{th} [\text{oct}, \text{dec}] \]
\[ \text{predator}(hares, SpE) \, \text{th} [\text{oct}, \text{dec}] \]
\[ \text{favorableArea}(hares, X, Y) \, \text{th} [\text{oct}, \text{dec}] \leftarrow \text{layer}(oil-seed, SpE), \]
\[ \text{belong}(X, Y, SpE) \]

Spring and Autumn restrict the temporal validity of the definition of habitat and predator. Thus, when these programs are intersected with a program which contains clauses defining such predicates, those rules holding from April to June or from October to December, respectively, are selected. Moreover, Spring and Autumn add a further constraint either on the temporal validity of favorableArea or on how a favorable zone is computed, i.e., from April to June (resp. from October to December) also corn (resp. oil-seeds) is required to grow there.

The program expression that captures the season-dependent knowledge is

\[
(Hares-\text{Habitat} \cap (\text{Spring} \cup \text{Autumn})) \cup \\
(Hares-\text{Habitat}\downarrow [\text{jan}, \text{mar}] \cup Hares-\text{Habitat}\downarrow [\text{mid}, \text{sep}])
\]

where the operator \( \downarrow \) allows us to restrict the temporal validity of the general set of clauses in Hares-\text{Habitat}.

Network analysis

We want to describe towns and roads of a region and inquire the system about the connections among them. This is a typical example of network analysis, one of the cornerstones of GIS functionality, that may find applications in many areas [117]. To do this kind of analysis inside our framework, we exploit the deductive power and the possibility of defining recursive predicates of constraint logic programming.

The description of towns and roads is obtained through the translation presented in the previous section specifying town and road respectively as properties of such objects (e.g., hasProperty(\( r_1 \), road) is generated if \( r_1 \) is the object identifier of a road. See Section 6.3.3). Now we introduce several predicates to support network analysis.

First of all we give the definition of a general predicate path, that given the identifiers of two objects, \( ObId_1 \) and \( ObId_2 \), and a list of properties, \( LProp \), returns a possible way to reach \( ObId_2 \) from \( ObId_1 \) crossing areas satisfying properties in \( LProp \).

\[
\text{path}(ObId_1, ObId_2, LProp, Acc, [ObId_2|Acc]) \leftarrow \text{obj}(ObId_1, SpE_1), \\
\text{obj}(ObId_2, SpE_2), \\
\text{nonEmpty}(SpE_1 \otimes SpE_2)
\]
path(ObId₁, ObId₂, LProp, Acc, LObj) ←
  hasProperty(ObId, Prop),
  member(Prop, LProp),
  nonMember(ObId, Acc),
  ObId ≠ ObId₂,
  obj(ObId, SpE),
  obj(ObId₁, SpE₁),
  nonEmpty(SpE₁ ⊗ SpE)
  path(ObId, ObId₂, LProp, [ObId|Acc], LObj)

where the predicates member and nonMember check whether an object does or does not belong to a list, respectively, and they are defined as usual. The fourth argument of path is an accumulator in which we collect the objects we have already selected during the computation, and the fifth argument is the list, in the inverse order, of the objects crossed to reach ObId₂ from ObId₁.

The meaning of the two clauses is straightforward: the first one states that if the two objects intersects then our search for the path is finished. Otherwise, we look for an object ObId which has one of the properties in LProp, it has not been selected yet, it is not ObId₂ and it intersects the object ObId₁. Finally, we make a recursive call to find a path between ObId and ObId₂.

To start the search and to return the list of the visited objects from ObId₁ up to ObId₂ we define the predicate linked as follows

linked(ObId₁, ObId₂, LProp, [ObId₁]) ← ObId₁ = ObId₂
linked(ObId₁, ObId₂, LProp, LObj) ← ObId₁ ≠ ObId₂,
  path(ObId₁, ObId₂, LProp, [ObId₁], Lrev),
  reverse(Lrev, LObj)

where reverse inverts the elements of its first argument which is a list.

At this point we can easily find a route to go from a town to another, if this route exists, by using the roads in GIS.

route(Town₁, Town₂, ListRoads) ← linked(Town₁, Town₂, [road], ListRoads)

The definition of the predicate route consists in specifying road as property for the objects we want to use to build a path from Town₁ to Town₂.

The main advantage of this approach is at the specification level: we declaratively state what is a route without having a fixed network. In other words, the only information we employ is the area of the represented objects, we do not use pre-defined nodes and links between them to move from one position to another. This leaves a larger freedom to specify conditions that the route has to satisfy and it is a general approach which is not application-dependent. Moreover, also the particular properties can be expressed without setting strange parameters as GIS functions
usually impose. In the predicate *linked* we have to set only a list of properties which correspond to the features GIS objects can enjoy.

On the other hand, the disadvantage concerns performance. In GIS sophisticated techniques like the *traveling salesperson algorithm* are used to generate the optimal routes whereas in the actual implementation within Sicstus Prolog with CLP(\(\mathbb{R}\)) [101] we only exploit backtracking provided by constraint logic programming. Thus solving a query of the kind \(\text{route}(\text{town1}, \text{town2}, L)\) is expensive in terms of time and the returned route is not in general optimal. However, our interest in this thesis is on proving the suitability of MuTACL to supply a GIS user with a set of facilities improving his/her capability of expressing analysis criteria.

It is worth noticing that all the constraints occurring in the *SpaceModule* are linear. This is because even though in the definition of some predicates, like *findSide*, quadratic constraints occur, during the computation the variables \(X_1, Y_1, X_2, Y_2, X_3, Y_3\) are replaced by constants. If the vertexes of the triangles are rational numbers then we handle linear constraints over rational numbers, a great advantage because many algorithms have been developed to efficiently implement a wide variety of operations on sets defined by this kind of constraints.

### 6.5 Conclusion

In this chapter we have shown that MuTACL to allow for the representation of spatial data, for the definition of facilities to tackle the complexity of these data, like the classical set-theoretic operations (union, intersection, etc.) between spatial objects, and it offers the possibility to establish spatio-temporal correlations, e.g., time-varying areas. Moreover, MuTACL has been used on top of GIS in order to provide users with a more friendly interface for GIS analysis. Even in this application, having a multi-theory setting has been proved to be very advantageous, because often knowledge used in GIS analysis is fragmented into different sources. For instance, one can get environmental restrictions from the local municipality, the general laws from the government, and the *best place* criteria from the planner, and, by employing the program composition operations, we can express complex queries on a combination of such analysis criteria.

Now we highlight the peculiar features of MuTACL comparing it with the proposals for handling spatio-temporal data. The first general remark on the state of the art in this field is that while the temporal [34, 112, 103] and spatial [59, 92] database technologies are relatively mature, their combination is far from straightforward. In this context, the constraint database approach [69] appears to be very promising. Constraint databases provide a uniform framework for modeling arbitrarily high dimensional data, they are thus naturally suited for temporal, spatial, and spatio-temporal applications.

Our spatio-temporal language is close to the approaches based on constraint databases [17, 57, 37, 36, 38]. As we have already pointed out it shares with them
the fundamental idea of using constraints to provide an explicit representation of the 
set of (possibly infinite) points belonging to a spatial object. [17, 57] are extensions 
of languages born to express only spatial data. Thus they offer more high-level 
mechanisms to query spatial data than temporal information. In fact, they can 
cancel only definite temporal information and there is no support for periodic, in-
definite temporal data as we stressed in the comparison with [57] in Section 6.1. 
In the contrary MutaCLP provides more facilities to reason on temporal data and 
to establish spatio-temporal relationships. For instance, we can describe continuous 
change in time as well as [37, 36, 38] do, whereas both [17] and [57] can represent 
only discrete change.
In [37, 38] the main goal is to prove that constraint databases are suitable as inter-
mediate level facilitating the interoperability of spatio-temporal data models. Thus 
it is shown how various temporal and spatial data models in literature, such as 2-
spaghetti [79], Worboys’s spatio-temporal model [116], TQuel Data Model [104], can 
be translated into constraint databases: the focus is on the representation of spatio-
temporal data, and no query language dealing with the complexity of such kind of 
information is given. Still in the spirit of providing an adequate model for the rep-
resentation of spatio-temporal data, in [37, 38] a new spatio-temporal model, called 
Parametric 2-Spaghetti Data Model, is proposed. This model is an extension of the 
2-Spaghetti Model where relations are parametric with respect to time (See Sec-
tion 3.3 for more details). The translation of this parametric model into constraint 
databases consists of constraints on the spatial dimension X, Y, and on the temporal 
dimension T with the possibility of interrelating them. This constraint-based rep-
resentation coincides with the way we establish spatio-temporal correlations, and 
it allows one to support continuous change. This last feature is exploited in [36] 
to animate spatio-temporal objects. However, even for the Parametric 2-Spaghetti 
data model no query language is presented.
A very recent paper [73] states the suitability of the scheme of indefinite con-
straint databases proposed by Koubarakis [71, 72] to represent indefinite information 
arising in temporal, spatial and spatio-temporal applications. This approach was 
originally developed for the handling of temporal information and it was one of 
the most interesting in the field of temporal databases. In [73] the framework has 
been extended by the possibility of defining rules on relations. Such rules look like 
Datalog clauses with the extra feature that the body can contain constraints, in 
practice they are CLP Horn clauses without function terms. The authors declare 
that for the purposes of the paper they are not recursive, thus their introduction 
is done simply for convenience so that new relations can be defined from existing 
one [73]. This is the only support added to handle spatial data and in fact it is 
used to define a moving object, that is a point whose coordinates are linear func-
tions of time, in the style of how we model time-varying objects. Thus it seems that 
there are not very particular high-level mechanisms specific for spatial data, but, 
as we do for MutaCLP, Koubarakis and Skiadopoulos model spatial data simply 
by using constraints. In our opinion this is a preliminary approach with respect to
the handling of the spatial dimension, in fact, as stated by the authors, they are looking for more “graceful” representations of spatial regions. Actually, the goal of the paper is the search for tractable classes of constraints in the area of temporal and spatial constraints.

To conclude, we have to improve the support for spatial data even if, as shown in the examples presented throughout the chapter, we are already able to perform interesting spatial and spatio-temporal analyses thanks also to the deductive power supplied by constraint logic programming. By defining recursive predicates we can compute the transitive closure of relations, an ability not provided by the traditional approaches in the database field. This ability is very important and in Section 6.4 we illustrated one of its applications in the network analysis, where it is used to search for connections between objects. More generally, we do not represent only data as in constraint databases but we can express also rules and this extra feature makes the difference if we want to use the language as specification and/or analysis language.

An interesting direction of future research regards the investigation of some important spatial properties, such as metric properties and topological relations between objects, which, at the moment, are not considered in our framework. In the spirit of Egenhofer’s 9-intersection model [40], we can think of extending our spatial module in order to include the definition of the boundary and of the interior of an object. Then we can find the topological relations existing between two convex objects by considering the intersections between their boundaries, their interiors and their complements. On the other hand, following Grumbach et al. [56] we could add to the language a boolean operation involving the distance between two regions in \( \mathbb{Q} \) defined as follows: \( \text{Dist} \mid_{d, \Theta, \alpha}(R, S) = \text{true} \) if \( d(p_1, p_2) \Theta \alpha \) holds for some \( p_1 \) in \( R \) and some \( p_2 \) in \( S \), and \( \text{false} \) otherwise, where \( d \) is a distance, \( \Theta \) is a predicate, e.g., \( =, \leq \) etc., and \( \alpha \in \mathbb{Q} \). Grumbach et al. points out that adding this operation preserves linearity and feasibility.
Chapter 7

Towards interoperability

A common problem facing many organizations today consists in having multiple, disparate information sources and repositories, including databases, GIS, knowledge bases, etc. In general it is difficult to get and fuse information from multiple sources, and thus there is the strong need for systems and tools enabling interoperability among heterogeneous sources. Making information systems work together entails addressing heterogeneity issues, which arise in many forms, ranging from the hardware/software platforms, to the data model and schema used to provide logical structure for the stored data, to the very kinds of data and information that are being stored (e.g., records, text, video).

In this chapter we focus on the semantic interoperability which addresses problems deriving from the use of different data models, schemas which do not match, from different interpretations of data, such as naming and format conflicts, scaling and unit conflicts and so on. This is a hot research area [115, 110, 33, 111, 63, 19] and various architectures for supporting database semantic interoperation can be found in literature. In [63] Hull distinguishes:

- **Integrated read-only views**: Mediation supports an integrated, read-only, view of data that resides in multiple databases.

- **Multiple databases sharing information**: Federation provides a framework in which several databases can join a federation. Each member of a federation extends its schema to incorporate a subset of the data held in other member databases.

- **Integrated read-write views**: Mediation with update is a natural extension of the mediation architecture to support updates. This gives rise to the view update problem: the effect of a view update on the underlying source database is typically ambiguous.

- **Coordinating multiple databases**: Workflow deals with managing separate data repositories in large organizations to help controlling the movement of goods,
services and financial instruments. Much of the research on supporting the workflow perspective in connection with heterogeneous databases has focused on models for concurrency control, and approaches to maintaining inter-database constraints.

The chapter is organized as follows. Section 7.1 defines the language STILan, which is obtained by enriching MuTACL with a message passing mechanism. Moreover, it shows how STILan offers the features required to be a mediator language for the semantic integration of different data sources, and in particular for the semantic interoperability of spatio-temporal knowledge bases. A concrete example that clarifies how STILan supports semantic interoperability is presented in Section 7.2. Finally, Section 7.3 contains some concluding remarks.

### 7.1 STILan: Space, Time and Interoperability

To attain interoperability among different information sources we adopt a mediator architecture with a wrapping layer, that is common in many projects (e.g., [33, 97]). It is depicted in Figure 7.1. Before introducing the language we are going to write mediators with, it is worth recalling what mediators are.

#### 7.1.1 Mediators

In [115] Wiederhold defines a mediator as a software module that exploits encoded knowledge about certain sets or subsets of data to create information for a higher layer of applications. Moreover, he couples this definition with the “three layers architecture” (Figure 7.2) where the mediators belong to a distinct middle layer,
thus distinguishing the function of mediation from the user-oriented processing and from access to data sources.

The functions of the mediators are the routing of requests to appropriate resources, the combination of the data received and their summarization for customers. It is evident that mediators often have to merge data from multiple sources and thus they must be able to compensate for mismatches in terms of name, scope, granularity of abstraction, temporal units, and domain definitions, to list only some examples. Therefore a mediator contains knowledge to be used to control the merging process, and it can also support abstraction and generalization over the underlying data in order to present data in an adequate format to customers.

The particular kind of mediator architecture, which will be adopted in the following, is represented in Figure 7.1. Observe that above each information source there is a wrapper which logically converts the underlying data to a common model and provides a common query language for extracting information. Wrapping does not affect the local models of the knowledge bases. It just builds a layer of information above it.

The advantages of this approach to interoperability are twofold: the mediator layer makes the user applications independent of the data resources, and at the same time, the different sources of data preserve their autonomy and they do not have to change their structures to include information only necessary for the integration.

### 7.1.2 Which language?

The ability to select either the source where information is extracted from or the system in which a query is evaluated is fundamental to support semantic integration. Therefore, following [13, 97], we enrich MuTACL by adding a construct \texttt{wrt} which allows one to express message passing.

We call \textit{STILan} the language which supports the composition operations $\cup$ and $\cap$, and message passing among program expressions. Now a plain program is defined as a finite set of \textit{extended} TACL clauses, univocally identified by a program name (belonging to \textit{Pname}). Roughly speaking, an extended TACL clause, for each atom
in its body, may specify in which program expression it must be solved. Asking the
solution of a goal $G$ in a program expression $Q$ can be seen as the request of a
service to $Q$ or as a message sent to $Q$. More precisely, an extended TACL code
is a clause of the form

$$A \alpha \leftarrow C_1, \ldots, C_k, B_1, \ldots, B_n$$

where:

- $A$ is an atom, and $\alpha$ is an optional temporal annotation,
- $C_1, \ldots, C_k$ are constraints,
- each $B_i$ is
  - either an atom, possibly annotated,
  - or a formula of the form $(G \boxempty \mathcal{E})$, where $G$ is an atom, possibly anno-
    tated, and $\mathcal{E}$ is a program expression.

The idea is that $A \alpha \boxempty Q$ means that $A \alpha$ has to be solved in $Q$. This intended
behavior is captured at the meta-level by extending the MuTACL code meta-interpreter,
deined in Section 5.3.2, by the following clause:

$$\text{demo}(\mathcal{E}, A \alpha \boxempty Q) \leftarrow \text{demo}(Q, A \alpha)$$

Thus as anticipated, message passing is modeled as a change of context.

**Example 19** The construct $\boxempty$ is very useful to define sophisticated views. In
particular we show its usage in the context of GIS analysis.

In Section 6.4 we consider a region of interest and we model it through the pro-
gram REGION. This area remains unchanged during the whole year. Actually, the
distribution of water, predators and vegetables vary along the year. Thus it is reason-
able to have at least four regions which correspond to the situation in Winter, Spring,
Summer and Autumn. We can suppose that these are four images of GIS data in
such periods of the year. We call these programs RegWin, RegSpr, RegSum and
RegAut, respectively.

Now we can define a predicate sLayer which allows one to have different GIS
layers according to seasons.

$$\begin{align*}
  sLayer(Prop, SpE) & \text{th } [\text{jan, mar}] \leftarrow \text{layer}(Prop, SpE) \boxempty \text{RegWin} \cup \text{SpaceModule} \\
  sLayer(Prop, SpE) & \text{th } [\text{apr, jun}] \leftarrow \text{layer}(Prop, SpE) \boxempty \text{RegSpr} \cup \text{SpaceModule} \\
  sLayer(Prop, SpE) & \text{th } [\text{jul, sep}] \leftarrow \text{layer}(Prop, SpE) \boxempty \text{RegSum} \cup \text{SpaceModule} \\
  sLayer(Prop, SpE) & \text{th } [\text{oct, dec}] \leftarrow \text{layer}(Prop, SpE) \boxempty \text{RegAut} \cup \text{SpaceModule}
\end{align*}$$

Thus to find the areas where a certain property is still present along the year we
define the predicate always as follows:
always((\text{Prop}, \text{Sp}E_1 \otimes (\text{Sp}E_2 \otimes (\text{Sp}E_3 \otimes \text{Sp}E_4)))) \text{th}[\text{jan}, \text{dec}] \gets

\text{sLayer}(\text{Prop}, \text{Sp}E_1)\text{th}[\text{jan}, \text{mar}],

\text{sLayer}(\text{Prop}, \text{Sp}E_2)\text{th}[\text{apr}, \text{jun}],

\text{sLayer}(\text{Prop}, \text{Sp}E_3)\text{th}[\text{jul}, \text{sep}],

\text{sLayer}(\text{Prop}, \text{Sp}E_4)\text{th}[\text{oct}, \text{dec}]

The required area is obtained by intersecting the zones having the desired property in the different seasons.

Next we analyse which abilities are required from the wrapping and mediators layers of the architecture, and we suggest how STILan can support them.

In the wrapping layer the representation of data coming from the different sources of the underlying layer must be made uniform in order to obtain a common data model. We have already pointed out that constraints provide us with a very powerful means to attain a uniform representation of different kinds of data in a common model. In particular, we recall that Chomicki and Revesz [38] have shown that the use of constraints allows one to define an intermediate level facilitating the interoperability of spatio-temporal data models. The idea is that data in such models can be mapped into a uniform constraint representation where equality constraints over non-spatial and non-temporal variables express the thematic attributes, a conjunction of linear arithmetic constraints over spatial variables expresses the spatial component and linear arithmetic constraints over temporal variables express the temporal component.

Following the same idea, STILan can be used as language facilitating the interoperability among different spatio-temporal data models. In particular, the examples below show how two well-known spatial and temporal data models in literature can be translated into STILan programs.

\textbf{Example 20} In Section 6.3 we have given an automatic translation of spatial data in the 2-Spaghetti Model into a MuTACLAP program (thus into a STILan program). In a more abstract way, without using triangulation but providing directly the analytic representation of the spatial component of an object, the abstract semantics of the 2-spaghetti relation in Example 3 of Section 3.2 can be represented by the following STILan program.

\begin{verbatim}
object(p_1, X, Y) \gets X = 10, Y = 4.
oobject(p_1, X, Y) \gets X = 10, Y = 4.
object(l_1, X, Y) \gets 5 \leq X, X \leq 9, Y = -X + 15.
oobject(l_1, X, Y) \gets 5 \leq X, X \leq 9, Y = -X + 15.
object(t_1, X, Y) \gets 2 \leq X, X \leq 6, 3 \leq Y, Y \leq 11, Y \leq -X + 9
noobject(t_1, X, Y) \gets 2 \leq X, X \leq 6, 3 \leq Y, Y \leq 11, Y \leq -X + 9
object(r_1, X, Y) \gets 1 \leq X, X \leq 12, 2 \leq Y, Y \leq 11.
oobject(r_1, X, Y) \gets 1 \leq X, X \leq 12, 2 \leq Y, Y \leq 11.
object(p_2, X, Y) \gets 3 \leq X, 5 \leq Y, X - 1 \leq Y, Y \leq -X + 13, X + 5 \geq Y.
oobject(p_2, X, Y) \gets 3 \leq X, 5 \leq Y, X - 1 \leq Y, Y \leq -X + 13, X + 5 \geq Y.
\end{verbatim}

\textbf{Example 21} TQuel [104] is a model for representing temporal data. In TQuel each relation \( r \) contains two special attributes called From and To to represent valid time
and the values of these temporal attributes are integers denoting the end-points of an interval. We can represent the relation \( r \) into STILan by using, for each tuple of the relation, a clause of this kind

\[
r(a_1, \ldots, a_n) \text{th} [t_1, t_2]
\]

where \( a_1, \ldots, a_n \) are the values of the non-special attributes of the tuple and \( t_1, t_2 \) are the values of From and End attributes respectively.\(^1\)

The mediator layer has to provide the user with a declarative interaction with complex systems. In order to do this, mediators must be able to query the systems, to process the results and return the user an answer. As the various existing approaches suggest \([115, 111, 97, 12, 19]\), a declarative language is a good choice for mediators. Moreover, it is useful to have rules to express policies and also a deductive inference engine to allow additional information to be derived from the data stored into the different sources. Finally, mechanisms to send messages and also to combine knowledge belonging to different systems seem to be necessary. All these features are supplied by STILan.

### 7.2 An example: the management of a library

This example concerns the managing of a library that offers its services to people from various departments. It extends the one presented in \([97]\) to the treatment of temporal information. Since here the aim is to highlight the features concerning semantic integration, we simplify it a little, considering only three knowledge bases.

We model three deductive databases, representing information about the Department of Computer Science, IEI, which is a research institute, and a library, respectively. The whole architecture of the system we are going to describe is shown in Figure 7.3.

#### Source Databases

DEPCS:

\[
\begin{align*}
\text{student}(\text{gianni}, \text{rossi}) & \text{th} [\text{nov 1 1995}, \infty]. \\
\text{student}(\text{davide}, \text{verdi}) & \text{th} [\text{nov 1 1991}, \text{jun 15 1996}]. \\
\vdots \\
\text{phd}(\text{maria}, \text{gialli}, 3, \text{maria@cs}) & \text{th} [\text{nov 1 1989}, \text{oct 31 1993}]. \\
\text{researcher}(\text{giuseppe}, \text{bianchi}, \text{giuseppe@cs}) & \text{th} [\text{may 18 1987}, \text{nov 1 1997}]. \\
\text{professor}(\text{michele}, \text{neri}, \text{michele@cs}) & \text{th} [\text{nov 1 1980}, \infty]. \\
\vdots
\end{align*}
\]

\(^1\)Actually, in this example the whole time line is \([-\infty, \infty]\). However, as already said, our results hold also for this case.
7.2. An example: the management of a library

authorized\((FN, LN, 180)\)\(\text{th} [T_1, T_2] \leftarrow \text{professor}(FN, LN)\)\(\text{th} [T_1, T_2] \)
authorized\((FN, LN, 30)\)\(\text{th} [T_1, T_2] \leftarrow \text{researcher}(FN, LN)\)\(\text{th} [T_1, T_2] \)
authorized\((FN, LN, 30)\)\(\text{th} [T_1, T_2] \leftarrow \text{phd}(FN, LN)\)\(\text{th} [T_1, T_2] \)
authorized\((FN, LN, 7)\)\(\text{th} [T, T_2] \leftarrow \text{student}(FN, LN)\)\(\text{th} [T_1, T_2], (T - T_1) \geq 3 \)

IEI:
researcher\((tornas, rosi, torn@iei.cn)\)\(\text{th} [\text{jun} 10\,1985, \infty] \).
researcher\((luca, rossi, luca@iei.cn)\)\(\text{th} [\text{jan}\,15\,1974, \text{sep} 30\,1999] \).

\ldots

allow\_loan\((FN, LN, 180)\)\(\text{th} [T_1, T_2] \leftarrow \text{researcher}(FN, LN)\)\(\text{th} [T_1, T_2] \)

LIBRARY:
book\((stitlingShapiro, theArtofProlog, 1986)\)\(\text{th} [\text{jan}\,15\,1987, \text{sep}\,30\,1990] \)
book\((stitlingShapiro, theArtofProlog, 1986)\)\(\text{th} [\text{oct}\,30\,1990, \text{sep}\,30\,1992] \)
book\((ullman, knowledgeBaseSystems, 1988)\)\(\text{th} [\text{jun}\,2\,1997, \text{may}\,15\,1999] \)
journal\((jlp, 38, 2, 1998)\)\(\text{th} [\text{dec}\,29\,1998, \text{jan}\,27\,1999] \)
journal\((jlp, 38, 3, 1999)\)\(\text{th} [\text{jan}\,30\,1999, \text{apr}\,15\,1999] \)
proceedings\((vldb, 1995)\)\(\text{th} [\text{sep}\,30\,1995, \infty] \)
proceedings\((vldb, 1994)\)\(\text{th} [\text{nov}\,30\,1994, \infty] \).

\ldots

loanable\((Author, Title, nil, nil, Year)\)\(\text{th} [0, \infty] \leftarrow \)
book\((Author, Title, Year)\)\(\text{th} [\ldots] \)
loanable\((nil, Title, Vol, Num, Year)\)\(\text{th} [0, \infty] \leftarrow \)
journal\((Title, Vol, Num, Year)\)\(\text{th} [\ldots] \)

Figure 7.3: Architecture of the library mediator system.
The database DepCS contains information about students and PhD students in computer science, and researchers and professors who work at the Department. For each person we specify when he/she has such position. For instance, Davide Verdi was a student from November 1 1991 to July 15 1996, whereas Gianni Rossi has not yet finished his studies, he is a student since November 1, 1995. Moreover, such a database defines the predicate \textit{authorized} which establishes when and for how many days a person can borrow a book. Whatever position a person has he/she can borrow a book only during the period he/she studies or works at the Computer Science Department. On the other hand, the period of the loan varies according to the position. For instance a professor can borrow a book for three months whereas a student can get a book from the library for seven days only if he/she has been studying computer science for at least three years (see the constraint $T - T_1 \geq 3$ in the last clause defining \textit{authorized}).

The database IEI contains information about researchers of the institute and about the authorization for loan. Notice that here the predicate through which the authorization policy is expressed is \textit{allowLoan}.

Finally, the database Library contains information about books, journals, proceedings and about which of these publications are loanable. Each publication is associated with the period in which it is available. The \textit{loanable} predicate establishes the policy of the library: it is possible to loan only books and journals, and not proceedings. Thus proceedings are always available after their arrival in the library, this is expressed by a right-unbounded interval (i.e., $[\text{nov} 30 \ 1994, \infty]$).

\textbf{Wrapped Databases}

In the wrapper layer we make the representation of data uniform by choosing a common model. Thus to model a person we use the predicate \textit{person} and to represent an authorization for loan we choose the predicate \textit{authorized-for-loan} and to denote a book, a proceeding or a journal, we employ the predicate \textit{publication}. The translation into the common model is done in a “dynamic” way by means of the message passing mechanism. We denote with the prefix “W” the wrapped databases. In this phase we cope with several semantic heterogeneities. In the W-DepCS, W-IEI and W-Library databases we solve a problem of semantic discrepancy between the local schemata and the chosen wrapped schema.

\textbf{W-DepCS}:

\begin{verbatim}
person(FN, LN, student, cs) th [T1, T2] ← student(FN, LN) th [T1, T2] wrt DepCS
person(FN, LN, professor, cs) th [T1, T2] ←
  professor(FN, LN, Email) th [T1, T2] wrt DepCS
person(FN, LN, researcher, cs) th [T1, T2] ←
  researcher(FN, LN, Email) th [T1, T2] wrt DepCS
\end{verbatim}
7.2. An example: the management of a library

\[ \text{person} (FN, LN, \text{phd}, cs) \text{ th} [T_1, T_2] \leftarrow \text{phd} (FN, LN, \text{Email}) \text{ th} [T_1, T_2] \text{ wrt DepCS} \]

\[ \text{authorized for Joan} (FN, LN, \text{Period}) \text{ th} [T_1, T_2] \leftarrow \]  
\[ \text{authorized} (FN, LN, \text{Period}) \text{ th} [T_1, T_2] \text{ wrt DepCS} \]

W-IEI:

\[ \text{person} (FN, LN, \text{researcher}, \text{iei}) \text{ th} [T_1, T_2] \leftarrow \]  
\[ \text{researcher} (FN, LN, \text{Email}) \text{ th} [T_1, T_2] \text{ wrt IEl} \]

\[ \text{authorized for Joan} (FN, LN, \text{Period}) \text{ th} [T_1, T_2] \leftarrow \]  
\[ \text{allow Joan} (FN, LN, \text{Period}) \text{ th} [T_1, T_2] \text{ wrt IEl} \]

W-Library:

\[ \text{publication} (\text{book}, \text{Author}, \text{Title}, \text{nil}, \text{nil}, \text{Year}) \text{ th} [T_1, T_2] \leftarrow \]  
\[ \text{book} (\text{Author}, \text{Title}, \text{Year}) \text{ th} [T_1, T_2] \text{ wrt LIBRARY} \]

\[ \text{publication} (\text{journal}, \text{nil}, \text{Title}, \text{Vol}, \text{Num}, \text{Year}) \text{ th} [T_1, T_2] \leftarrow \]  
\[ \text{journal} (\text{Title}, \text{Vol}, \text{Num}, \text{Year}) \text{ th} [T_1, T_2] \text{ wrt LIBRARY} \]

\[ \text{publication} (\text{proceedings}, \text{nil}, \text{Title}, \text{nil}, \text{nil}, \text{Year}) \text{ th} [T_1, T_2] \leftarrow \]  
\[ \text{proceedings} (\text{Title}, \text{Year}) \text{ th} [T_1, T_2] \text{ wrt LIBRARY} \]

\[ \text{loanable} (\text{Author}, \text{Title}, \text{Vol}, \text{Num}, \text{Year}) \text{ th} [T_1, T_2] \leftarrow \]  
\[ \text{loanable} (\text{Author}, \text{Title}, \text{Vol}, \text{Num}, \text{Year}) \text{ th} [T_1, T_2] \text{ wrt LIBRARY} \]

Notice that the value \textit{nil} for an attribute in the common model is used when the local schema of a relation does not have such an attribute. For instance, a book has neither volume nor number, thus these fields have value \textit{nil} when a book is seen as a publication.

Mediators

MED-1:

\[ \text{loan} (FN, LN, \text{Author}, \text{Title}, \text{Vol}, \text{Num}, \text{Year}) \text{ th} [T, T + \text{Period}] \leftarrow \]  
\[ \text{loanable} (\text{Author}, \text{Title}, \text{Vol}, \text{Num}, \text{Year}) \text{ at} T \text{ wrt W-LIBRARY,} \]  
\[ \text{authorized for Joan} (FN, LN, \text{Period}) \text{ at} T \text{ wrt W-IEI} \cup \text{W-DEPCS} \]

Loan:

\[ \text{loan} (FN, LN, \text{Author}, \text{Title}, \text{Vol}, \text{Num}, \text{Year}) \text{ th} [T_1, T_2] \leftarrow \]  
\[ \text{publication} (X, \text{Author}, \text{Title}, \text{Vol}, \text{Num}, \text{Year}) \text{ th} [T_1, T_2] \text{ wrt W-LIBRARY} \]
MED-2:

\[
\text{loan \_to \_user}(FN, LN, Author, Title, Vol, Num, Year) \text{th}[T_1, T_2] \leftarrow \\
\text{loan}(FN, LN, Author, Title, Vol, Num, Year) \text{th}[T_1, T_2] \text{ wrt MED-1 \& LOAN}
\]

\[
\text{consult}(FN, LN, Author, Title, Vol, Num, Year) \text{at } T \leftarrow \\
\text{publication}(X, Author, Title, Vol, Num, Year) \text{at } T \text{ wrt W-LIBRARY}, \\
\text{person}(FN, LN, Position, Year, Dept) \text{at } T \text{ wrt W-IEI \& W-DePACs}
\]

MED-1 states the general loan regulation: a person can borrow a publication if he/she is authorized and the publication is loanable. Moreover, the person can keep the publication for a period varying according to his/her position and institution. LOAN contains a restricting rule which states that a loan is possible in a certain period if the publication is available in that period.

MED-2 interacts with the user and defines both the policy for loaning and for consulting a publication. The first rule refines the information obtained by MED-1 by using the further constraint in LOAN. It returns the user the period \([t_1, t_2]\) in which he/she can borrow the publication. Notice that the time period \([t_1, t_2]\) is the non-empty intersection between the period a person can borrow the publication (i.e., \(\text{loan}(\ldots) \text{th}[s_1, s_2]\) in MED-1) and the period in which the publication is available (i.e., \(\text{publication}(\ldots) \text{th}[r_1, r_2]\)). The second rule establishes that a person can consult a publication at a certain date \(T\) if the publication is available at \(T\) and the person is a member of IEI or he/she works or studies at the department of Computer Science.

### 7.3 Conclusion

In this chapter we have set up the bases to use STILan, i.e., MuTACL enriched with message passing, as a mediator language facilitating the interoperability of heterogeneous knowledge bases. Actually, STILan provides both a powerful tool for the representation of complex data, such as spatial, temporal and spatio-temporal data, and high-level mechanisms to combine information sources and to reason about spatio-temporal information.

There are many approaches in literature that use mediators to provide semantic integration among heterogeneous knowledge sources. Here we briefly describe HERMES [111] and Tsimmis [33].

The aim of the HERMES system [111] is to take the first steps towards the development of a principled methodology for integrating multiple data sources and reasoning systems, and to propose a mediator language within which access to the data sources and reasoning systems can be expressed uniformly. The language used for writing semantic integration code is logical and rule-based, and hence declarative. To extract information from different domains, the language uses a domain call. The idea is that each domain internally provides a set of operations through which
the functionalities of the domain are accessed. In principle this corresponds to our message passing mechanism, but indeed we employ it only to require the execution of a query inside logical theories whereas domain calls allow one to execute operations in spatial databases, relational databases etc. Actually, domain calls are similar to the way we have modeled the direct invocations to GIS functionalities.

A mediatory clause is of the form:

\[ A : [\mu_0, \tau_0] \leftarrow A_1 & \ldots & A_n \]

where each \( A_i \) is either an annotated atom or is a domain call. The kind of annotation supplied consists of a pair \([\mu, \tau]\) where \(\mu\) is an expression representing value between 0 and 1 inclusive and \(\tau\) is an expression representing a set of non-negative real numbers. These annotations are used to support reasoning capability over uncertainty and time.

The main differences with our approach are that no wrapping layer is needed because the interaction with the sources is obtained through the domain calls, and no high-level mechanisms, such as \(\cup\) and \(\cap\), are defined to combine knowledge. Our language is more expressive at a data specification level and it supports a more sophisticated reasoning on temporal information. We think that having different data represented in a common model favors their manipulation and allows one to establish correlations, though it is usually very expensive. It would be interesting to investigate on the possibility of adding domain calls to our language so that we can choose whether to invoke a call or to represent data at a logical level, a perspective already presented in the case of GIS analysis.

In the TSIMMIS project, Ullman et al. [33] define a mediator language based on an object-oriented extension of SQL. This language is provided with a “message passing” mechanism to refer to the wrapped databases. Furthermore, it has higher order features, that allows one to deal with schematic discrepancies in a compact way. The goal of the TSIMMIS Project is to develop tools that facilitate the rapid integration of heterogeneous information sources that may include both structured and unstructured data. TSIMMIS has components that extract properties from unstructured objects, translate information into a common object model (wrappers), combine information from several sources, allow browsing of information, and manage constraints across heterogeneous sites. In the architecture of TSIMMIS, the mediation layer works on a common model obtained by translating the various data sources by means of wrappers. Our proposed architecture is inspired by TSIMMIS, but our idea is that the common model is represented as Horn clauses and our language is an extension of constraint logic programming enriched with composition operations and message passing. We think that the key difference between such project and our proposal lies in the fact that our language has a clear formal semantics.

Moreover, we have already mentioned and described the work by Chomicki and Revesz [37, 38] on the interoperability of spatio-temporal databases where, however,
only data interoperability is supported. Another approach that addresses the problem of interoperability, in particular, among temporal databases is proposed in [19]. The paper presents an architecture of a multidatabase system where an appropriate formalization of the intended semantics is associated with each temporal relation and temporal database. This allows a temporal mediator, which is the central component of the system, to access the databases to retrieve implicit information in terms of time granularities different from those used to store data. The approach in this paper deserves a more careful investigation in order to try to introduce such facilities on time granularities in our language.

Finally, in [18], we addressed the problem of consistency maintainance among deductive databases. When working with multiple databases, able to perform updates, the problem of interoperation has also an aspect of semantical consistency. In fact, it is essential to maintain the consistency of data stored in different databases, which evolve independently but whose data can be related. For instance, suppose that there are two distinct databases, one containing information about users of a library, and the other one about students of a school. Even if the databases ignore each other, there is a policy that connects their data: when a student enters the school or leaves it, the library is required to add a new user or to delete the user respectively.

We have faced this problem in the field of deductive databases by introducing the language HU-Datalog [18], a logical language able to support cooperative queries, updates and update propagation. We model the sources of information as deductive databases, sharing the same logical language to express queries and updates, but containing independent, even if possibly related, data. Although this represents a separate work, not integrated in the main stream of the thesis, as future work it would be interesting to enrich STILan with updates and active rules in the style of HU-Datalog to cope with the problem of consistency maintenance.
Chapter 8

Conclusions

In this thesis we have proposed a framework where temporal and spatial information can be represented and handled, and, at the same time, knowledge can be separated into different theories and combined by means of meta-level composition operations.

In particular, we have defined a language MuTACL that joins the advantages of TACL in handling temporal information, with the ability to structure and compose programs. Representing knowledge in separate programs allow us to use knowledge from different sources. Thus information can be stored at different sites and combined in a modular way by employing the meta-level operations. This modular approach favors the reuse of the knowledge encoded in the programs for future applications. Inside MuTACL we can describe discrete or continuous time, work with time points and time periods, and model definite, indefinite and periodic temporal information, which can be distributed among different theories. Moreover, MuTACL allows for the representation of spatial data, and for the definition of facilities to tackle the complexity of these data. For instance we have described how the classical set-theoretic operations (union, intersection, etc.) between spatial objects can be defined in MuTACL. The fact that spatial and temporal data are integrated in the same framework offers the possibility to establish spatio-temporal correlations, e.g., of expressing time-varying areas.

MuTACL retains the appealing feature of (constraint) logic programming of having a well-founded definition of alternative equivalent semantics. More precisely we have defined a top-down semantics via a meta-interpreter, and a bottom-up semantics based on an immediate consequence operator, and we have proved that the meta-interpreter is sound and complete with respect to the bottom-up semantics. Furthermore, a number of algebraic properties of the composition operations can be established by devising an algebra of MuTACL program expressions.

We have shown how MuTACL can be used to integrate GIS technology and constraint logic programming, in order to supply the user with a declarative language which supports and improves GIS analysis at least at the specification level. In this way a declarative language is provided where spatio-temporal and thematic information can be represented in a uniform way and we can exploit the features
of constraint logic programming such as (recursion and constraint handling), and of program composition to perform sophisticated spatio-temporal reasoning. Moreover, the fact of having a unifying language is also promising for what concerns the interoperability among different GIS, which is nowadays very difficult and a hot research area [117, 37, 38].

Finally, we have given some suggestions on how MuTACL can be employed to attain interoperability among different knowledge bases. This led us to the definition of STILan, a language which enriches MuTACL with a message passing construct, a very important mechanism through which one can select either the source where information is extracted from or the system in which a query has to be evaluated. The features of MuTACL, joined with the message passing capability, make STILan a powerful tool both for the representation and for the reasoning on complex data, such as spatial, temporal and spatio-temporal data.

An interesting topic for future research is the treatment of negation. We have already discussed that in the line of Frühwirth, we can embody the "negation by default" of logic programming into MuTACL by exploiting the logical equalities proved in [50]:

\[ \neg \text{th} I \leftrightarrow \neg \text{in} I \quad (\neg \text{in} I) \leftrightarrow \neg (A \text{th} I) \]

Consequently, the meta-interpreter is extended with two clauses which use such equalities:

\[
\text{demo}(E, \neg \text{th} I) \leftarrow \neg \text{demo}(E, A \text{ in} I) \\
\text{demo}(E, \neg \text{in} I) \leftarrow \neg \text{demo}(E, A \text{ th} I)
\]

However the interactions between negation by default and program composition operators is still to be fully understood. Some results on the semantics interactions between operators and negation by default are presented in [24], where, however, the handling of time is not considered.

Another research direction is about designing more powerful composition operations, e.g., hierarchical operators which define hierarchical relations among programs. Moreover, we would like to improve the support for spatial data by investigating some important spatial properties, such as metric properties and topological relations between objects, which, at the moment, are not considered in our framework. We would like also to analyze the possibility of enriching the set of annotations in order to model some other temporal phenomena, like a property which holds in an interval but not in its subintervals, as the ones studied in the ChronoBase Model [106].

Finally, it would be interesting to extend our construction to the more general paradigm of Annotated Constraint Logic where atoms can be labelled by a general class of annotations. The paradigm ACL, endowed with high-level mechanisms for structuring programs and for combining them, could be useful, in particular, to provide a uniform treatment of spatio-temporal information both represented by means of annotations.
Appendix A

Proofs of soundness and completeness

A.1 Meta-interpreter for programs with annotated clauses

This section contains the proof of Theorem 3, i.e., we prove the soundness and completeness of the meta-interpreter defined in Section 5.1.1 with respect to the least fixpoint semantics of a program expression presented in Section 5.1.1. We split the proof by showing separately the statements 1 and 2, which express soundness, proved in Theorem 8, and completeness, proved in Theorem 9, respectively.

In the following we will use \( \mathcal{P}, \mathcal{Q} \) and \( \mathcal{R} \) to denote program expressions, \( A, B \) (possibly with subscripts) to denote ground atoms, \( I, J, K \) and \( H \) (possibly with subscripts) to denote ground intervals and, given a program expression \( \mathcal{P}, V \) will denote the meta-program containing the meta-level representation of the object level programs occurring in \( \mathcal{P} \), the axiomatization of time points, the order relation between time points, the subinterval relation \( (\subseteq) \), the intersection operation \( (\wedge) \) between intervals, and the meta-program consisting of the clauses \((5.2)-(5.7)\).

A.1.1 Soundness

In order to prove the soundness of the meta-interpreter, we first show two lemmata. The first one states that if a conjunctive goal is provable in an interval \( I \), then its atomic conjuncts are provable in intervals whose intersection contains \( I \).

Lemma 2 Let \( \mathcal{P} \) be a program expression and \( V \) be the corresponding meta-program. For any object level atomic formulae \( B_1, \ldots, B_n \) and any ground interval \( I \) the following statement holds:

\[
\text{for all } h \quad \text{demo}(\mathcal{P}, (B_1, \ldots, B_n), I) \in T^h_I \implies \exists I_1, \ldots, I_n : \\
\{\text{demo}(\mathcal{P}, B_1, I_1), \ldots, \text{demo}(\mathcal{P}, B_n, I_n)\} \subseteq T^h_V \land I \subseteq \bigcap_{j=1}^{n} I_j.
\]
Proof For \( n = 1 \) the implication trivially holds. For \( n \geq 2 \) the proof can be carried out by induction on \( h \) exploiting the definition of \( T_V \) and clause (5.3) of the meta-interpreter.

The second lemma states that if we can derive a clause \( A \leftarrow B_1, \ldots, B_n \), holding in \( I_0 \), from the meta-program \( V \), and \( B_1, \ldots, B_n \), holding in certain intervals, belong to an interpretation \( \mathcal{I} \), then the head of the clause, \( A \), is derivable from \( \mathcal{P} \) and it holds in the intersection of the validity intervals of the clause and of its body.

**Lemma 3 (Virtual Clauses Lemma)** Let \( \mathcal{P} \) be a program expression and \( V \) be the corresponding meta-program. For any object level atomic formulae \( A, B_1, \ldots, B_n \), any ground intervals \( I_0, \ldots, I_n \) and any object level interpretation \( \mathcal{I} \), the following statement holds:

\[
\text{clause}(\mathcal{P}, A, (B_1, \ldots, B_n), I_0) \in T_V^\circ \land \{(B_1, I_1), \ldots, (B_n, I_n)\} \subseteq \mathcal{I} \land \\
\left(\bigcap_{j=0,n} I_j\right) \text{ not empty} \implies (A, \bigcap_{j=0,n} I_j) \in T_P^{\text{int}}(\mathcal{I})
\]

**Proof** The proof is by structural induction on \( \mathcal{P} \).

(\( \mathcal{P} \) is a plain program \( P \)).

\[
\text{clause}(P, A, (B_1, \ldots, B_n), I_0) \in T_V^\circ \land \{(B_1, I_1), \ldots, (B_n, I_n)\} \subseteq \mathcal{I} \land \\
\left(\bigcap_{j=0,n} I_j\right) \text{ not empty} \implies \\
\{P \text{ is a plain program and definition of } T_V \text{ and clause (5.8)}\} \\
A \leftarrow B_1, \ldots, B_n \square I_0 \in \text{ground}_L(P) \land \{(B_1, I_1), \ldots, (B_n, I_n)\} \subseteq \mathcal{I} \land \\
\left(\bigcap_{j=0,n} I_j\right) \text{ not empty} \implies \\
\{\text{definition of } T_P^{\text{int}}\} \\
(A, \bigcap_{j=0,n} I_j) \in T_P^{\text{int}}(\mathcal{I})
\]

(\( \mathcal{P} = \mathcal{Q} \cup \mathcal{R} \)).

\[
\text{clause}(\mathcal{Q} \cup \mathcal{R}, A, (B_1, \ldots, B_n), I_0) \in T_V^\circ \land \\
\{(B_1, I_1), \ldots, (B_n, I_n)\} \subseteq \mathcal{I} \land \left(\bigcap_{j=0,n} I_j\right) \text{ not empty} \implies \\
\{\text{definition of } T_V \text{ and clauses (5.5) and (5.6)}\} \\
\{(\text{clause}(\mathcal{Q}, A, (B_1, \ldots, B_n), I_0) \in T_V^\circ \land \{(B_1, I_1), \ldots, (B_n, I_n)\} \subseteq \mathcal{I}\} \lor \\
\{(\text{clause}(\mathcal{R}, A, (B_1, \ldots, B_n), I_0) \in T_V^\circ \land \{(B_1, I_1), \ldots, (B_n, I_n)\} \subseteq \mathcal{I}\}\} \land \\
\left(\bigcap_{j=0,n} I_j\right) \text{ not empty} \implies \\
\{\text{inductive hypothesis}\} \\
(A, \bigcap_{j=0,n} I_j) \in T_Q^{\text{int}}(\mathcal{I}) \lor (A, \bigcap_{j=0,n} I_j) \in T_Q^{\text{int}}(\mathcal{I}) \implies \\
\{\text{definition of } T_Q^{\text{int}}\} \\
(A, \bigcap_{j=0,n} I_j) \in T_Q^{\text{int}}(\mathcal{I})
\]
\( \mathcal{P} = \mathcal{Q} \cap \mathcal{R} \).

\[
\text{clause}(\mathcal{Q} \cap \mathcal{R}, A, (B_1, \ldots, B_n), I_0) \in T^\omega_V \wedge \\
\{(B_1, I_1), \ldots, (B_n, I_n)\} \subseteq \mathcal{I} \wedge \left( \bigcap_{j=0,n} I_j \right) \text{ not empty}
\]
\[\implies \{\text{definition of } T_V, \text{ clause (5.7) and remark 5.10 and } 0 \leq m \leq n\} \]
\[\exists J, K : \text{ clause}(\mathcal{Q}, A, (B_1, \ldots, B_m), K) \in T^\omega_V \wedge \\
\text{clause}(\mathcal{R}, A, (B_{m+1}, \ldots, B_n), J) \in T^\omega_V \wedge I_0 = K \cap J \wedge \\
\{(B_1, I_1), \ldots, (B_n, I_n)\} \subseteq \mathcal{I} \wedge \left( \bigcap_{j=0,n} I_j \right) \text{ not empty}
\]
\[\implies \{\text{inductive hypothesis}\} \\
\left( A, K \cap \left( \bigcap_{j=1,n} I_j \right) \right) \in T^\text{int}_{\mathcal{Q}}(\mathcal{I}) \wedge \\
I_0 = K \cap J \wedge \left( \bigcap_{j=0,n} I_j \right) \text{ not empty}
\]
\[\implies \{\text{definition of } \cap' \text{ and properties of } \cap\} \\
\left( A, \bigcap_{j=0,n} I_j \right) \in T^\text{int}_{\mathcal{Q} \cap \mathcal{R}}(\mathcal{I})
\]

\[\blacksquare\]

The soundness of the meta-interpreter states that an object level atom holding in a certain interval \( H \) is derivable from \( \mathcal{P} \) if its meta-level representation holding in an interval \( I \) contained in \( H \) is derivable from \( V \).

**Theorem 8 (Soundness)** Let \( \mathcal{P} \) be a program expression and let \( V \) be the corresponding meta-program. For any object level atomic formula \( A \) and any ground interval \( I \), the following statement holds:

\[\text{demo}(\mathcal{P}, A, I) \in T^\omega_V \implies \exists H : (A, H) \in (T^\text{int}_\mathcal{P})^\omega \wedge I \subseteq H.\]

**Proof** We first show that for all \( i \)

\[\text{demo}(\mathcal{P}, A, I) \in T^i_V \implies \exists H : (A, H) \in (T^\text{int}_\mathcal{P})^\omega \wedge I \subseteq H. \quad (A.1)\]

The proof is by induction on \( i \).

**(Base case).** Trivial since \( T^0_V = \emptyset \).

**(Inductive case).** Assume that

\[\text{demo}(\mathcal{P}, A, I) \in T^i_V \implies \exists H : (A, H) \in (T^\text{int}_\mathcal{P})^\omega \wedge I \subseteq H.\]

Then:

\[\text{demo}(\mathcal{P}, A, I) \in T^{i+1}_V \iff \{\text{definition of } T^i_V\} \]
\[\text{demo}(\mathcal{P}, A, I) \in T^i_V(T^i_V) \iff \{\text{definition of } T_V, \text{ clause (5.4) and remarks 5.9 and 5.10}\} \]
\[ \exists K, J : \{ \text{clause}(\mathcal{P}, A, G, K), \text{demo}(\mathcal{P}, G, J) \} \subseteq T^i_V \land I \subseteq K \cap J \]
\[ \implies \{ \text{Lemma 2 and } G = B_1, \ldots, B_n \} \]
\[ \exists I_1, \ldots, I_n : \text{clause}(\mathcal{P}, A, (B_1, \ldots, B_n), K) \in T^i_V \land \]
\[ \{ \text{demo}(\mathcal{P}, B_1, I_1), \ldots, \text{demo}(\mathcal{P}, B_n, I_n) \} \subseteq T^i_V \land J \subseteq \bigcap_{j=1}^n I_j \land I \subseteq K \cap J \]
\[ \implies \{ \text{inductive hypothesis} \} \]
\[ \exists H_1, \ldots, H_n : \text{clause}(\mathcal{P}, A, (B_1, \ldots, B_n), K) \in T^i_V \land \]
\[ \{(B_1, H_1), \ldots, (B_n, H_n)\} \subseteq (T^{\text{int}}_P)^\omega \land \]
\[ I_1 \subseteq H_1 \land \ldots \land I_n \subseteq H_n \land J \subseteq \bigcap_{j=1}^n I_j \land I \subseteq K \cap J \]
\[ \implies \{ \text{monotonicity of } T_V \text{ and } I \subseteq K \cap J \text{ and } J \subseteq \bigcap_{j=1}^n I_j \} \]
\[ \text{clause}(\mathcal{P}, A, (B_1, \ldots, B_n), K) \in T^\omega_V \land \{(B_1, H_1), \ldots, (B_n, H_n)\} \subseteq (T^{\text{int}}_P)^\omega \land \]
\[ I_1 \subseteq H_1 \land \ldots \land I_n \subseteq H_n \land I \subseteq K \cap \left( \bigcap_{j=1}^n I_j \right) \]
\[ \implies K \cap \left( \bigcap_{j=1}^n H_j \right) \text{ is not empty because} \]
\[ K \cap \left( \bigcap_{j=1}^n I_j \right) \subseteq K \cap \left( \bigcap_{j=1}^n H_j \right) \text{ and } K \cap \left( \bigcap_{j=1}^n I_j \right) \text{ is not empty} \]
\[ \text{since it contains } I \]
\[ \left( A, K \cap \left( \bigcap_{j=1}^n H_j \right) \right) \in (T^{\text{int}}_P)^{\omega} \land I \subseteq K \cap \left( \bigcap_{j=1}^n H_j \right) \]
\[ \implies \{(T^{\text{int}}_P)^{\omega} \text{ is a fixpoint of } T^{\text{int}}_P \} \]
\[ \left( A, K \cap \left( \bigcap_{j=1}^n H_j \right) \right) \in (T^{\text{int}}_P)^{\omega} \land I \subseteq K \cap \left( \bigcap_{j=1}^n H_j \right) \]

We are now able to prove the soundness of the meta-interpreter with respect to the least fixpoint semantics.

\[ \text{demo}(\mathcal{P}, A, I) \in T^\omega_V \]
\[ \implies \{ \text{definition of } T^\omega_V \} \]
\[ \exists : \text{demo}(\mathcal{P}, A, I) \in T^i_V \]
\[ \implies \{ \text{Statement (A.1)} \} \]
\[ \exists H : (A, H) \in (T^{\text{int}}_P)^{\omega} \land I \subseteq H \]

\section*{A.1.2 Completeness}

In order to prove the completeness of the meta-interpreter we first prove a technical lemma.

\textbf{Lemma 4} Let \( \mathcal{P} \) be a program expression and let \( V \) be the corresponding meta-program. For any object level atomic formula \( A \), any ground interval \( I \) and any object level interpretation \( \mathcal{I} \), the following statement holds:

\[ (A, I) \in T^{\text{int}}_P(\mathcal{I}) \implies \exists (B_1, I_1), \ldots, (B_n, I_n), I_0 : \]
\[ \text{clause}(\mathcal{P}, A, (B_1, \ldots, B_n), I_0) \in T^\omega_V \land \]
\[ \{(B_1, I_1), \ldots, (B_n, I_n)\} \subseteq \mathcal{I} \land I = \bigcap_{j=0}^n I_j. \]
**Proof** The proof is by structural induction on \( \mathcal{P} \).

\( \mathcal{P} \text{ is a plain program } \mathcal{P} \).

\[(A, I) \in T^\text{int}_\mathcal{P}(\mathcal{I}) \]
\[\iff \{ \text{definition of } T^\text{int}_\mathcal{P} \} \]
\[\exists (B_1, I_1), \ldots, (B_n, I_n), I_0: \]
\[A \leftarrow B_1, \ldots, B_n \square I_0 \in \text{ground}_L(\mathcal{P}) \land \{(B_1, I_1), \ldots, (B_n, I_n)\} \subseteq \mathcal{I} \land \]
\[I = \bigcap_{j=0}^{n} I_j \land I \neq \emptyset \]
\[\implies \{ \text{definition of meta-level representation} \} \]
\[\exists (B_1, I_1), \ldots, (B_n, I_n), I_0: \]
\[\text{clause}(P, A, (B_1, \ldots, B_n), I_0) \leftarrow \text{nempty}(I_0) \in \text{ground}_L(V) \land \]
\[\{(B_1, I_1), \ldots, (B_n, I_n)\} \subseteq \mathcal{I} \land I = \bigcap_{j=0}^{n} I_j \land I \neq \emptyset \]
\[\implies \{ \text{definition and monotonicity of } T^\nu \text{ and } I \neq \emptyset \} \]
\[\exists (B_1, I_1), \ldots, (B_n, I_n), I_0: \text{clause}(P, A, (B_1, \ldots, B_n), I_0) \in T^\nu \land \]
\[\{(B_1, I_1), \ldots, (B_n, I_n)\} \subseteq \mathcal{I} \land I = \bigcap_{j=0}^{n} I_j \]

\( \mathcal{P} = \mathcal{Q} \cup \mathcal{R} \).

\[(A, I) \in T^\text{int}_{\mathcal{Q} \cup \mathcal{R}}(\mathcal{I}) \]
\[\iff \{ \text{definition of } T^\text{int}_{\mathcal{Q} \cup \mathcal{R}} \} \]
\[(A, I) \in T^\text{int}_\mathcal{Q}(\mathcal{I}) \cup T^\text{int}_\mathcal{R}(\mathcal{I}) \]
\[\iff \{ \text{property of set-theoretic union} \} \]
\[(A, I) \in T^\text{int}_\mathcal{Q}(\mathcal{I}) \lor (A, I) \in T^\text{int}_\mathcal{R}(\mathcal{I}) \]
\[\implies \{ \text{inductive hypothesis} \} \]
\[\exists (B^1_1, I^1_1), \ldots, (B^1_n, I^1_n), I^1_0: \text{clause}(\mathcal{Q}, A, (B^1_1, \ldots, B^1_n), I^1_0) \in T^\nu \land \]
\[\{(B^1_1, I^1_1), \ldots, (B^1_n, I^1_n)\} \subseteq \mathcal{I} \land I = \bigcap_{j=0}^{n} I^1_j \lor \]
\[\exists (B^2_1, I^2_1), \ldots, (B^2_m, I^2_m), I^2_0: \text{clause}(\mathcal{R}, A, (B^2_1, \ldots, B^2_m), I^2_0) \in T^\nu \land \]
\[\{(B^2_1, I^2_1), \ldots, (B^2_m, I^2_m)\} \subseteq \mathcal{I} \land I = \bigcap_{j=0}^{m} I^2_j \]
\[\implies \{ \text{clauses (5.5) and (5.6) and } T^\nu \text{ is a fixpoint of } T^\nu \} \]
\[\exists (B^1_1, I^1_1), \ldots, (B^1_n, I^1_n), I^1_0: \text{clause}(\mathcal{Q} \cup \mathcal{R}, A, (B^1_1, \ldots, B^1_n), I^1_0) \in T^\nu \land \]
\[\{(B^1_1, I^1_1), \ldots, (B^1_n, I^1_n)\} \subseteq \mathcal{I} \land I = \bigcap_{j=0}^{n} I^1_j \lor \]
\[\exists (B^2_1, I^2_1), \ldots, (B^2_m, I^2_m), I^2_0: \text{clause}(\mathcal{Q} \cup \mathcal{R}, A, (B^2_1, \ldots, B^2_m), I^2_0) \in T^\nu \land \]
\[\{(B^2_1, I^2_1), \ldots, (B^2_m, I^2_m)\} \subseteq \mathcal{I} \land I = \bigcap_{j=0}^{m} I^2_j \]

which clearly implies the thesis.

\( \mathcal{P} = \mathcal{Q} \cap \mathcal{R} \).

\[(A, I) \in T^\text{int}_{\mathcal{Q} \cap \mathcal{R}}(\mathcal{I}) \]
\[\iff \{ \text{definition of } T^\text{int}_{\mathcal{Q} \cap \mathcal{R}} \} \]
\[(A, I) \in T^\text{int}_\mathcal{Q}(\mathcal{I}) \cap T^\text{int}_\mathcal{R}(\mathcal{I}) \]
\[\iff \{ \text{definition of } \cap' \} \]
\[\exists (A, K) \in T^\text{int}_\mathcal{Q}(\mathcal{I}) \land \exists (A, J) \in T^\text{int}_\mathcal{R}(\mathcal{I}): I = K \land J \land I \neq \emptyset \]
\[\implies \{ \text{inductive hypothesis} \} \]
\[\exists (B^1_1, I^1_1), \ldots, (B^1_n, I^1_n), I^1_0: \text{clause}(\mathcal{Q}, A, (B^1_1, \ldots, B^1_n), I^1_0) \in T^\nu \land \]
\[\{(B^1_1, I^1_1), \ldots, (B^1_n, I^1_n)\} \subseteq \mathcal{I} \land K = \bigcap_{j=0}^{n} I^1_j \land
(\exists (B_1^2, I_1^2), \ldots, (B_m^2, I_m^2), I_0^2 : \text{clause}(\mathcal{R}, A, (B_1^2, \ldots, B_m^2), I_0^2) \in T_V^\omega \land
\{ (B_1^2, I_1^2), \ldots, (B_m^2, I_m^2) \} \subseteq \mathcal{I} \land J = \bigcap_{i=0,m} I_j^2 \land I = K \cap J
\implies \{ \text{clause (5.7), remark 5.10 and } T_V^\omega \text{ is a fixpoint of } T_V \}
\exists (B_1^1, I_1^1), \ldots, (B_n^1, I_n^1), (B_1^2, I_1^2), \ldots, (B_m^2, I_m^2), I_0, I_0^2 : \text{clause}(Q \cap \mathcal{R}, A, (B_1^1, \ldots, B_n^1, B_1^2, \ldots, B_m^2), I_0) \in T_V^\omega \land
\{ (B_1^1, I_1^1), \ldots, (B_n^1, I_n^1), (B_1^2, I_1^2), \ldots, (B_m^2, I_m^2) \} \subseteq \mathcal{I} \land
I_0 = I_1^1 \cap I_0^2 \land I = \left( \bigcap_{i=0,n} I_i^1 \right) \cap \left( \bigcap_{j=0,m} I_j^2 \right)
\implies \{ \text{Lemma 4} \}
\exists (B_1, I_1), \ldots, (B_h, I_h), I_0 : \text{clause}(Q \cap \mathcal{R}, A, (B_1, \ldots, B_h), I_0) \in T_V^\omega \land
\{ (B_1, I_1), \ldots, (B_h, I_h) \} \subseteq \mathcal{I} \land I = \bigcap_{j=0,h} I_j
\]

\[\blacksquare\]

Now we can prove the completeness of the meta-interpreter with respect to the least fixpoint semantics.

**Theorem 9 (Completeness)** Let \( \mathcal{P} \) be a program expression and let \( V \) be the corresponding meta-program. For any object level atomic formula \( A \) and any ground interval \( I \), the following statement holds:

\[(A, I) \in (T_{\mathcal{P}}^{\text{int}})^\omega \implies \text{demo}(\mathcal{P}, A, I) \in T_V^\omega.\]

**Proof** We first show that for all \( h \)

\[(A, I) \in (T_{\mathcal{P}}^{\text{int}})^h \implies \text{demo}(\mathcal{P}, A, I) \in T_V^\omega.\] (A.2)

The proof is by induction on \( h \).

(Base case). Trivial since \( (T_{\mathcal{P}}^{\text{int}})^0 = \emptyset \).

(Inductive case). Assume that \((A, I) \in (T_{\mathcal{P}}^{\text{int}})^h \implies \text{demo}(\mathcal{P}, A, I) \in T_V^\omega \).

Then:

\[(A, I) \in (T_{\mathcal{P}}^{\text{int}})^{h+1} \iff \{ \text{definition of } (T_{\mathcal{P}}^{\text{int}})^{\alpha} \}\]

\[(A, I) \in T_{\mathcal{P}}^{\text{int}}((T_{\mathcal{P}}^{\text{int}})^h) \iff \{ \text{Lemma 4} \}\]

\[\exists (B_1, I_1), \ldots, (B_n, I_n), I_0 : \text{clause}(\mathcal{P}, A, (B_1, \ldots, B_n), I_0) \in T_V^\omega \land
\{ (B_1, I_1), \ldots, (B_n, I_n) \} \subseteq (T_{\mathcal{P}}^{\text{int}})^h \land I = \bigcap_{i=0,n} I_j \iff \{ \text{inductive hypothesis} \}\]

\[\text{clause}(\mathcal{P}, A, (B_1, \ldots, B_n), I_0) \in T_V^\omega \land
\{ \text{demo}(\mathcal{P}, B_1, I_1), \ldots, \text{demo}(\mathcal{P}, B_n, I_n) \} \subseteq T_V^\omega \land I = \bigcap_{j=0,n} I_j.\]
\[ \implies \{ \text{definition of } T_V \text{ and clause (5.3) used } n-1 \text{ times, remark (5.9) and } T_V^\omega \text{ is a fixpoint of } T_V \} \]

\[ \text{clause}(\mathcal{P}, A, (B_1, \ldots, B_n), I_0) \in T_V^\omega \land \text{demo}(\mathcal{P}, (B_1, \ldots, B_n), J) \in T_V^\omega \land \]

\[ J = \bigcap_{j=1}^n I_j \land I = \bigcap_{j=0}^n I_j \]

\[ \implies \{ \text{definition of } T_V \text{ and clause (5.4), remarks (5.9) and (5.10) and } T_V^\omega \text{ is a fixpoint of } T_V \} \]

\[ \text{demo}(\mathcal{P}, A, I) \in T_V^\omega \]

We now prove the completeness of the meta-interpreter of the program expressions with respect to the least fixpoint semantics.

\[ (A, I) \in (T_P^{int})^\omega \]

\[ \implies \{ \text{definition of } (T_P^{int})^\omega \} \]

\[ \exists h : (A, I) \in (T_P^{int})^h \]

\[ \implies \{ \text{statement (A.2)} \} \]

\[ \text{demo}(\mathcal{P}, A, I) \in T_V^\omega \]

\[ \blacksquare \]

### A.2 MuTACLP meta-interpreter

This section presents the full proofs of the soundness and completeness results for MuTACLP meta-interpreter. We first fix some notational conventions. In the following we will denote by \( \mathcal{E}, \mathcal{N}, \mathcal{R} \) and \( \mathcal{Q} \) generic program expressions, and by \( \mathcal{C} \) the fixed constraint domain where the constraints of object programs are interpreted. Let \( \mathcal{M} \) be the fixed constraint domain, where the constraints of the meta-interpreter defined in Section 5.3.2 are interpreted. We denote by \( A, B \) elements of \( \mathcal{C} \)-base\(_L\), with \( \alpha, \beta, \gamma \) annotations in \( Ann \) and by \( C \) a \( \mathcal{C} \)-ground instance of a constraint. All symbols may have subscripts. In the following for simplicity we will drop the reference to \( \mathcal{C} \) and \( \mathcal{M} \) in the name of the immediate consequence operators. Moreover we refer the program containing the meta-level representation of object level programs and clauses (5.20)-(5.29) as “the meta-program \( V \) corresponding to a program expression”.

We will say that an interpretation \( I \subseteq \mathcal{C} \)-base\(_L \times Ann \) satisfies the body of a \( \mathcal{C} \)-ground instance \( A \alpha \leftarrow C_1, \ldots, C_k, B_1 \alpha_1, \ldots, B_n \alpha_n \) of a clause, or in symbols \( I \models C_1, \ldots, C_k, B_1 \alpha_1, \ldots, B_n \alpha_n \), if

1. \( \mathcal{D} \models C_1, \ldots, C_k \) and

2. there are annotations \( \beta_1, \ldots, \beta_n \) such that \( \{(B_1, \beta_1), \ldots, (B_n, \beta_n)\} \subseteq I \) and \( \mathcal{D} \models \alpha_1 \subseteq \beta_1, \ldots, \alpha_n \subseteq \beta_n \).
Furthermore, will often denote a sequence $C_1, \ldots, C_k$ of $\mathcal{C}$-ground instances of constraints by $\bar{C}$, while a sequence $B_1 \alpha_1, \ldots, B_n \alpha_n$ of annotated atoms in $\mathcal{C}$-base $l \times Ann$ will be denoted by $\bar{B}$. For example, with this convention a clause of the kind $A \alpha \leftarrow C_1, \ldots, C_k, B_1 \alpha_1, \ldots, B_n \alpha_n$ will be written as $A \alpha \leftarrow \bar{C}, \bar{B}$, and, similarly, in the meta-level representation, we will write $\text{clause}(\mathcal{E}, A \alpha, (\bar{C}, \bar{B}))$ in place of $\text{clause}(\mathcal{E}, A \alpha, (C_1, \ldots, C_k, B_1 \alpha_1, \ldots, B_n \alpha_n))$.

### A.2.1 Soundness

In order to show the soundness of the meta-interpreter (restricted to the atoms of interest), we first prove the following lemma stating that if a conjunctive goal is provable at the meta-level then also its atomic conjuncts are provable at the meta-level.

**Lemma 5** Let $\mathcal{E}$ be a program expression and let $V$ be the corresponding meta-interpreter. For any $B_1 \alpha_1, \ldots, B_n \alpha_n$ with $B_i \in \mathcal{C}$-base $l$ and $\alpha_i \in Ann$ and for any $C_1, \ldots, C_k$, with $C_i$ a $\mathcal{C}$-ground instance of a constraint, we have:

For all $h$

\[
\text{demo}(\mathcal{E}, (C_1, \ldots, C_k, B_1 \alpha_1, \ldots, B_n \alpha_n)) \in T^h_V \implies \{\text{demo}(\mathcal{E}, B_1 \alpha_1), \ldots, \text{demo}(\mathcal{E}, B_n \alpha_n)\} \subseteq T^h_V \land \mathcal{D}_C \models C_1, \ldots, C_k
\]

**Proof** The result easily follows from the definition of $T^h_V$ and from clauses (5.21) and (5.26) of the meta-interpreter. We next present the detailed proof.

Notice that if $n = 1$ and $k = 0$ the proof is immediate and if $n = 0$ and $k = 1$ the claim trivially follows by the definition of $T^h_V$ and clause (5.26).

For $n + k \geq 2$, the proof is by induction on $h$.

**Base case.** Trivial since $T^0_V = \emptyset$.

**Inductive case.** Assume that

\[
\text{demo}(\mathcal{E}, (C_1, \ldots, C_k, B_1 \alpha_1, \ldots, B_n \alpha_n)) \in T^h_V \implies \{\text{demo}(\mathcal{E}, B_1 \alpha_1), \ldots, \text{demo}(\mathcal{E}, B_n \alpha_n)\} \subseteq T^h_V \land \mathcal{D}_C \models C_1, \ldots, C_k
\]

Then:

\[
\text{demo}(\mathcal{E}, (C_1, \ldots, C_k, B_1 \alpha_1, \ldots, B_n \alpha_n)) \in T^{h+1}_V \iff \{\text{definition of } T^h_V\}
\]

\[
\text{demo}(\mathcal{E}, (C_1, \ldots, C_k, B_1 \alpha_1, \ldots, B_n \alpha_n)) \in T_V(T^h_V) \iff \{\text{definition of } T_V, \text{ clause (5.21) and }\}
\]

\[
(G_1, \ldots, G_{n+k}) = (C_1, \ldots, C_k, B_1 \alpha_1, \ldots, B_n \alpha_n)
\]

\[
\exists 1 \leq r \leq n + k : \{\text{demo}(\mathcal{E}, (G_1, \ldots, G_r)), \text{demo}(\mathcal{E}, (G_{r+1}, \ldots, G_{n+k}))\} \subseteq T^h_V \implies \{\text{inductive hypothesis or previous remark if } r = 1 \text{ or } r = n + k - 1\}
\]

\[
\{\text{demo}(\mathcal{E}, G_i) \mid 1 \leq i \leq r, \ G_i \text{ is an annotated atom}\} \subseteq T^h_V \land
\]
\[ D_C \models G_{m_1}, \ldots, G_{m_k} \text{ with } G_{m_1}, \ldots, G_{m_k} \text{ the constraints in } \{G_1, \ldots, G_r\} \land \\
\{\text{demo}(E, G_i) \mid r + 1 \leq i \leq n + k, \ G_i \text{ is an annotated atom} \} \subseteq T^h \land \\
D_C \models G_{t_1}, \ldots, G_{t_j} \text{ with } G_{t_1}, \ldots, G_{t_j} \text{ the constraints in } \{G_{r+1}, \ldots, G_{n+k}\} \\
\implies \{\text{monotonicity of } T_V\} \\
\{\text{demo}(E, B_1 \alpha_1), \ldots, \text{demo}(E, B_n \alpha_n)\} \subseteq T^{h+1} \land \ D_C \models C_1, \ldots, C_k \]

The next two lemmata relate the clauses computed from a program expression \( E \) at the meta-level, called “virtual clauses”, with the set of consequences of \( E \). The first lemma states that whenever we can find a virtual clause computed from \( E \) whose body is satisfied by \( I \), the head \( A \alpha \) of the clause is a consequence of the program expression \( E \). The second one shows how the head of a virtual clause can be “joined” with an already existing annotated atom in order to obtain an atom with a larger th annotation.

**Lemma 6 (Virtual Clauses Lemma 1)** Let \( E \) be a program expression and \( V \) be the corresponding meta-interpreter. For any sequence \( \bar{C} \) of \( C \)-ground instances of constraints, for any \( A \alpha, \bar{B} \) in \( C\text{-base}_L \times \text{Ann} \) and any interpretation \( I \subseteq C\text{-base}_L \times \text{Ann} \), we have:

\[
\text{clause}(E, A \alpha, (\bar{C}, \bar{B})) \in T^*_V \land I \models \bar{C}, \bar{B} \implies (A, \alpha) \in T_E(I).
\]

**Proof** The proof is by structural induction on \( E \).

(\( E \) is a plain program \( P \)).

\[
\text{clause}(P, A \alpha, (\bar{C}, \bar{B})) \in T^*_V \land I \models \bar{C}, \bar{B} \\
\implies \{\text{definition of } T_V \text{ and clause (5.30)}\} \\
A \alpha \leftarrow \bar{C}, \bar{B} \in \text{ground}_C(P) \land I \models \bar{C}, \bar{B} \\
\implies \{\text{definition of } T_P\} \\
(A, \alpha) \in T_P(I)
\]

(\( E = Q \cup R \)).

\[
\text{clause}(Q \cup R, A \alpha, (\bar{C}, \bar{B})) \in T^*_V \land I \models \bar{C}, \bar{B} \\
\implies \{\text{definition of } T_V \text{ and clauses (5.27) and (5.28)}\} \\
(clause(Q, A \alpha, (\bar{C}, \bar{B})) \in T^*_V \land I \models \bar{C}, \bar{B} \lor \\
clause(R, A \alpha, (\bar{C}, \bar{B})) \in T^*_V \land I \models \bar{C}, \bar{B} \\
\implies \{\text{inductive hypothesis}\} \\
(A, \alpha) \in T_Q(I) \lor (A, \alpha) \in T_R(I) \\
\implies \{\text{definition of } T_{Q \cup R}\} \\
(A, \alpha) \in T_{Q \cup R}(I)
\]

(\( E = Q \cap R \)).

\[
\text{clause}(Q \cap R, A \alpha, (\bar{C}, \bar{B})) \in T^*_V \land I \models \bar{C}, \bar{B} \\
\implies \{(\bar{C}', \bar{B}') \text{ and } (\bar{C}'', \bar{B}'') \text{ partition of } (\bar{C}, \bar{B})\}, \text{ definition of } T_V \text{ and clause (5.29)}\}
clause(Q, Aβ, (C', B)) ∈ TV ∧ clause(R, Aγ, (C'', B'')) ∈ TV ∧ I |= C', B' ∧ I |= C'', B'' ∧ DC |= β ∩ γ = α

⇒ {inductive hypothesis}

(A, β) ∈ TQ(I) ∧ (A, γ) ∈ TR(I) ∧ DC |= β ∩ γ = α

⇒ {definition of ⊙}

(A, α) ∈ TQ∩R(I)

Lemma 7 (Virtual Clauses Lemma 2) Let E be a program expression and V be the corresponding meta-program. For any A th[s1, s2], A th[r1, r2], B in C-baseL × Ann, for any sequence C of C-ground instances of constraints, and any interpretation I ⊆ C-baseL ∗ Ann, the following statement holds:

clause(E, A th[s1, s2], (C, B)) ∈ TV ∧ I |= C, B ∧ (A, th[r1, r2]) ∈ I ∧ DC |= s1 < r1, r1 ≤ s2, s2 < r2

⇒ (A, th[s1, r2]) ∈ TE(I)

Proof The proof is by structural induction on E and it is very similar to the proof of Lemma 6.

(E is a plain program P).

clause(P, A th[s1, s2], (C, B)) ∈ TV ∧ I |= C, B ∧ (A, th[r1, r2]) ∈ I ∧ DC |= s1 < r1, r1 ≤ s2, s2 < r2

⇒ {P is a plain program and definition of TV and clause (5.30)}

A th[s1, s2] ← C, B ∈ groundc(P) ∧ I |= C, B ∧ (A, th[r1, r2]) ∈ I ∧ DC |= s1 < r1, r1 ≤ s2, s2 < r2

⇒ {definition of TP}

(A, th[s1, r2]) ∈ TP(I)

(E = Q ∪ R).

clause(Q ∪ R, A th[s1, s2], (C, B)) ∈ TV ∧ I |= C, B ∧ (A, th[r1, r2]) ∈ I ∧ DC |= s1 < r1, r1 ≤ s2, s2 < r2

⇒ {definition of TV and clauses (5.27) and (5.28)}

(clause(Q, A th[s1, s2], (C, B)) ∈ TV ∧ I |= C, B ∧ (A, th[r1, r2]) ∈ I ∧ DC |= s1 < r1, r1 ≤ s2, s2 < r2) ∨

(clause(R, A th[s1, s2], (C, B)) ∈ TV ∧ I |= C, B ∧ (A, th[r1, r2]) ∈ I ∧ DC |= s1 < r1, r1 ≤ s2, s2 < r2)

⇒ {inductive hypothesis}

(A, th[s1, r2]) ∈ TQ(I) ∨ (A, th[s1, r2]) ∈ TR(I)

⇒ {definition of TQ∪R}

(A, th[s1, r2]) ∈ TQ∪R(I)

(E = Q ∩ R).

clause(Q ∩ R, A th[s1, s2], (C, B)) ∈ TV ∧ I |= C, B ∧ (A, th[r1, r2]) ∈ I ∧ DC |= s1 < r1, r1 ≤ s2, s2 < r2
\[
\implies \{ (\bar{C}', \bar{B}') \text{ and } (\bar{C}'', \bar{B}'') \text{ partition of } (\bar{B}, \bar{C}), \text{ definition of } T_V, \text{ axiom } (\text{th} \cap) \text{ and clause } (5.29) \}
\]

\[
\text{clause}(Q, A \text{ th }[t_1, t_2], (\bar{C}', \bar{B}')) \in T_V^\omega \land \text{clause}(R, A \text{ th }[w_1, w_2], (\bar{C}'', \bar{B}'')) \in T_V^\omega \land \\
\s_1 = \max\{t_1, w_1\} \land \s_2 = \min\{t_2, w_2\} \land I \models \bar{C}', \bar{B}' \land I \models \bar{C}'', \bar{B}'' \land \\
(A, \text{ th }[r_1, r_2]) \in I \land D_C \models \s_1 < r_1, r_1 < \s_2, \s_2 < r_2
\]

We distinguish three cases with respect to the relation between \(w_2, t_2\) and \(r_2\).

- \(D_C \models t_2 < r_2, w_2 < r_2\).
  Since \(D_C \models t_1 < r_1, r_1 \leq t_2, t_2 < r_2\) and \(D_C \models w_1 < r_1, r_1 \leq w_2, w_2 < r_2\), by inductive hypothesis we have \((A, \text{ th }[t_1, r_2]) \in \mathbb{T}_Q(I) \land (A, \text{ th }[w_1, r_2]) \in \mathbb{T}_R(I)\).
  Recalling that \(s_1 = \max\{t_1, w_1\}\) and \(D_C \models s_1 < r_2\), by definition of \(\cap\) we obtain \((A, \text{ th }[s_1, r_2]) \in \mathbb{T}_{Q \cap R}(I)\).

- \(D_C \models t_2 < r_2, r_2 \leq w_2\).
  Since \(D_C \models t_1 < r_1, r_1 \leq t_2, t_2 < r_2\), by inductive hypothesis on \(Q\), we have that \((A, \text{ th }[t_1, r_2]) \in \mathbb{T}_Q(I)\). Furthermore, Lemma 6 applied to \(R\) ensures that \((A, \text{ th }[w_1, w_2]) \in \mathbb{T}_R(I)\). Since \(s_1 = \max\{t_1, w_1\}\) and \(D_C \models r_2 \leq w_2, s_1 < r_2\), by definition of \(\cap\) we can conclude \((A, \text{ th }[s_1, r_2]) \in \mathbb{T}_{Q \cap R}(I)\).

- \(D_C \models r_2 \leq t_2, w_2 < r_2\).
  By symmetry from the previous case.

The analyzed cases are exhaustive: \(D_C \not\models r_2 \leq t_2, r_2 \leq w_2\) since \(D_C \models s_2 < r_2\) and \(s_2 = \min\{t_2, w_2\}\).

Now, the soundness of the meta-interpreter can be proved by showing that if an annotated atom \(A \alpha\) is provable at the meta-level from the program expression \(E\) then \(A \gamma\) is a consequence of \(E\) for some \(\gamma\) such that \(A \gamma \Rightarrow A \alpha\), i.e., the annotation \(\alpha\) is less or equal to \(\gamma\).

**Theorem 10 (soundness)** Let \(E\) be a program expression and let \(V\) be the corresponding meta-program. For any \(A \alpha\) with \(A \in \mathcal{C}_{basel}\) and \(\alpha \in \text{Ann}\), the following statement holds:

\[
\text{demo}(E, A \alpha) \in T_V^\omega \implies (A, \alpha) \in \mathbb{F}(E)
\]

**Proof** We first show that for all \(h\)

\[
\text{demo}(E, A \alpha) \in T_V^h \implies \exists \gamma: (A, \gamma) \in T_E^\omega \land D_C \models \alpha \subseteq \gamma. \tag{A.3}
\]

The proof is by induction on \(h\).

(\textbf{Base case}). Trivial since \(T_V^0 = \emptyset\).

(\textbf{Inductive case}). Assume that

\[
\text{demo}(E, A \alpha) \in T_V^h \implies \exists \gamma: (A, \gamma) \in T_E^\omega \land D_C \models \alpha \subseteq \gamma.
\]

Then:

\[
\text{demo}(E, A \alpha) \in T_V^{h+1} \implies \exists \gamma: (A, \gamma) \in T_E^\omega \land D_C \models \alpha \subseteq \gamma.
\]
\[\text{demo}(\mathcal{E}, A \alpha) \in T^b_{V+1}
\]
\[\iff\{\text{definition of } T^b_V\}
\]
\[\text{demo}(\mathcal{E}, A \alpha) \in T_V(T^b_V)\]

We have four cases corresponding to clauses (5.22), (5.23), (5.24) and (5.25). We only show the cases related to clause (5.22) and (5.23) since the others are proved in an analogous way.

(clause (5.22)) \[\{\alpha = \text{th} [t_1, t_2], \text{definition of } T_V \text{ and clause (5.22)}\}\]
\[\{\text{clause}(\mathcal{E}, \text{A th} [s_1, s_2], (\bar{C}, \bar{B})), \text{demo}(\mathcal{E}, (\bar{C}, \bar{B}))\} \subseteq T^b_V \land
\]
\[\mathcal{D}_C \models s_1 \leq t_1, t_2 \leq s_2, t_1 \leq t_2\]
\[\implies\{\text{Lemma 5 and } (\bar{C}, \bar{B}) = (C_1, \ldots, C_k, B_1 \alpha_1, \ldots, B_n \alpha_n)\}\]
\[\text{clause}(\mathcal{E}, \text{A th} [s_1, s_2], (C_1, \ldots, C_k, B_1 \alpha_1, \ldots, B_n \alpha_n)) \in T^b_V \land
\]
\[\{(B_1, \beta_1), \ldots, (B_n, \beta_n)\} \subseteq T^f_{\mathcal{E}} \land \mathcal{D}_C \models \alpha_1 \subseteq \beta_1, \ldots, \alpha_n \subseteq \beta_n \land
\]
\[\mathcal{D}_C \models C_1, \ldots, C_k \land \mathcal{D}_C \models s_1 \leq t_1, t_2 \leq s_2, t_1 \leq t_2\]
\[\implies\{T_V^f = \bigcup_{i \in \mathbb{N}} T^i_V\}\]
\[\text{clause}(\mathcal{E}, \text{A th} [s_1, s_2], (C_1, \ldots, C_k, B_1 \alpha_1, \ldots, B_n \alpha_n)) \in T^f_{\mathcal{E}} \land
\]
\[\{(B_1, \beta_1), \ldots, (B_n, \beta_n)\} \subseteq T^f_{\mathcal{E}} \land \mathcal{D}_C \models \alpha_1 \subseteq \beta_1, \ldots, \alpha_n \subseteq \beta_n \land
\]
\[\mathcal{D}_C \models C_1, \ldots, C_k \land \mathcal{D}_C \models s_1 \leq t_1, t_2 \leq s_2, t_1 \leq t_2\]
\[\implies\{\text{Lemma 6}\}\]
\[\text{A, th} [s_1, s_2] \in T^f_{\mathcal{E}}(T^f_{\mathcal{E}}) \land \mathcal{D}_C \models s_1 \leq t_1, t_2 \leq s_2, t_1 \leq t_2\]
\[\implies\{T^f_{\mathcal{E}} \text{ is a fixpoint of } T^f_{\mathcal{E}} \text{ and } \mathcal{D}_C \models s_1 \leq t_1, t_2 \leq s_2, t_1 \leq t_2\}\]
\[\text{A, th} [s_1, s_2] \in T^f_{\mathcal{E}} \land \mathcal{D}_C \models \text{th} [t_1, t_2] \sqsubseteq \text{th} [s_1, s_2]\]

(clause (5.23)) \[\{\alpha = \text{th} [t_1, t_2], \text{definition of } T_V \text{ and clause (5.23)}\}\]
\[\{\text{clause}(\mathcal{E}, \text{A th} [s_1, s_2], (\bar{C}, \bar{B})), \text{demo}(\mathcal{E}, (\bar{C}, \bar{B})), \text{demo}(\mathcal{E}, \text{A th} [s_2, t_2])\} \subseteq T^b_V \land
\]
\[\mathcal{D}_C \models s_1 \leq t_1, t_1 < s_2, s_2 < t_2\]
\[\implies\{\text{Lemma 5 and } (\bar{C}, \bar{B}) = (C_1, \ldots, C_k, B_1 \alpha_1, \ldots, B_n \alpha_n)\}\]
\[\text{clause}(\mathcal{E}, \text{A th} [s_1, s_2], (C_1, \ldots, C_k, B_1 \alpha_1, \ldots, B_n \alpha_n)) \in T^b_V \land
\]
\[\text{demo}(\mathcal{E}, B_1 \alpha_1), \ldots, \text{demo}(\mathcal{E}, B_n \alpha_n), \text{demo}(\mathcal{E}, \text{A th} [s_2, t_2])) \subseteq T^b_V \land
\]
\[\mathcal{D}_C \models C_1, \ldots, C_k \land \mathcal{D}_C \models s_1 \leq t_1, t_1 < s_2, s_2 < t_2\]
\[\implies\{\text{inductive hypothesis}\}\]
\[\exists \beta, \beta_1, \ldots, \beta_n : \text{clause}(\mathcal{E}, \text{A th} [s_1, s_2], (C_1, \ldots, C_k, B_1 \alpha_1, \ldots, B_n \alpha_n)) \in T^b_V \land
\]
\[\{(B_1, \beta_1), \ldots, (B_n, \beta_n), (A, \beta)\} \subseteq T^f_{\mathcal{E}} \land
\]
\[\mathcal{D}_C \models \alpha_1 \subseteq \beta_1, \ldots, \alpha_n \subseteq \beta_n, \text{th} [s_2, t_2] \sqsubseteq \beta \land
\]
\[\mathcal{D}_C \models C_1, \ldots, C_k \land \mathcal{D}_C \models s_1 \leq t_1, t_1 < s_2, s_2 < t_2.
\]

Since \(\mathcal{D}_C \models \text{th} [s_2, t_2] \sqsubseteq \beta\) then \(\beta = \text{th} [w_1, w_2]\) with \(\mathcal{D}_C \models w_1 \leq s_2, t_2 \leq w_2\). Hence we distinguish two cases according to the relation between \(w_1\) and \(s_1\).
• $D_c \models w_1 \leq s_1$.
  In this case we immediately conclude because $D_c \models \text{th}[t_1, t_2] \sqsubseteq \text{th}[w_1, w_2]$, and thus $(A, \text{th}[w_1, w_2]) \in T^c_v \land D_c \models \text{th}[l_1, l_2] \sqsubseteq \text{th}[w_1, w_2]$.

• $D_c \not\models s_1 < w_1$.
  In this case clause $(E, A \text{th}[s_1, s_2], (C_1, \ldots, C_k, B_1, \alpha_1, \ldots, B_n, \alpha_n)) \in T^c_v$, since $T^c_v = \bigcup_{i \in \mathbb{N}} T^i_v$. Moreover, from $D_c \not\models s_1 < w_1, w_1 \leq s_2, s_2 < t_2, t_2 \leq w_2$, by Lemma 7 we obtain $(A, \text{th}[s_1, w_2]) \in T^c_v T^c_x$. Since $T^c_v$ is a fixpoint of $T^c_x$ and $D_c \models s_1 \leq t_1, t_2 \leq w_2$ we can conclude $(A, \text{th}[s_1, w_2]) \in T^c_v$ and $D_c \models \text{th}[t_1, t_2] \sqsubseteq \text{th}[s_1, w_2]$.

We are finally able to prove the soundness of the meta-interpreter with respect to the least fixpoint semantics.

\[
\begin{align*}
demo(E, A \alpha) & \in T^c_v \\
\iff & \{T^c_v = \bigcup_{i \in \mathbb{N}} T^i_v\} \\
& \exists h : \demo(E, A \alpha) \in T^h_v \\
\iff & \{\text{Statement (A.3)}\} \\
& \exists \beta : (A, \beta) \in T^c_x \land D_c \models \alpha \sqsubseteq \beta \\
\iff & \{\text{definition of } F^c\} \\
& (A, \alpha) \in F^c(E).
\end{align*}
\]

\[\blacksquare\]

### A.2.2 Completeness

We first need a lemma stating that if an annotated atom $A \alpha$ is provable at the meta-level in a program expression $E$ then we can prove at the meta-level the same atom $A$ with any other “weaker” annotation (namely $A \gamma$, with $\gamma \sqsubseteq \alpha$).

**Lemma 8** Let $E$ be a program expression and $V$ be the corresponding meta-program. For any $A \in C$-base$_L$ and $\alpha \in \text{Ann}$, the following statement holds:

\[
demo(E, A \alpha) \in T^c_v \implies \{\demo(E, A \gamma) \mid \gamma \in \text{Ann}, D_c \models \gamma \sqsubseteq \alpha\} \subseteq T^c_v
\]

**Proof** We first show that for all $h$

\[
demo(E, A\alpha) \in T^h_v \implies \{\demo(E, A\gamma) \mid \gamma \in \text{Ann}, D_c \models \gamma \sqsubseteq \alpha\} \subseteq T^h_v \quad (A.4)
\]

The proof is by induction on $h$.

(Base case). Trivial since $T^0_v = \emptyset$.

(Inductive case). Assume that

\[
demo(E, A \alpha) \in T^h_v \implies \{\demo(E, A \gamma) \mid \gamma \in \text{Ann}, D_c \models \gamma \sqsubseteq \alpha\} \subseteq T^h_v,
\]

Then:
\[\text{demo}(\mathcal{E}, A \alpha) \in T_{V}^{h+1}\]
\[\iff \{\text{definition of } T_{V}^{i}\}\]
\[\text{demo}(\mathcal{E}, A \alpha) \in T_{V}(T_{V}^{h})\]

We have four cases corresponding to clauses (5.22), (5.23), (5.24) and (5.25) of the meta-interpreter.

**Clause (5.22)** \(\{\alpha = \text{th}[t_1, t_2], \text{definition of } T_{V} \text{ and clause (5.22)}\}\)
\[\{\text{clause}(\mathcal{E}, A \text{th}[s_1, s_2], (\bar{C}, \bar{B})), \text{demo}(\mathcal{E}, (\bar{C}, \bar{B}))\} \subseteq T_{V}^{h} \land\]
\[\mathcal{D}_{C} \models s_{1} \leq t_{1}, t_{2} \leq s_{2}, t_{1} \leq t_{2}\]

The annotations \(\beta\) less than \(\text{th}[t_{1}, t_{2}]\), namely such that \(\mathcal{D}_{C} \models \beta \subseteq \text{th}[t_{1}, t_{2}]\), are of two kinds:

- \(\beta = \text{th}[w_{1}, w_{2}]\) with \(\mathcal{D}_{C} \models t_{1} \leq w_{1}, w_{1} \leq w_{2}, w_{2} \leq t_{2}\). In this case we have \(\mathcal{D}_{C} \models s_{1} \leq w_{1}, w_{1} \leq w_{2}, w_{2} \leq s_{2}\) and by using definition \(T_{V}\), clause (5.22) and \(\{\text{clause}(\mathcal{E}, A \text{th}[s_{1}, s_{2}], (\bar{C}, \bar{B})), \text{demo}(\mathcal{E}, (\bar{C}, \bar{B}))\} \subseteq T_{V}^{h}\) we can conclude \(\text{demo}(\mathcal{E}, A \text{th}[w_{1}, w_{2}]) \in T_{V}(T_{V}^{h}) = T_{V}^{h+1}\).

- \(\beta = \text{in}[w_{1}, w_{2}]\) with \(\mathcal{D}_{C} \models w_{1} \leq t_{2}, t_{1} \leq w_{2}, w_{1} \leq w_{2}\). We reason as in the previous case, but using clause (5.24) and the fact that \(\mathcal{D}_{C} \models w_{1} \leq t_{2}, t_{2} \leq s_{2}, s_{1} \leq w_{2}, w_{1} \leq w_{2}\). Hence we can conclude \(\text{demo}(\mathcal{E}, A \text{in}[w_{1}, w_{2}]) \in T_{V}^{h+1}\).

**Clause (5.23)** \(\{\alpha = \text{th}[t_{1}, t_{2}], \text{definition of } T_{V} \text{ and clause (5.23)}\}\)
\[\{\text{clause}(\mathcal{E}, A \text{th}[s_{1}, s_{2}], (\bar{C}, \bar{B})), \text{demo}(\mathcal{E}, (\bar{C}, \bar{B})), \text{demo}(\mathcal{E}, A \text{th}[s_{2}, t_{2}])\} \subseteq T_{V}^{h} \land\]
\[\mathcal{D}_{C} \models s_{1} \leq t_{1}, t_{1} < s_{2}, s_{2} < t_{2}\]

The annotations \(\beta\) less than \(\text{th}[t_{1}, t_{2}]\) are of two kinds:

- \(\beta = \text{th}[w_{1}, w_{2}]\) with \(\mathcal{D}_{C} \models t_{1} \leq w_{1}, w_{1} \leq w_{2}, w_{2} \leq t_{2}\).

  We distinguish three cases according to the relation of \(s_{2}\) with \(w_{1}\) and \(w_{2}\):

  - \(\mathcal{D}_{C} \models w_{2} \leq s_{2}\).

    In this case \(\mathcal{D}_{C} \models s_{1} \leq w_{1}, w_{1} \leq w_{2}, w_{2} \leq s_{2}\). By using the definition of \(T_{V}\), clause (5.22) and the fact that \(\text{clause}(\mathcal{E}, A \text{th}[s_{1}, s_{2}], (\bar{C}, \bar{B})) \in T_{V}^{h}\) and \(\text{demo}(\mathcal{E}, (\bar{C}, \bar{B})) \in T_{V}^{h}\) we conclude \(\text{demo}(\mathcal{E}, A \text{th}[w_{1}, w_{2}]) \in T_{V}^{h+1}\).

  - \(\mathcal{D}_{C} \models s_{2} \leq w_{1}, w_{2} < s_{2}\).

    In this case \(\mathcal{D}_{C} \models \text{th}[w_{1}, w_{2}] \subseteq \text{th}[s_{2}, t_{2}]\). Furthermore, recalling that \(\text{demo}(\mathcal{E}, A \text{th}[s_{2}, t_{2}]) \in T_{V}^{h}\), by inductive hypothesis we can deduce that \(\text{demo}(\mathcal{E}, A \text{th}[w_{1}, w_{2}]) \in T_{V}^{h}\). By monotonicity of \(T_{V}\) we conclude.

  - \(\mathcal{D}_{C} \models w_{1} < s_{2}, s_{2} < w_{2}\).

    In this case \(\mathcal{D}_{C} \models s_{1} \leq w_{1}, w_{1} < s_{2}, s_{2} < w_{2}, w_{2} \leq t_{2}\). Moreover, since \(\text{demo}(\mathcal{E}, A \text{th}[s_{2}, t_{2}]) \in T_{V}^{h}\), by inductive hypothesis we deduce that \(\text{demo}(\mathcal{E}, A \text{th}[s_{2}, w_{2}]) \in T_{V}^{h}\) and by using the definition of \(T_{V}\), clause (5.23) and \(\{\text{clause}(\mathcal{E}, A \text{th}[s_{1}, s_{2}], (\bar{C}, \bar{B})), \text{demo}(\mathcal{E}, (\bar{C}, \bar{B}))\} \subseteq T_{V}^{h}\) we conclude that \(\text{demo}(\mathcal{E}, A \text{th}[w_{1}, w_{2}]) \in T_{V}^{h+1}\).
\[ \beta = \text{in}[w_1, w_2] \text{ with } D_C \models w_1 \leq t_2, t_1 \leq w_2, w_1 \leq w_2. \]

We distinguish two subcases:

- \( [w_1, w_2] \cap [s_1, s_2] \neq \emptyset. \)
  In this case \( D_C \models w_1 \leq s_2, s_1 \leq w_2, w_1 \leq w_2. \) Observing that we have \( \{ \text{clause}(E, \text{th} [s_1, s_2], (\bar{C}, \bar{B})), \text{demo}(E, (\bar{C}, \bar{B})) \} \subseteq T^h_V \), by using the definition of \( T_V \) and clause (5.24) we conclude \( \text{demo}(E, \text{A in} [w_1, w_2]) \in T^h_V + 1. \)

- \( [w_1, w_2] \cap [s_2, t_2] \neq \emptyset. \)
  In this case \( D_C \models \text{in}[w_1, w_2] \sqsubseteq \text{th}[s_2, t_2]. \) Since \( \text{demo}(E, \text{A th} [s_2, t_2]) \in T^h_V \), by inductive hypothesis we have that \( \text{demo}(E, \text{A in} [w_1, w_2]) \in T^h_V. \)

By monotonicity of \( T_V \) we conclude.

\begin{align*}
\text{(clause (5.24))} & \quad \{ \alpha = \text{in}[t_1, t_2], \text{definition of } T_V \text{ and clause (5.24)} \} \\
D_C & \models t_1 \leq s_2, s_1 \leq t_2, t_1 \leq t_2
\end{align*}

If \( D_C \models \gamma \sqsubseteq \text{in}[t_1, t_2] \) then \( \gamma = \text{in}[w_1, w_2] \) with \( D_C \models w_1 \leq t_2, t_2 \leq w_2. \) Hence we have \( D_C \models w_1 \leq s_2, s_1 \leq w_2, w_1 \leq w_2 \) and by using definition \( T_V \), clause (5.24) and the fact that \( \{ \text{clause}(E, \text{A th} [s_1, s_2], (\bar{C}, \bar{B})), \text{demo}(E, (\bar{C}, \bar{B})) \} \subseteq T^h_V \) we can conclude that \( \text{demo}(E, \text{A in} [w_1, w_2]) \in T^h_V + 1. \)

\begin{align*}
\text{(clause (5.25))} & \quad \{ \alpha = \text{in}[t_1, t_2] \}
\end{align*}

Reasoning as in the previous case, by using the definition of \( T_V \) and clause (5.25), we can conclude. \( \blacksquare \)

Now the completeness result for MuTACL meta-interpreter basically relies on two technical lemmata (Lemma 11 and Lemma 12). Roughly speaking they assert that when \( \text{th} \) and \( \text{in} \) annotated atoms are derivable from an interpretation \( I \) by using the \( T_E \) operator then we can find corresponding virtual clauses in the program expression \( E \) which permit to derive the same or greater information.

Remarkably, in the case of the \( \text{th} \)-annotated atom \( \text{A th} [t_1, t_2] \), differently from what happens in the case of the meta-interpreter for programs with annotated clauses (see Section 5.1.1), in general there is not a single virtual clause with head the atom itself, but we can find many virtual clauses whose heads “cover” the atom \( \text{A th} [t_1, t_2]. \)

Let us first introduce some preliminary notions and results.

**Definition 12 (covering)** A covering for a \( \text{th} \)-annotation \( \text{th} [t_1, t_2] \) is a sequence of annotations \( \{ \text{th} [t^i_1, t^i_2] \}_{i \in \{1, \ldots, n\}} \), such that \( D_C \models \text{th} [t_1, t_2] \sqsubseteq \text{th} [t^i_1, t^i_2] \) and for any \( i \in \{1, \ldots, n\} \)

\[ D_C \models t^i_1 \leq t^i_2, t^i_1 \leq t^i_2, t_1 < t^i_1 + 1 \]
In words, a covering of a th annotation \( \text{th} [t_1, t_2] \) is a sequence of annotations \( \{ \text{th} [t_i^1, t_i^2] \}_{i \in \{1, \ldots, n\}} \) such that each of the intervals overlaps with its successor, and the union of such intervals includes \([t_1, t_2]\). The next simple lemma observes that, given two annotations and a covering for each of them, we can always build a covering for their greatest lower bound.

**Lemma 9** Let \( \text{th} [t_1, t_2] \) and \( \text{th} [s_1, s_2] \) be annotations and \( \text{th} [w_1, w_2] = \text{th} [t_1, t_2] \cap \text{th} [s_1, s_2] \). Let \( \{ \text{th} [t_i^1, t_i^2] \}_{i \in \{1, \ldots, n\}} \) and \( \{ \text{th} [s_j^1, s_j^2] \}_{j \in \{1, \ldots, m\}} \) be coverings for \( \text{th} [t_1, t_2] \) and \( \text{th} [s_1, s_2] \), respectively. Then a covering for \( \text{th} [w_1, w_2] \) can be extracted from

\[
\{ \text{th} [t_i^1, t_i^2] \cap \text{th} [s_j^1, s_j^2] \mid i \in \{1, \ldots, n\} \land j \in \{1, \ldots, m\} \}
\]

In the hypothesis of the previous lemma \( [w_1, w_2] = [t_1, t_2] \cap [s_1, s_2] \). Thus the result of the lemma is simply a consequence of the distributivity of set-theoretical intersection with respect to union.

**Definition 13** Let \( E \) be a program expression, let \( V \) be the corresponding metaprogram and let \( I \subseteq C - \text{base}_L \times \text{Ann} \) be an interpretation. Given an annotated atom \((A, \text{th} [t_1, t_2]) \in C - \text{base}_L \times \text{Ann} \), an \((E, I)\)-set for \((A, \text{th} [t_1, t_2])\) is a set

\[
\{ \text{clause}(E, A \text{th} [t_i^1, t_i^2], (\bar{C}^i, \bar{B}^i)) \}_{i \in \{1, \ldots, n\}} \subseteq T^\omega_V
\]

such that

1. \( \{ \text{th} [t_i^1, t_i^2] \}_{i \in \{1, \ldots, n\}} \) is a covering of \( \text{th} [t_1, t_2] \), and
2. for \( i \in \{1, \ldots, n\} \), \( I \models \bar{C}^i, \bar{B}^i \)

An interpretation \( I \subseteq C - \text{base}_L \times \text{Ann} \) is called \( \text{th} \)-closed with respect to \( E \) (or \( E \)-closed, for short) if there is an \((E, I)\)-set for every annotated atom \((A, \text{th} [t_1, t_2]) \in I \).

The next lemma presents some properties of the notion of \( E \)-closedness, which essentially state that the property of being \( E \)-closed is invariant with respect to some obvious algebraic transformations of the program expression \( E \). The proofs are trivial and thus are omitted.

**Lemma 10** Let \( E, R \) and \( N \) be program expressions and let \( I \) be an interpretation. Then the following properties hold, where \( \text{op} \in \{ \cup, \cap \} \)

1. \( I \) is \((E \text{ op} E)\)-closed iff \( I \) is \( E \)-closed;
2. \( I \) is \((E \text{ op} R)\)-closed iff \( I \) is \((R \text{ op} E)\)-closed;
3. \( I \) is \(((E \text{ op} R) \text{ op} N)\)-closed iff \( I \) is \( E \) \( \text{ op} (R \text{ op} N)\)-closed;
4. if \( I \) is \( E \)-closed then \( I \) is \((E \cup R)\)-closed;
5. if \( I \) is \((E \cap R)\)-closed then \( I \) is \( E \)-closed;

6. \( I \) is \((E \cap R) \cup N\)-closed iff \( I \) is \((E \cup N) \cap (R \cup N)\)-closed.

We next prove that if we apply the \( T_E \) operator to an \( E \)-closed interpretation, then for any derived \th\-annotated atom there exists an \((E, I)\)-set (see Definition 13). This result represents a basic step towards the completeness proof. In fact, it tells us that starting from the empty interpretation, which is obviously \( E \)-closed, and iterating the \( T_E \) then we get, step after step, \th\-annotated atoms which can be also derived from the virtual clauses of the program expression at hand. For technical reasons, to make the induction work, we indeed prove a slightly stronger property.

**Lemma 11** Let \( E \) and \( Q \) be program expressions, let \( V \) be the corresponding meta-program\(^1\) and let \( I \subseteq C\)-base \( L \times \text{Ann} \) be an \((E \cup Q)\)-closed interpretation. Then for any atom \((A, \th [t_1, t_2]) \in T_E(I)\) there exists an \((E \cup Q, I)\)-set.

**Proof** The proof is done by structural induction on the program expression \( E \).

\((E \text{ is a plain program } P)\).

Let \((A, \th [t_1, t_2]) \in T_P(I)\). By definition of \( T_P \) we have two possibilities

1. \( A \th [t_1, t_2] \leftarrow \bar{C}, \bar{B} \in \text{ground}_C(P) \land I \models \bar{C}, \bar{B} \land \text{D}_C \models t_1 \leq t_2 \)

Hence, by definition of the meta-level representation of plain programs, according to clause (5.30), we have that

\[
\text{clause}(P, A \th [t_1, t_2], (\bar{C}, \bar{B})) \in T^v \]

Hence, by clause (5.27), \( \text{clause}(P \cup Q, A \th [t_1, t_2], (\bar{C}, \bar{B})) \in T^v \) and thus this single virtual clause is a \((P \cup Q, I)\)-set for \((A, \th [t_1, t_2])\).

2. \( A \th [t_1, t_2'] \leftarrow \bar{C}, \bar{B} \in \text{ground}_C(P) \land I \models \bar{C}, \bar{B} \land (A, \th [t'_1, t_2]) \in I \land \text{D}_C \models t_1 < t'_1, t'_1 \leq t_2, t_2' < t_2 \)

Again, by definition of the meta-level representation according to clause (5.30), definition of \( T^v \) and, observing that \( \text{D}_C \models t_1 \leq t_2' \) we obtain that

\[
\text{clause}(P, A \th [t_1, t_2'], (\bar{C}, \bar{B})) \in T^v
\]

and therefore, by definition of the union operator (clause (5.27) in the meta-interpreter)

\[
\text{clause}(P \cup Q, A \th [t_1, t_2'], (\bar{C}, \bar{B})) \in T^v \quad \text{(A.5)}
\]

Furthermore, since by hypothesis \( I \) is \((P \cup Q)\)-closed, there is a \((P \cup Q, I)\)-set for \((A, \th [t'_1, t_2])\)

\[
\{ \text{clause}(P \cup Q, A \th [t'_1, t_2'], (\bar{C}^i, \bar{B}^i)) \}_{i \in \{1, \ldots, n\}} \subseteq T^v \quad \text{(A.6)}
\]

\(^1\)The meta-program contains the meta-level representation of the plain programs in \( E \) and \( Q \).
Now, if $D_C \models t_1 \leq t_1$ we can conclude because (A.6) is a $(P \cup Q, I)$-set for $A \text{th} [t_1, t_2]$. If instead $D_C \models t_1 < t_1', t_1' \leq t_2'$ then it is immediate to conclude that by adding the clause in (A.5) to the $(P \cup Q)$-set for $(A, \text{th} [t_1', t_2')]$ in (A.6), we can obtain a $(P \cup Q, I)$-set for $(A, \text{th} [t_1, t_2])$.

$(E = R \cup N)$
Let $(A, \text{th} [t_1, t_2]) \in T_E (I) = T_R (I) \cup T_N (I)$. There are two possible cases:

1. $(A, \text{th} [t_1, t_2]) \in T_R (I)$.
   Since by hypothesis $I$ is $((R \cup N) \cup Q)$-closed, by point (3) of Lemma 10, $I$ is $(R \cup (N \cup Q))$-closed, and therefore, by inductive hypothesis, there exists a $(R \cup (N \cup Q), I)$-set for $(A, \text{th} [t_1, t_2])$, which is trivially a $((R \cup N) \cup Q, I)$-set.

2. $(A, \text{th} [t_1, t_2]) \in T_N (I)$.
   By points (2) and (3) of Lemma 10, the interpretation $I$ is $(N \cup (R \cup Q))$-closed. Thus, by inductive hypothesis, there is a $(N \cup (R \cup Q), I)$-set for $(A, \text{th} [t_1, t_2])$, which is obviously a $((R \cup N) \cup Q, I)$-set.

$(P = R \cap N)$
Consider any annotated atom $(A, \text{th} [t_1, t_2]) \in T_E (I) = T_R (I) \cap T_N (I)$. By definition of the operator $\cap$ there are $(A, \text{th} [s_1, s_2]) \in T_R (I)$ and $(A, \text{th} [w_1, w_2]) \in T_N (I)$ such that

$$\text{th} [t_1, t_2] = \text{th} [s_1, s_2] \cap \text{th} [w_1, w_2]$$

By hypothesis, the interpretation $I$ is $((R \cap N) \cup Q)$-closed and thus, by points (6) and (5) of Lemma 10, $I$ is also $(R \cup Q)$-closed. Therefore, by inductive hypothesis, we can find a $(R \cup Q, I)$-set for $(A, \text{th} [s_1, s_2])$:

$$\{ \text{clause}(R \cup Q, A \text{th} [s_1, s_2'], (\bar{C}_R^i, \bar{B}_R^i)) \}_{i \in \{1, \ldots, n\}}$$

and, similarly, since $I$ is also $(N \cup Q)$-closed, we can find an $(N \cup Q, I)$-set for $(A, \text{th} [w_1, w_2])$:

$$\{ \text{clause}(N \cup Q, A \text{th} [w_1, w_2'], (\bar{C}_N^i, \bar{B}_N^i)) \}_{j \in \{1, \ldots, m\}} T_v$$

By Lemma 9 there exists a covering $\{\text{th} [t_1^k, t_2^k]\}_{k \in \{1, \ldots, l\}}$ for $\text{th} [t_1, t_2]$, such that for any $k$ there are $i_k$ and $j_k$ such that

$$\text{th} [t_1^k, t_2^k] = \text{th} [s_1^{i_k}, s_2^{i_k}] \cap \text{th} [w_1^{j_k}, w_2^{j_k}]$$

We can finally conclude that the above covering gives rise to a $((R \cap N) \cup Q, I)$-set for $(A, \text{th} [t_1, t_2])$. In fact for each of the annotations $\text{th} [t_1^k, t_2^k]$ for $k \in \{1, \ldots, l\}$ there are three possibilities:
1. **clause**(\(\mathcal{R}, A \text{th} [s_1^{i_k}, s_2^{i_k}], (\vec{C}^{i_j}_{R}, \vec{B}^{i_j}_{R})\)) \(\in T_V^0\)

   **clause**(\(\mathcal{N}, A \text{th} [w_1^{i_j}, w_2^{i_j}], (\vec{C}^{i_j}_{N}, \vec{B}^{i_j}_{N})\)) \(\in T_V^0\)

   In this case, by definition of intersection operator (clause (5.29) in the meta-interpreter), we have that

   **clause**(\(\mathcal{R} \cap \mathcal{N}, A \text{th} [t_1^{i_j}, t_2^{i_j}], (\vec{C}^{i_j}_{R}, \vec{C}^{i_j}_{N}, \vec{B}^{i_j}_{R}, \vec{B}^{i_j}_{N})\)) \(\in T_V^0\)

   and therefore **clause**((\(\mathcal{R} \cap \mathcal{N}\) \(\cup\) \(\mathcal{Q}, A \text{th} [w_1^{i_j}, w_2^{i_j}], (\vec{C}^{i_j}_{R}, \vec{C}^{i_j}_{N}, \vec{B}^{i_j}_{R}, \vec{B}^{i_j}_{N})\)) \(\in T_V^0\),

   by definition of union operator (clause (5.27)). Moreover, by construction, the body of the clause is satisfied by \(I\), namely \(I \models \vec{C}^{i_j}_{R}, \vec{C}^{i_j}_{N}, \vec{B}^{i_j}_{R}, \vec{B}^{i_j}_{N}\).

2. **clause**(\(\mathcal{Q}, A \text{th} [s_1^{i_k}, s_2^{i_k}], (\vec{C}^{i_k}_{R}, \vec{B}^{i_k}_{R})\)) \(\in T_V^0\)

   In this case by definition of union operator (clause (5.28)), we have that

   **clause**(\((\mathcal{R} \cap \mathcal{N}) \cup \mathcal{Q}, A \text{th} [s_1^{i_k}, s_2^{i_k}], (\vec{C}^{i_k}_{R}, \vec{B}^{i_k}_{R})\)) \(\in T_V^0\)

   and, by construction, \(I \models \vec{C}^{i_k}_{R}, \vec{B}^{i_k}_{R}\).

3. **clause**(\(\mathcal{Q}, A \text{th} [w_1^{i_j}, w_2^{i_j}], (\vec{C}^{i_j}_{N}, \vec{B}^{i_j}_{N})\)) \(\in T_V^0\)

   As in the previous point we have that

   **clause**((\(\mathcal{R} \cap \mathcal{N}\) \(\cup\) \(\mathcal{Q}, A \text{th} [w_1^{i_j}, w_2^{i_j}], (\vec{C}^{i_j}_{N}, \vec{B}^{i_j}_{N})\)) \(\in T_V^0\)

   and, by construction, \(I \models \vec{C}^{i_j}_{N}, \vec{B}^{i_j}_{N}\).

\[\blacksquare\]

**Corollary** 2 *Let \(\mathcal{E}\) be any program expression and let \(V\) be the corresponding meta-program. Then for any \(h \in \mathbb{N}\) the interpretation \(T^h_{\mathcal{E}}\) is \(\mathcal{E}\)-closed. Therefore \(T^h_{\mathcal{E}}\) is \(\mathcal{E}\)-closed.*

**Proof** The fact that \(T^h_{\mathcal{E}}\) is \(\mathcal{E}\)-closed for each \(h \in \mathbb{N}\) is proved by induction on \(h\).

**(Base case).** Obvious, since \(T^0_{\mathcal{E}} = \emptyset\) is trivially \(\mathcal{E}\)-closed.

**(Inductive case).** Assume that \(T^h_{\mathcal{E}}\) is \(\mathcal{E}\)-closed. By Lemma 11 for any atom \((A, \text{th} [t_1, t_2])\) \(\in T^{h+1}_{\mathcal{E}} = T_{\mathcal{E}}(T^h_{\mathcal{E}})\) there exists a \((\mathcal{E}, T^h_{\mathcal{E}})\)-set. But since \(T^h_{\mathcal{E}} \subseteq T^{h+1}_{\mathcal{E}}\) we conclude that there is also a \((\mathcal{E}, T^{h+1}_{\mathcal{E}})\)-set for \((A, \text{th} [t_1, t_2])\). Therefore \(T^{h+1}_{\mathcal{E}}\) is \(\mathcal{E}\)-closed.

Since the union of \(\mathcal{E}\)-closed interpretations is clearly \(\mathcal{E}\)-closed, we immediately deduce also the final statement of the corollary. 

\[\blacksquare\]

Another technical lemma is needed for dealing with the in annotations, which comes in pair with Lemma 11.
Lemma 12 Let $\mathcal{E}$ be a program expression, let $V$ be the corresponding meta-program and let $I$ be any $\mathcal{E}$-closed interpretation. For any atom $(A, \text{in}[t_1, t_2]) \in \mathbb{T}_\mathcal{E}(I)$ we have

$$\text{clause}(\mathcal{E}, A \alpha, (\bar{C}, \bar{B})) \in T_\mathcal{E} \land I \models \bar{C}, \bar{B} \land \mathcal{D}_C \models \text{in}[t_1, t_2] \sqsubseteq \alpha$$

Proof The proof is done by induction on the structure of $\mathcal{E}$.

($\mathcal{E}$ is a plain program $P$)
Let $(A, \text{in}[t_1, t_2]) \in \mathbb{T}_P(I)$. By definition of $\mathbb{T}_P$

$$A \in \text{in}[t_1, t_2] \leftarrow \bar{C}, \bar{B} \in \text{ground}_C(P) \land I \models \bar{C}, \bar{B} \land \mathcal{D}_C \models t_1 \leq t_2$$

By the meta-level representation of plain programs, according to clause (5.30), we have

$$\text{clause}(P, A \text{in}[t_1, t_2], (\bar{C}, \bar{B})) \in T_\mathcal{E}$$

Recalling that $I \models \bar{C}, \bar{B}$, and observing that $\mathcal{D}_C \models \text{in}[t_1, t_2] \sqsubseteq \text{in}[t_1, t_2]$ we conclude.

($\mathcal{E} = \mathcal{R} \cup \mathcal{N}$)
Let $(A, \text{in}[t_1, t_2]) \in \mathbb{T}_{\mathcal{R} \cup \mathcal{N}}(I) = \mathbb{T}_\mathcal{R}(I) \cup \mathbb{T}_\mathcal{N}(I)$. Therefore $(A, \text{in}[t_1, t_2]) \in \mathbb{T}_\mathcal{R}(I)$ or $(A, \text{in}[t_1, t_2]) \in \mathbb{T}_\mathcal{N}(I)$. In the first case, by inductive hypothesis

$$\text{clause}(\mathcal{R}, A \alpha, (\bar{C}, \bar{B})) \in T_\mathcal{E} \land I \models \bar{C}, \bar{B} \land \mathcal{D}_C \models \text{in}[t_1, t_2] \sqsubseteq \alpha$$

Since, by clause (5.27) in the meta-interpreter, $\text{clause}(\mathcal{R} \cup \mathcal{N}, A \alpha, (\bar{C}, \bar{B})) \in T_\mathcal{E}$, we can conclude. If instead $(A, \text{in}[t_1, t_2]) \in \mathbb{T}_\mathcal{N}(I)$, a symmetric reasoning allows us to reach the desired conclusion.

($\mathcal{E} = \mathcal{R} \cap \mathcal{N}$)
Let $(A, \text{in}[t_1, t_2]) \in \mathbb{T}_{\mathcal{R} \cap \mathcal{N}}(I) = \mathbb{T}_\mathcal{R}(I) \cap \mathbb{T}_\mathcal{N}(I)$. By definition of the operator $\cap$ there are $(A, \beta) \in \mathbb{T}_\mathcal{R}(I)$ and $(A, \gamma) \in \mathbb{T}_\mathcal{N}(I)$ such that

$$\text{in}[t_1, t_2] = \beta \cap \gamma.$$

Several cases may arise corresponding the axioms $(\text{th} \cap')$, $(\text{th} \cap)$, $(\text{th} \cap'')$, $(\text{in} \cap)$ defining the greatest lower bound $\cap$. We next analyse only the first and the last cases, since the others can be proved analogously.

- $(\text{th} \cap')$
  In this case $\beta = \text{th}[s_1, s_2], \gamma = \text{th}[r_1, r_2], t_1 = \min\{s_2, r_2\}, t_2 = \max\{s_1, r_1\}, \mathcal{D}_C \models t_1 < t_2.$

  Since by hypothesis $I$ is $(\mathcal{R} \cap \mathcal{N})$-closed, point (5) of Lemma 10, $I$ is $\mathcal{R}$-closed and thus $(\mathcal{R} \cup \mathcal{R})$-closed, by point (1) of the same lemma. Hence, recalling that $(A, \text{th}[s_1, s_2]) \in \mathbb{T}_\mathcal{R}(I)$, we can apply Lemma 11 to deduce that there exists
an \((R \cup R, I)\)-set, or equivalently an \((R, I)\)-set for the atom \((A, \text{th} [s_1, s_2])\). In other words we can find

\[
\{\text{clause}(R, A \text{th}[s_1, s_2], (\bar{C}_R, \bar{B}_R))\}_{i \in \{1, \ldots, n\}} \subseteq T^\omega_V
\]
such that \(I \models \bar{C}_R, \bar{B}_R\) and \(\{\text{th}[s_1, s_2]\}_{i \in \{1, \ldots, n\}}\) is a covering of \(\text{th}[s_1, s_2]\).

Reasoning in a similar way on \((A \text{th} [r_1, r_2]) \in \mathbb{T}_N(I)\), we find

\[
\{\text{clause}(N, A \text{th}[r_1, r_2], (\bar{C}_N, \bar{B}_N))\}_{j \in \{1, \ldots, m\}} \subseteq T^\omega_V
\]
such that \(I \models \bar{C}_N, \bar{B}_N\) and \(\{\text{th}[r_1, r_2]\}_{j \in \{1, \ldots, m\}}\) is a covering of \(\text{th}[r_1, r_2]\).

Let \(\delta = \text{th}[s_1, s_2] \cap \text{th}[r_1, r_2]\) if \(D_C \models s_1 < r_1\) otherwise \(\delta = \text{th}[s_1, s_2] \cap \text{th}[r^m_1, r^m_2]\), then it is easy to check that \(D_C \models \text{in}[t_1, t_2] \subseteq \delta\). Furthermore, by definition of the intersection operator (clause (5.29) in the meta-interpreter) we have

\[
\text{clause}(R \cap N, A \delta, (\bar{C}_R, \bar{C}_N, \bar{B}_R, \bar{B}_N)) \in T^\omega_V
\]

and by construction \(I \models \bar{C}_R, \bar{C}_N, \bar{B}_R, \bar{B}_N\).

- (in \(\cap\))

In this case \(\beta = \text{in} [s_1, s_2]\), \(\gamma = \text{in} [r_1, r_2]\), \(t_1 = \min \{s_1, r_1\}\), \(t_2 = \max \{s_2, r_2\}\), \(D_C \models s_1 \leq s_2, r_1 \leq r_2\).

By inductive hypothesis we have that

\[
\text{clause}(R, A \alpha', (\bar{C}', \bar{B}')) \in T^\omega_V \land I \models \bar{C}', \bar{B}' \land D_C \models \text{in} [s_1, s_2] \subseteq \alpha'
\]

and

\[
\text{clause}(N, A \alpha'', (\bar{C}'', \bar{B}'')) \in T^\omega_V \land I \models \bar{C}'', \bar{B}'' \land D_C \models \text{in} [r_1, r_2] \subseteq \alpha''
\]

Let \(\delta = \alpha' \cap \alpha''\). Since \(D_C \models \text{in} [s_1, s_2] \subseteq \alpha', \text{in} [r_1, r_2] \subseteq \alpha''\), by the properties of the greatest lower bound \(D_C \models \text{in} [t_1, t_2] \subseteq \delta\). Then, by definition of the intersection operator (clause (5.29) in the meta-interpreter), we can conclude that

\[
\text{clause}(R \cap N, A \delta, (\bar{C}', \bar{C}'', \bar{B}', \bar{B}'')) \in T^\omega_V
\]

where \(I \models \bar{C}', \bar{C}'', \bar{B}', \bar{B}''\) and \(D_C \models \text{in} [t_1, t_2] \subseteq \delta\), as desired. 

\[\blacksquare\]

**Theorem 11 (Completeness)** Let \(E\) be a program expressions and \(V\) be the corresponding meta-program. For any \(A \in C\)-base\(_L\) and \(\alpha \in \text{Ann} \) the following statement holds:

\[
(A, \alpha) \in F^< (E) \implies \text{demo} (E, A \alpha) \in T^\omega_V
\]
\textbf{Proof} We first show that for all \( h \)
\[
(A, \alpha) \in T_E^h \implies \text{demo}(E, A \alpha) \in T_V^v. \tag{A.7}
\]
The proof is by induction on \( h \).
\textbf{(Base case).} Trivial since \( T_E^0 = \emptyset \).
\textbf{(Inductive case).} Assume that
\[
(A, \alpha) \in T_E^h \implies \text{demo}(E, A \alpha) \in T_V^v
\]
Observe that, under the above assumption,
\[
T_E^h \models \tilde{C}, \tilde{B} \implies \text{demo}(E, (\tilde{C}, \tilde{B})) \in T_V^v \tag{A.8}
\]
In fact let \( \tilde{C} = C_1, \ldots, C_k \) and \( \tilde{B} = B_1 \alpha_1, \ldots, B_n \alpha_n \). Then the notation \( T_E^h \models \tilde{C} \)
amounts to say that for each \( i \), \( D_C \models C_i \) and thus \( \text{demo}(E, C_i) \in T_V^v \), by definition of
\( T_V \) and clause (5.26). Furthermore \( T_E^h \models \tilde{B} \) means that for each \( i \), \( (B_i, \beta_i) \in T_E^h \) and
\( D_C \models \alpha_i \equiv \beta_i \). Hence \( \text{demo}(E, B_i \alpha_i) \in T_V^v \) and thus, by Lemma 8, \( \text{demo}(E, B_i \alpha_i) \in T_V^v \).
By several applications of clause (5.21) in the meta-interpreter we finally deduce
\( \text{demo}(E, (\tilde{C}, \tilde{B})) \in T_V^v \).

It is convenient to treat separately the cases of \text{th} and \text{in} annotations. If we assume that \( \alpha = \text{th}[t_1, t_2] \), then
\[
(A, \text{th}[t_1, t_2]) \in T_E^{h+1} \implies \text{definition of } T_E^h
\]
\[
(A, \text{th}[t_1, t_2]) \in T_E^h \implies \text{Lemma 11 and } T_E^k \text{ is } E\text{-closed by Corollary 2}
\]
\[
\text{clause}(E, A \text{th}[t_1, t_2], (\tilde{C}_i, \tilde{B}_i)) \forall i \in \{1, \ldots, n\} \subseteq T_V^v \land
\]
\[
T_E^h \models \tilde{C}_i, \tilde{B}_i \text{ for } i \in \{1, \ldots, n\} \land
\]
\[
\text{th}[t_1, t_2] \text{ covering of } \text{th}[t_1, t_2]
\]
\[
\text{definition of } T_V, \text{ clause (5.22) and } T_V^v \text{ is a fixpoint of } T_V
\]
\[
\text{demo}(E, A \text{th}[t_1, t_2]) \in T_V^v \land
\]
\[
\text{clause}(E, A \text{th}[t_1, t_2], (\tilde{C}_i, \tilde{B}_i)) \forall i \in \{1, \ldots, n-1\} \subseteq T_V^v \land
\]
\[
\text{demo}(E, (\tilde{C}_i, \tilde{B}_i)) \in T_V^v \text{ for } i \in \{1, \ldots, n-1\} \land
\]
\[
\text{th}[t_1, t_2] \text{ covering of } \text{th}[t_1, t_2]
\]
\[
\text{definition of } T_V, \text{ clause (5.23), Lemma 8 and } T_V^v \text{ is a fixpoint of } T_V
\]
\[
\text{demo}(E, A \text{th}[t_1, t_2]) \land \text{clause}(E, A \text{th}[t_1, t_2], (\tilde{C}_i, \tilde{B}_i)) \forall i \in \{1, \ldots, n-2\} \subseteq T_V^v \land
\]
\[
\text{demo}(E, (\tilde{C}_i, \tilde{B}_i)) \in T_V^v \text{ for } i \in \{1, \ldots, n-2\} \land
\]
\[
\text{th}[t_1, t_2] \text{ covering of } \text{th}[t_1, t_2]
\]
\[
\text{by exploiting several times clause (5.23) as above}
\]
\[ \text{demo} (\mathcal{E}, A \th [t_1, t_2]) \land \{ \th [t'_1, t'_2] \}_{i \in \{1, \ldots, n\}} \text{ covering of } \th [t_1, t_2] \]
\[ 
\implies 
\{ \text{by def. of covering } \mathcal{D}_c \models \th [t_1, t_2] \subseteq \th [t'_1, t'_2] \text{ and Lemma 8} \}
\]
\[ \text{demo} (\mathcal{E}, A \th [t_1, t_2]) \in T^\omega_V \]

Instead, if \( \alpha = \in \th \), then

\[ (A, \in [t_1, t_2]) \in T^\infty \]
\[ \iff 
\{ \text{definition of } T^\infty \}
\]
\[ (A, \in [t_1, t_2]) \in T \]
\[ \implies 
\{ \text{Lemma 12} \}
\]
\[ \text{clause} (\mathcal{E}, A \beta, (C, B)) \in T^\omega_V \land T^\infty \models C, B \land \mathcal{D}_c \models \in [t_1, t_2] \subseteq \beta \]
\[ \implies 
\{ \text{previous remark (A.8)} \}
\]
\[ \text{demo} (\mathcal{E}, A \beta) \in T^\omega_V \land \mathcal{D}_c \models \in [t_1, t_2] \subseteq \beta \]
\[ \implies 
\{ \text{Lemma 8} \}
\]
\[ \text{demo} (\mathcal{E}, A \in [t_1, t_2]) \in T^\omega_V \]

We now prove the completeness of the meta-interpreter of the program expressions with respect to the least fixpoint semantics.

\[ (A, \alpha) \in \mathbb{F}^C (\mathcal{E}) \]
\[ \implies 
\{ \text{definition of } \mathbb{F}^C (\mathcal{E}) \}
\]
\[ \exists \gamma \in \text{Ann} : (A, \gamma) \in T^\omega_V \land \mathcal{D}_c \models \alpha \subseteq \gamma \]
\[ \implies 
\{ \text{Lemma 8} \}
\]
\[ \text{demo} (\mathcal{E}, A \alpha) \in T^\omega_V \]

\[ \blacksquare \]
Bibliography


