Calendars as Types

Data Modeling, Constraint Reasoning, and Type Checking with Calendars

Stephanie Spranger

München 2005

Calendars as Types

Data Modeling, Constraint Reasoning, and Type Checking with Calendars

Stephanie Spranger

Dissertation
an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig–Maximilians–Universität
München

vorgelegt von
Stephanie Spranger
aus Preetz

München, 10. Oktober 2005

Erstgutachter: Prof. Dr. François Bry

Zweitgutachter: Prof. Dr. Nachum Dershowitz (Tel Aviv University, Israel)

Tag der mündlichen Prüfung: 22. November 2005

Acknowledgments

This work has been developed during my three-year scholarship from the PhD program "Logics in Computer Science" (GKLI) at the University of Munich, funded by the German Research Society (DFG). Special thanks are due to Helmut Schwichtenberg who has launched this PhD program.

Above all, I thank my supervisor François Bry, the head of the research unit "Programming and Modeling Languages" (PMS) at the Institute for Informatics at the University of Munich. Without his confidence in my skills and perseverance on the one hand side and his numerous valuable ideas on the other hand side, surely, this work wouldn't have been realized.
Nachum Dershowitz, many thanks to you for examining this thesis and for supporting me during my work. Your interest in calendars, and in particular, your book on calendric calculations has inspired me for this work.
I have to thank the team from PMS for a pleasant working atmosphere. Many colleagues from the research unit PMS, from the PhD program GKLI as well as from the REWERSE project (funded by the European Commission and by the Swiss Federal Office for Education and Science) gave me a lot of ideas. Thanks in particular to Hans Jürgen Ohlbach, Bernhard Lorenz, Klaus Schulz, Martin Hofmann, and Emmanuel Coquery. And I have to thank François Fages who gave me a hint to easily show completeness of the constraint solver, introduced in this thesis.
I thank Arnaud Lallouet for good ideas and an encouraging teamwork in Munich and Orléans and Matthias Wagner for his invitation to present this work at DoCoMo Research Labs Europe and the resulting interesting comments and remarks.
Furthermore, I thank Jutta Haußer from Japanese studies ("Japan Zentrum") of the University of Munich for an interesting interdisciplinary cooperation; I have learned several things about Japanese time and calendar culture.

I have to exceptionally thank the student Frank-André Rieß who has worked with me on this project for a long and fruitful period of time.

I want to thank several colleagues from media research ("Institut für Kommunikationswissenschaften") of the University of Munich for good "lunch-times". Finally, I thank my family and my friends who have sympathized with me for this work.

Munich, 10th October 2005   Stephanie Spranger

"Le temps passe, et progressivement tout devient vrai ce qu'on a eu de mensonger."
(Marcel Proust, 1871 – 1922, A la recherche du temps perdu, La fugitive)

Abstract

This thesis investigates real-life calendars, calendar and time expressions, and time and date formats. The thesis aims at the development of computer-based tools for modeling and processing such calendric data, primarily in the "World Wide Web", in particular regarding today's internationalization efforts and the vision of the "Semantic Web". Applications are not only appointment scheduling problems and travel planning referring to calendar expressions like "consultation hour", "meeting", or "available departure times" but also mobile applications: a mobile application listing pharmacies in the surrounding of a (mobile) user will preferably only mention those that are currently open. The temporal and calendric data used in such applications is inherently heterogeneous and context-dependent, referring to cultural, professional, legal, and/or locational aspects. Today's Web formalisms lack appropriate and specific tools for temporal and calendric data modeling and for efficiently processing such data.

To realize the goal mentioned above, I have chosen a programming language approach to time and calendars that essentially differs from logic-based and algebraic approaches. The thesis underlying this work is twofold:

1. *"Calendar as Type"*: time and calendar expressions such as "day", month", "year", "consultation hour", and "meeting" are not modeled in a logic or an algebra but, instead, by means of data types. The user is provided with a set of language constructs (so-called type constructors). Advantages of this approach are: user-friendly modeling, increase of efficiency and consistency, program and document annotation, and abstraction.

2. *"Theory Reasoning"*: Problems such as appointment scheduling or travel planning are formulated in the environment of a constraint solver specific to arbitrary calendar domains (i.e. user-defined data types like "day" or "consultation hour") rather than by axiomatization, commonly used with approaches based on ontology modeling and reasoning. The constraint solver refers to and relies on (user-defined) calendric types, it maintains the semantics of different calendric types like "day" and "consultation hour", and it allows for efficient constraint solving with arbitrary calendric data.

Concerning Computer Science, this work is related to research in the area of modeling language design and design of language constructs for programming languages based on concepts and theories from research on type systems and constraint programming. Theoretical aspects of this work are the use of data types to model time and calendar expressions with type checking approaches and constraint solving over different domains referring to such calendric types. Practical aspects of this work are user-friendly language constructs for modeling calendars and calendric data and constraints which can be efficiently processed. Such specialized language constructs and processing tools for data that refer to a specific domain such as calendars and time are (or will be) important for several Semantic Web applications. Due to use of data types and type checking approaches to temporal and calendric data, the proposed tools can be integrated into any Web language.

The thesis shows perspectives for future research on using data types and specific inference algorithms (e.g. constraint solving) for modeling and reasoning on specific theories (e.g. topologies and locational data).

Zusammenfassung

Untersuchungsgegenstand der Dissertation sind Kalender, Kalender- und Zeitbegriffe und Uhrzeit- und Datumsangaben des täglichen Lebens. Ziel ist die Entwicklung von computergestützten Werkzeugen zur Modellierung und Verarbeitung solcher Begriffe und Daten, die vor allem im „World Wide Web" und insbesondere im Zuge der Internationalisierungsbestrebungen und der Vision des sogenannten „Semantic Web" an Bedeutung gewinnen. Anwendungsbeispiele sind nicht nur Terminabsprachesysteme und Reisebuchungs- und Planungssysteme in denen Zeitbegriffe wie „Sprechstunde", „Konferenz" und „verfügbare Abflugtermine" auftauchen können, sondern auch mobile System – beispielsweise wenn eine Person eine offene Apotheke in ihrer Nähe sucht. Wesentliche Eigenschaften, der in solchen Anwendungen verwendeten Zeitbegriffe und Uhrzeit- und Datumsangaben, sind Heterogenität und Kontextabhängigkeit, wobei sich der Kontext auf kulturelle, berufliche, gesetzliche und/oder ortsgebundene Aspekte beziehen kann. Insbesondere gegenwärtigen Webformalismen fehlen umfangreiche und geeignete Zeit- und Kalendermodelle und Strategien zur Modellierung und Verarbeitung solcher Begriffe und Daten.

Zur Realisierung des in dieser Arbeit formulierten Zieles ist ein programmiersprachenbasierter Ansatz gewählt worden, der sich wesentlich von bisherigen logikbasierten und algebraischen Ansätzen zur Beschreibung und Verarbeitung von Kalender- und Zeitbegriffen unterscheidet. Die beiden nachfolgenden Thesen formulieren die Kernidee, die dieser Arbeit zugrundeliegt.

1. *„Calendar as Type"*: Kalenderbegriffe, insbesondere Zeiteinheiten, wie zum Beispiel „Tag", „Monat" und „Jahr" und Zeitbegriffe wie zum Beispiel „Sprechstunde" und „Konferenz" werden nicht logisch oder algebraisch sondern mittels Datentypen modelliert. Dazu werden dem Benutzer von der entwickelten Sprache deklarative Sprachkonstrukte (sog. Typkonstruktoren) zur Verfügung gestellt. Vorteil dieses Ansatzes sind: benutzerfreundliche Modellierung, Erhöhung von Effizienz und Konsistenz, Annotation von Dokumenten und Programmen und Datenabstraktion.

2. *„Theory Reasoning"*: Anfragen wie zum Beispiel Terminabsprachen oder eine Reiseplanung werden als Constraintproblem formuliert und mittels speziell für Kalender- und Zeitbegriffe geeignetem Constraintlösen über beliebigen endlichen Kalenderbereichen (also benutzerdefinierte Datentypen wie „Tag" oder „Sprechstunde") beantwortet und nicht wie bei webbasierten ontologischen Ansätzen üblich mittels Axiomen. Der Constraintlöser bezieht sich dabei auf (benutzerdefinierte) Kalenderdatentypen

wodurch sowohl die Sematik von Kalenderbegriffen wie „Tag" oder „Sprechstunde" erhalten als auch die Effizienz des Constraintlösers verbessert wird.

In der Informatik ist die Arbeit im Bereich des Entwurfs von Modellierungssprachen und Sprachkonstrukten für Programmiersprachen unter Verwendung von Konzepten und Theorien der Entwicklung von Typsystemen und Constraintprogrammierung einzuordnen. Theoretische Aspekte der Arbeit sind die Verwendung von Datentypen zur Modellierung (von Kalender- und Zeitbegriffen) und Lösen von Constraints über verschiedenen Wertebereichen. Praktische Aspekte der Arbeit sind benutzerfreundliche Sprachkonstrukte zur Modellierung von Kalendern und Kalender- und Zeitbegriffen und von Anfragen über Kalender- und Zeitdaten, die maschinell gut verarbeitet werden können und die für viele Anwendungen, insbesondere im „Semantic Web", hochnotwendig sind (oder sein werden). Dadurch daß Datatypen und Typprüfung verwendet werden, können die entwickelten Werkzeuge in beliebge Websprachen integriert werden.

Die Arbeit zeigt Forschungsperspektiven im Bereich der Verwendung von Datentypen und speziellen Inferenzalgorithmen (zum Beispiel Constraintlösen) zur Modellierung und Verarbeitung spezieller Theorien (zum Beispiel Topologien und Ortsangaben) auf.

Notations

notation	**description**
τ, σ, ρ	calendric type (time granularity);
	kind of points over calendric type
τ^*, σ^*, ρ^*	kind of intervals over calendric type
τ^n, σ^n, ρ^n	kind of durations over calendric type
α, β, γ	type variable
$\forall \alpha :: K$	kinding expression, read as "for every type α of kind K",
	$K := \{\tau, \tau^*, \tau^n\}$
$\sigma \preceq \tau$	aggregation
$\sigma \subseteq \tau$	inclusion
$c_{\sigma \to \tau}, c_{\tau \to \sigma}$	conversion function and its inverse
$\mathcal{C}$	finite set of aligned calendars, i.e. $\mathcal{C} := \{\tau_0, \tau_1, \ldots, \tau_n\}$
	and for all $i \in \{1..n\}$ $\tau_i \preceq \otimes \subseteq \tau_0$
e	CaTTS object language expression
te	CaTTS-DL object language expression
ce	CaTTS-CL object language expression
Γ	typing context
$\forall \alpha :: K.\, \Gamma \vdash e : \alpha$	typing statement
$trans(\forall \alpha :: K.\, \Gamma \vdash e : \alpha)$	translation of typing statement $\forall \alpha :: K.\, \Gamma \vdash e : \alpha$
$\forall \alpha, \beta :: K.\, \alpha \leq \beta$	subtype statement, $\leq := \preceq \otimes \subseteq$
$\forall \alpha, \beta :: K.\, c_\alpha^\beta$	coercion, i.e. translation of subtyping statement
	$\forall \alpha, \beta :: K.\, \alpha \leq \beta$
X, Y, Z	constraint variable
$X \in D^\tau$,	calendar domain expression for events
$D^\tau := n..m, \tau$	associated domain, $n, m \in \mathbb{Z}$, $n \leq X \leq m$
$X \in D^{\tau^*}$,	calendar domain expression for tasks
$D^{\tau^*} := n..m + d^-..d^+, \tau$	associated domain, $n, m, d^-, d^+ \in \mathbb{Z}$,
	$n \leq S_X \leq m$, $d^- \leq d_X \leq d^+$, $n + d^- - 1 \leq E_X \leq m + d^+ - 1$
$S_X := n..m$	starting point of X
$d_X := d^-..d^+$	duration of X
$E_X := n_E..m_E$	ending point of X; $n_E := n + d^- - 1$, $m_E := m + d^+ - 1$
$\mathcal{D}$	finite set of calendar domains
$\mathcal{D}_\mathcal{E}$	finite set of calendar domain expressions
C	constraint
$\mathcal{C}$	finite set of constraints
$\langle \mathcal{C}; \mathcal{D}_\mathcal{E} \rangle$	constraint problem

Naming and Subscribing Conventions

The choice of metavariable names, numeric subscriptions, and primes is guided throughout this thesis by the following principles.

1. In syntax definitions, the bare metavariable e is used for all CaTTS expressions, τ for all calendric types of kind point, τ^* for all types of kind interval, τ^n for all types of kind duration, α for all types (of any kind), etc.

2. In typing rules, the main expression (i.e. the one whose type is being calculated) is always called te in CaTTS-DL and ce in CaTTS-CL. Its subterms are named te_1, te_2, etc. (resp. ce_1, ce_2, etc.).

3. In typing rules and throughout the text, instead of $\forall\alpha :: \tau.\ \vdash e : \alpha$ (resp. $\forall\alpha :: \tau^* \vdash e : \alpha$, $\forall\alpha :: \tau^n \vdash e : \alpha$), it is written $\vdash e : \tau$ (resp. $\vdash e : \tau^*$, $\vdash e : \tau^n$), for short.

4. In typing rules, the type of an expression e is called α, written $e : \alpha$, if it does not have a specific kind; otherwise it is called τ,τ^*, or τ^n.

5. In constraint reasoning, the whole constraint problem being reduced is called $\langle\mathcal{C}; \mathcal{D}_\mathcal{E}\rangle$, and the problem it reduces to is called $\langle\mathcal{C}'; \mathcal{D}'_\mathcal{E}\rangle$.

6. In constraint reasoning, the type of a constraint variables X is called τ or τ^*, written X^τ (resp. X^{τ^*}), short for X is associated to the domain D of type τ, i.e. $X \in D^\tau$ (resp. short for X is associated to the domain D of type τ^*, i.e. $X \in D^{\tau^*}$).

7. The duration (domain) of the domain a variable X^{τ^*} is associated with is written $d_{X^{\tau^*}}$. The type of $d_{X^{\tau^*}}$ is called τ^n.

8. The index (domain) of the domain a variable X^τ is associated with is written i_{X^τ}. The type of i_{X^τ} is called $\mathbb{Z}$.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"Dreifach kommt die Zeit:
Zögernd kommt die Zukunft herangezogen,
pfeilschnell ist das Jetzt entflogen,
ewig still steht die Vergangenheit."
(Friedrich von Schiller, 1759–1805)

Time and calendars play an important role in Artificial Intelligence, in Database Systems, and, in recent times, also in the Web and Semantic Web. Temporal reasoning is a major research field in Artificial Intelligence for applications such as time tabling, scheduling, and planning; in Database Systems for applications such as query answering and change detection and reaction. Many database and information systems as well as many advanced Web and Semantic Web applications and Web services like database updates, active systems, medical monitoring, information services, appointment scheduling, travel planning, Web trading and logistics, and so-called adaptive (or context-aware) applications and systems refer to temporal and calendric data. Most existing or foreseen mobile computing applications refer not only to locations but also to time [BLOS03]. For example, a mobile application listing pharmacies in the surrounding of a user will preferably only mention those that are currently open, i.e. it refers to rather sophisticated temporal and calendric data.

The temporal and calendric data involved in such applications and systems are most often rather complex, sometimes involving different calendars with various regulations and lots of irregularities (e.g. leap years). Calendars are arbitrary human abstractions of the physical time, enabling to measure and to refer to time in different units like "day", "week", "working day", and "teaching term". Examples of calendars are cultural calendars like the Gregorian, the Julian, the Hebrew, and the old and new Chinese calendars as well as professional calendars like the academic calendar of a university or the legal calendar used in some state including legal holidays, a specification of the legal working year, due dates for taxes, etc. Hence, temporal and calendric data are not only irregular but they also depend on cultural, legal, professional, and/or locational contexts. For example the date "12/02/2005" is interpreted as 12^{th} February 2005 in France while it is interpreted as 2^{nd} December 2005 in the US. Time and calendar expressions like "month" or "teaching term" can be interpreted regarding different calendars. The specification of (religious) holidays depends on the calendar used such as "Christmas Day" which refers to 25^{th} December if the Gregorian calendar is used but to 7^{th} January if the Julian calendar is used to determine the date of Christmas on the Gregorian calendar. Several legal holidays are determined by regions such as "Epiphany" which is a legal holiday only in some German federal states. Beyond, calendric expressions such as "Friday evening" depend on some cultural interpretation: while Friday evening refers to the eve of Friday in most western countries, in some Islamic countries this expression refers to the eve of Thursday.

In fact, time and calendars seem to be fundamental issues associated with any time-dependent phenomenon in any dynamic system. These and further considerations gave birth to a large field of research in Artificial Intelligence and Database Systems that can be summarized to the effort of developing frameworks for temporal knowledge representation and reasoning. Such frameworks usually comprise a formalization of the aspects of time and calendars and means to temporal and calendric data representation and reasoning. Research in temporal knowledge representation and reasoning (in Artificial Intelligence and Database Systems) has mainly focused on set-theoretic and logic-based formalisms to time and calendars.

Recently, similar problems concerning time and calendars to those in Artificial Intelligence and/or in Database Systems appear in existing and emerging Web and Semantic Web applications and Web services. In fact, applications that involve arbitrary calendric data possibly referring to different calendars are typical for the Semantic Web: systems and applications in the Semantic Web cannot be considered being closed and they cannot demand uniform data modeling. Temporal and calendric data on the Web and the Semantic Web is extremely distributed and heterogenous. Furthermore, such data should support recent internationalization and localization efforts in the Web. Thus, existing and emerging Web and Semantic Web applications and Web services give rise to further considerations concerning time and calendar models and representation and reasoning approaches.

On the current Web, temporal and calendric data and expressions can hardly be interpreted by computers. The vision of the Semantic Web is to enrich the current Web with well-defined meaning and to enable computers to meaningfully process temporal and calendric data and expression. Nowadays research in the Semantic Web mainly focuses on *ontology-based* modeling (even of temporal and calendric data) using *generic* languages such as OWL or RDF which refer to *axiomatic reasoning* approaches designed for arbitrary Semantic Web applications.

The work reported about in this thesis claims that temporal data and calendars require *specific* modeling and processing tools, even in the Semantic Web, that goes far beyond ontology modeling and axiomatic reasoning approaches. This work is based on a programming language approach to data modeling and reasoning with calendars and calendric and temporal data. This approach combines ideas and techniques developed for modern programming languages, in particular type checking approaches including subtyping (to relate different calendars and calendric types) with theory reasoning approaches, in particular constraint programming techniques. The application of choice is appointment scheduling involving arbitrary calendars, called *multi-calendar appointment scheduling*. Thus, this approach complements both research in the area of temporal knowledge representation and reasoning and research in the area of the Semantic Web with programming language approaches to conveniently express temporal and calendric data and expressions in a user-friendly calendar modeling language and with theory reasoning approaches to efficiently model and process problems *specific* to the particular application domain of calendars and time. The work's underlying thesis is twofold:

1. *"Calendar as Type":* calendars are more conveniently expressed with dedicated language constructs. Types and type checking are as useful and desirable with calendric data types as with whatever other data types. Types complement data with machine readable and processable semantics. Type checking is a very popular and well established "lightweight formal method" to ensure program and system behavior and to enforce high-level modularity properties yielding in abstraction. Types and type checking enhance efficiency and consistency of any language.

2. *"Theory Reasoning":* calendars are more efficiently processed with dedicated reasoning methods than with "axiomatic reasoning" of ontology languages like RDF and OWL. This makes search space restrictions possible that would not be possible if calendars and temporal notions would be specified in a generic formalism such as first-order logic and processed with generic reasoning methods such as first-order logic theorem provers.

1.1 Field of Research

This thesis is related to research in the field of modeling language design, in particular modeling languages with reasoning approaches for Information Systems and Web and Semantic Web applications and Web services such as scheduling and planning systems, information services, so-called adaptive (or context-aware) applications and systems as well as mobile computing applications. Such systems and applications often involve rather complex temporal and calendric data.

The thesis introduces into a declarative modeling language for temporal and calendric data modeling and reasoning for which a programming language approach has been chosen: specific language constructs, type constructors, are provided to user-friendly modeling of calendric data as well as to express constraints on such data. This type language for calendars comes with types and type checking approaches and with efficient reasoning approaches specific to the application domain of time and calendars. This language which is called CaTTS has been developed within the last two and a half years.

The design of the time model underlying CaTTS has been influenced by research on temporal knowledge representation, in particular by an interval calculus for time and by a system of so-called time granularities to express time in different units such as days and months. Since time granularities do not have an internal structure in the sense of types, they are not immediately suitable as types in a type language for calendars. Thus, influenced by research in type theory and specification languages for theorem provers, time granularities are defined as (sub)types of other calendric types by specifying predicates, adapting the idea of *predicate types*. Interdependent with such type predicates always defining calendric subtypes of other calendric types, the operational semantics of this language is defined by a subtyping semantics of *implicit coercion* that allows for comparison of different calendric types during reasoning. Reasoning with temporal and calendric data is based on Constraint Programming techniques, in particular those developed for combinatoric problems such as scheduling.

Theoretical aspects of this work are the use of data types for modeling time and calendar expressions with type checking approaches and constraint solving over different domains referring to such types. Practical aspects of this work are user-friendly language constructs for modeling calendars and calendric data and constraints which can be processed efficiently. Such specialized language constructs and processing tools for specific data are (or will be) important for several Semantic Web applications. Due to the use of data types and type checking approaches, the proposed tools can be integrated in any Web language.

The theory of time and calendars is not the only "specific theory" that frequently appears in systems and applications such as those previously addressed. Thus, the work reported about in this thesis gives rise to future research on using data types and specific reasoning methods (e.g. constraint solving) for modeling and reasoning on specific theories (e.g. topologies and location data).

1.2 Importance of Time and Calendars for the Semantic Web

Temporal and calendric data are probably more than any other data domain a subject to user interpretation, depending on some cultural context. Interdependent with culture, time and calendars are specified in the context of professional, legal, and/or locational aspects. Thus, time and calendars should be a major concern of internationalization and localization, the Semantic Web community is striving for. In fact, many Web and Semantic Web applications and Web services refer to temporal and calendric data.

1.2.1 Cultural Concerns

In the western, Christian-dominated world, time is considered as irretrievable and unrepeatable, i.e. time progresses irresistibly. Time continuously goes on, and regarding the finiteness of human life time, time is bounded, and, thus, regarded as extremely valuable. This Christian understanding of time has led to a strict culture of counting, measuring, and dividing time, considering for example the hourly prayers of monks determined by using a sundial such as that illustrated in Figure 1.1, p.34, calendars of traders in the Middle Ages, "Ora et labora", or the Millennium Countdown in 2000. The Christian understanding of time is thus *linear*.

In Asia, however, a *circular* understanding of time predominates. It inherently differs from the western perception of a linear time: in the Asian, i.e. predominantly the Chinese cultural sphere, a year is separated into several circles, depending on the corresponding system of beliefs. In a country like Japan all these calendrical systems overlap, so it can be quite complicated and confusing to correctly refer to a day. Therefore, fixing a day for a certain occasion can be quite difficult and many calendars have to be consulted. To plan a wedding on a day, when – as it is done repeatedly during the year – the entrance of Buddha into Nirvana is commemorated, for example, is supposed to be very inauspicious.

Calendars (and clocks) are human abstractions of the physical time. They allow for measuring time in different units like "day", "week", "working day", and "teaching term", predominately depending on culture, especially religion and history. For example, the working days of a professional calendar used in some country are defined according to the days in the underlying cultural calendar in use. Examples of cultural calendars are the Gregorian, the Julian, the Hebrew, the old and new Japanese calendars, and the old and new Chinese calendars.

Although nowadays most people use the Gregorian calendar to measure time, to make appointments, or to schedule travels, more (often rather subtle) differences are present than one might think at first:

Christmas Day means 25th December in Greece, but 7th January in Russia, although, Greece and Russia are both dominated by the same (Christian Orthodox) religion. However, in Russia the Christmas Day is determined using the Julian calendar to determine the date of Christmas on the Gregorian calendar while in Greece this day is determined according to the Gregorian calendar which is used in both countries in everyday life. Furthermore, Russia, Greece, and most other Christian Orthodox countries calculate Easter according to the Julian calendar although the Gregorian calendar is used for specifying other religious celebrations. However, note that Orthodox Swedes celebrate Gregorian Easter.

Another particularity regarding (religious) holidays arises in Germany: people in Bavaria do not work on Epiphany (6^{th} January) while people in Schleswig-Holstein work on this day. The reason is that legal holidays are defined at the German federal level, not at the state level, i.e. Epiphany is a legal holiday in Bavaria but not in Schleswig-Holstein.

Yet another curiosity concerns German universities and technical institutes: while at a technical institute a lecture would be announced by a time slot like "8:15 to 9:45", at a university a lecture with the same time slot would be announced "8:00 to 10:00 c.t.". The abbreviation c.t., Latin for "cum tempore", defines a time shift by a quarter of an hour.

Beyond different interpretations of dates or holidays, the common-sense understanding of some calendar expressions varies: in Western countries like France or the US, "Friday evening" denotes the eve of Friday but in some Islamic countries, the eve of Thursday.

Years are numbered differently in Japan and China than in European countries (and not in the same way in Japan, continental China, and Taiwan). For example 24^{th} December 1926 is referred to in Japan as Taishō 15, the 12^{th} month, the 24^{th} day, while 25^{th} December 1926 is Shōwa 1, the 12^{th} month, the 25^{th} day.[1] The rationale is that, following the old Chinese practice, years are numbered in the Japanese calendar after the era name. These era names are chosen by each emperor as a kind of governing slogan. The slogan of the present Japanese emperor, whose reign started on 8^{th} January 1989, Heisei, has the meaning "creating peace" Frequently, however, the meanings of these era names being Chinese compounds are quite ambiguous and therefore often open – even if unsaid – new interpretations according to changing situations. Shōwa, the era which started 1926, can mean "brilliant Japan" as well as "radiant peace". Since in modern times it had become the habit not to change the governing slogan during one emperor's reign, this possibility to reread the meaning nevertheless permitted a new start even under the reign of still the same emperor.

On the one hand, these era names show an understanding of time which is, as can be repeatedly found in Asia, circular, inherently differing from the western understanding of a linear time.

On the other hand, these era names show that calendar systems manifest – as it is the case for the era names, as well – a great deal of inherent power. To become able to be in charge of time – even if it might be only your own – time has to be modeled. Therefore, clocks and calendars as common devices can also be seen as one step to personal independence as well as to common-sense. With the growing of a merchant class in pre-

[1]Even though for designating the months in Japanese now their mere numbers are mainly used, there exist other names that evoke the old calender. That is by mentioning the first month, for example Japanese automatically think of spring, because in the past the first month denoted the beginning of spring. This can still be seen in different greetings for the New Year, where "spring"is frequently used.

modern Japan such devices for common use became more and more on demand, as there were a sine qua non for monetary acts. But since the calendar system that days was so complicated, for example the long and short months have changed every year and have involved so much power. It was a strict state monopoly to print calendars. To circumvent this law different modes of riddle like modeling calendars in prints, so called *e-goyomi,* were invented. The length of the months and often even the year was interwoven with the overall picture of the respective prints. Even though these calendars were strictly forbidden, due to their playful approach during the 18^{th} and 19^{th} centuries private editions were soon “en vogue” as well as coveted collection articles. Two examples illustrate their working in the following.

At the first glance, Figure 1.2, p.35 illustrates only a good luck charm for New Year, showing the God of Good Fortune, Ebisu, together with a red snapper, *tai*[2]. However, the thread loosely laid around the fishing rod tells us in hardly legible letters the year, the 3^{rd} year in the era Meiwa (1766), a dog-year, while the numbers for the long months are hidden in the folds of the garment of the God of Fortune and the numbers for the short months in the outline of the fish.

Figure 1.3, p.36 is one of the popular *bijin-e,* pictures of beautiful women, of the year 1824. Distracted by the beauty the spectator hardly notices the delicate numbers headed by the character for “large” – this signifying that the following numbers are the long months – at the corpus of the shamisen (the musical instrument), the woman is holding in her hands.

1.2.2 Internationalization Efforts

Since the Web and the Semantic Web are accessible world-wide and for arbitrary purpose such as information, communication, data querying and exchange as well as applications and services like scheduling appointments and planning travels, Web languages and standards need to be usable world-wide.

The W3C Internationalization Activity (`http://www.w3.org/International`) has to ensure that W3C’s formats and protocols are usable world-wide in all (Web and Semantic Web) languages and in all writing systems. This internationalization activity should propose and coordinate the adoption by the W3C of techniques, conventions, technologies, and designs that enable and enhance the use of W3C technology and the Web world-wide, with and among the various different languages, scripts, regions, and cultures. Internationalization becomes even more important in the Semantic Web where not only formats and protocols need to be usable world-wide with *any* (Semantic) Web language but also the semantic annotations of the data as well as manipulation and processing tools for such data.

Many Web and Semantic Web applications and Web services which are involved in these internationalization efforts such as web-based information and scheduling systems

[2]Interestingly enough, this is still another of the so popular rebuses in Japanese culture: tai (the name of the fish) is seen as an abbreviation for ”omedetai”(to bring blessings upon s.o.).

and adaptive Web systems also refer to rather complex temporal and calendric data and expressions. Examples of such advanced Web systems are news servers, scheduling systems, e-commerce systems, and various Web services like flight booking, online banking, or train schedules which might be also queried. Salient to advanced Web applications is that they often explicitely or implicitly use various calendars. However, W3C internationalization and localization efforts concerning calendric and temporal data do not go beyond locale specification in Web protocols, and the use of XML Schema's predefined data types for calendric data dedicated to the Gregorian calendar [W3C01] in Web and Semantic Web applications and languages. XML Schema hardly provides any means for temporal reasoning and no multi-calendar reasoning. The presence of temporal and calendric data in many Web sites and applications and the Semantic Web requirements to meaningfully model and process such data, however, let temporal and calendric data modeling and reasoning become an important aspect of internationalization and localization efforts in the Web and the Semantic Web.

1.2.3 Applications

The so-called *"Web adaptation"* is receiving increasing attention in Web circles. Adaptation basically means delivering and/or rendering data in a context-dependent manner. One distinguishes between "semantical adaptation" adapting the data themselves, and "representational adaptation", adapting how data are rendered. For example, a web-based e-commerce catalog might adapt offers to former purchases of a user and/or render Web pages using font sizes specified by the user and/or the rendering device (desktop, cellular phone, or handheld computer) used. Temporal information on the Web mostly refers to semantical adaptation, and in modeling "contexts", temporal data often play an essential role. In the following, three scenarios of advanced Web applications that make use of calendric and temporal data are described: appointment scheduling, event planning, and budgeting.

1.2.3.1 Web-based Appointment Scheduling

Appointment scheduling is a problem faced daily by many people at work. In general, scheduling an appointment among a number of people requires considering several time constraints (such as "John does not accept a meeting on a Monday before 9:30 am") as well as the already scheduled appointments registered in the calendars of the persons concerned. An appointment scheduler tries to find a match (or a best match) between the given constraints. Advanced systems might determine priorities on appointments. Appointment scheduling often requires advanced temporal and calendric reasoning capabilites for processing planning requests such as "Mary plays tennis for one hour in the morning every second week during working time", for scheduling activities spread over two consecutive days, or for scheduling activities of different calendric types like what is the latest possible working day in February for a Valentine Day present delivered in due time (i.e. before 14th February).

It is desirable that a web-based appointment scheduler provides some form of calendar-based semantical adaptation for appointments expressed in terms of the calendar in use in the country (or countries) where one works and lives in. These calendars present more differences than one might think at first. For example Christmas Day means 7^{th} January in Russia and some (but not all) Slavic countries but 25^{th} December in Greece. Also, years are numbered differently in Japan and China than in western countries (and not in the same way in Japan, continental China, and Taiwan). Many more such examples could be given (cf. Section 1.2.1). Thus, for being usable world-wide, an appointment scheduler must refer to various calendar systems. Moreover, it should provide with adaptation to the calendar system preferred to each user making it possible to communicate with each other without having to take care of the different calendars in use. In fact, multi-calendar temporal reasoning is an essential, still rather neglected aspect of the so-called internationalization (cf. Section 1.2.2) the developers of the Web strive for. Beside various calendars and for obvious reasons, time zones and the various daylight saving times also have to be supported by modern, web-based appointment schedulers.

The following scenario illustrates the temporal notions and temporal reasoning concerns an appointment scheduler calls for.

Example 1.1 *Three business men plan a trip to London. This trip should be arranged within the next three months. They estimate the time to spend on the trip to two and a half days and specify a time interval during which the trip should take place. After having defined these temporal constraints on the agenda of each business man, the appointment scheduler queries the electronic calendars of the participants for their personal time constraints within the considered time interval. The appointment scheduler reasons over the temporal constraints and returns the (consistent!) answers to the problem, if any. In doing this, various calendars, time zones, as well as calendric expressions such as "legal holiday" and "working day" might be involved.*

An appointment as considered in Example 1.1 is a convex time interval represented by its ending points, and it has a duration. Other examples similar to 1.1 demonstrate the need for non-convex time intervals. For example the answer to the question "when are Anna, Bob, and Cindy all staying in New York" might be a non-convex interval. Furthermore, an appointment scheduler might refer to conjunctions and disjunctions of temporal constraints expressed in terms of the above-mentioned basic temporal notions. Each of these basic temporal notions can, in turn, be expressed with calendric types, i.e. units such as hour, day, week, month, trimester, and semester. Note that some of these notions might have different interpretations depending on the used calendar. For example, months are differently defined in the Islamic, Gregorian, Iranian, and Coptic calendars, for citing a few still widely used.

1.2.3.2 Web-based Event Planning

The events considered in this section are social events (like concerts or theater performances), or professional events (like venues of conventions and conferences). An event planning system is a software aiming at assisting people planning and/or managing a large number of events subject to possibly complex constraints. For example, planning the concerts of a large city might have to ensure a regular distribution over the time if those concerts aim at similar audiences. Event planning is concerned with inter-related time-dependent events. The events to be considered might be already finalized, i.e. certain, or instead potential, i.e. subject to cancellation. In contrast to appointment scheduling, event planning is in general a continuous, or at least a long lasting process: while scheduling an appointment can be seen and realized as a punctual task, scheduling events often requires long lasting, intertwined tasks. Thus, the temporal reasoning system subjacent to an event planning system must be able to manage an ongoing planning process. The following scenario illustrates the temporal notions and temporal reasoning aspects that an event planning system in general needs.

Example 1.2 *Mary is responsible for planning, managing, and surveying the cultural events of a large city. For some event, the following might have to be planned: Renting a service (e.g. a catering service) could involve calling a catering service in due time, scheduling appointments with a responsible person, conclude a contract, provide with access to premises and facilities at some point of time, oversee the service provided in due time, etc. Indeed, consistency with planned events and their sub-tasks must be checked. For example, two subcontractors cannot necessarily use the same resources, for example rooms.*

Thus, an event planning system might recall to work-flow management systems. An essential difference is that, in contrast to a standard work-flow management system, an event planning system will have to support common-sense or real life calendar expressions like “hour”, “day”, “week” or “month”. An event planning system perfectly fitting the needs of Mary from Example 1.2 will also have to support various calendars and time zones (e.g. for many artists that come from abroad). Furthermore, the city is likely to have different cultural communities the celebrations of which have to be taken into account in planning cultural events.

One probably will have to distinguish between “fully specified events” and “incompletely specified events”. The former refer to specific time and date occurrence and some event’s sub-tasks having fully specified temporal constraints. The latter refers to sub-tasks which are already committed but not yet fully scheduled. Most likely, an event planning system will have to support partially ordered activities (or tasks, or sub-tasks), incomplete information, and it will have to verify the consistency of temporal constraints between (inter-related) events.

1.2.3.3 Web-based Budgeting

A budgeting system might be seen as a temporal planning system tuned to financial control. Budgeting systems take into account both, when and in which order budgeting tasks occur. They also take into account a task’s evolution for this is often critical for a correct determination of future budgets and their related budgeting plans. Let us consider a scenario.

Example 1.3 *A budgeting system for public schools guides the school's financial analyst through the process of creating a budget that can be easily managed, consulted, balanced, and compared with previous budgets of the school, with the budgeting plan, and with the current year's budget. The budgeting system computes (and stores) the budgets of all budget sections together with some constraints. It also computes monthly reports including absolute and relative deviations from the running year's budget and extrapolation for the future based on previous year's balance-sheets. Further reports give the budget for each term of references separately as well as the currently available resources.*

Thus, a budgeting system refers to several temporal notions and constraints. Budgeting refers to different time domains because it uses both histories (past data) and extrapolations (future data).

It is worth stressing that the Web provides an infrastructure making it possible for an appointment scheduler, event planner, and budgeting system to refer to the calendars of several persons at different places and possibly moving from place to place. Web applications like those mentioned above suggest that what one might call *"multi-calendar temporal data modeling and reasoning"* is likely to become much more important in the future than it has been in the past.

1.3 Thesis' Contribution: Calendar Types and Constraints

The work reported about in this thesis is the Calendar and Time Type System CaTTS [BS04, BRS05]. CaTTS allows for declaratively modeling calendars and calendric data as types. Furthermore, CaTTS complements such data with machine readable and processable semantics based on *"theory reasoning"* approaches for efficiency reasons. CaTTS goes far beyond predefined date and time types after the Gregorian calendar. CaTTS consists of two languages, a *type definition language*, CaTTS-DL, and a *constraint language*, CaTTS-CL, of a (common) parser for both languages, and of a language processor for each language. The system CaTTS is illustrated in Figure 1.4, p.37.

Using the (type) definition language CaTTS-DL, one can specify in a rather simple manner more or less complex, cultural or professional calendars. Specificities like leap seconds, leap years, and time zones can be easily expressed in CaTTS-DL. Calendars expressed in CaTTS-DL can be composed in the sense that the language offers a means for modules. Thus, one can extend a standard calendar such as the Gregorian calendar used in Germany with a particular teaching calendar like the one of a specific German university.

Using the language fragment CaTTS-TDL (for Type Definition Language) of CaTTS-DL, one can define *calendric data types* specific to a particular calendar – such as "working day", "Easter Monday", "exam week", or "CS123 lecture" (defining the times when the Computer Science lecture number 123 takes place). Calendric types are always defined as *subtypes* of other (already defined) calendric types by user-defined *predicates*. For example, the type “exam week” can be defined from the type “week” by specifying a predicate that selects those weeks which are examination weeks.

Using the language fragment CaTTS-FDL (for F̲ormat D̲efinition L̲anguage) of CaTTS-DL, one can specify *date formats* for such data types – such as “5.10.2004”, “2004/10/05”, or “Tue Oct 5 16:39:36 CEST 2004”.

The language processor of CaTTS-DL is a *type checker* including “meta type checking” approaches for CaTTS-DL calendar specifications themselves. CaTTS’ type checker can be used for type checking of programs or specifications in *any* language (e.g. SQL, XQuery, XSLT, XML Schema, RDF, OWL), using calendric data and date formats enriched with type annotations after some calendar specified in CaTTS-DL. In particular, it is used for (static) type checking of temporal constraint programs in CaTTS-CL the constraint language of CaTTS . CaTTS-CL is type checked by *translating* a CaTTS-CL program into an equivalent program that can be solved by CaTTS-CL’s language processor. Therefore, the subtyping relation between calendric types is interpreted by *implicit coercion.* That means, whenever $\alpha \leq \beta$ (read as “type α is a subtype of type β”) is provable between calendric types, a conversion from α to β is performed. This conversion remains *implicit* in CaTTS-CL expressions. This makes CaTTS-CL programs easier to write and read. CaTTS-CL programs are evaluated by type checking them using the “high-level” typing and subtyping rules for CaTTS-CL. The conversion becomes explicit by using a “lower-level” language *without* subtyping to evaluate CaTTS-CL programs. This lower-level language is called CL_{catts}, the language used by CaTTS-CL’s language processor. Thus, CaTTS provides no evaluation rules for the high-level constraint language CaTTS-CL. Evaluation of CaTTS-CL programs is rather achieved by providing a *translation* of high-level CaTTS-CL expressions with subtyping into lower-level CL_{catts} expressions, and then using the evaluation relation of CaTTS-CL’s language processor to obtain the operational behavior of CaTTS-CL programs. This translation interprets subtyping in CaTTS-CL in terms of *conversion constraints* in CL_{catts}. In particular, the proof that α is a subtype of β (i.e. the derivation from applying the high-level subtyping and typing rules on CaTTS-CL expressions) generates a *coercion* c_α^β from α to β. This coercion is expressed in terms of CL_{catts} conversion constraints.

This thesis claims that types and type checking are as useful and desirable with calendric data types as with other data types for the following reasons.

- Types complement data with machine readable and processable semantics.
- Types enable (meaningful) annotation of data.
- Type checking is a very popular and well established “lightweight formal method” to ensure program and system behavior and to enforce high-level modularity properties.
- Type checking enhances efficiency and consistency of programming and modeling languages like Web query languages.

Specific aspects of calendars make type checking with calendars an interesting challenge: an appointment scheduler inferring an appointment for a phone conference of two persons (where one is in Munich and the other in Tel Aviv) refers not only to several time constraints formulated by the conference attendees but also to various temporal and calendric data of different types. Types give such data their intended semantics, for example that some data refer to days. Type checking ensures certain consistency on the data and constraints when processing them, for example that a week can never be during a day, i.e. this time constraint is inconsistent.

Using CaTTS' constraint language CaTTS-CL, one can express a wide range of time constraints referring to the types defined in calendar(s) specified in the definition language CaTTS-DL. For example, if one specifies in CaTTS-DL a calendar defining both, the Gregorian calendar (with types such as "Easter Monday" or "legal holiday") and the teaching calendar of a given university (with types such as "working day", "CS123 lecture", and "exam week"), then one can refer in CaTTS-CL to "days that are neither legal holidays, nor days within an examination week" and express constraints on such days such as "after Easter Monday and before June". Thus, using CaTTS-CL one can express real-life, Web, Semantic Web, and Web service related problems such as searching for train connections or making appointments (e.g. for audio or video conferences).

The language processor of CaTTS-CL is a *multi-calendar reasoner* based on *constraint solving techniques* for problems expressed in CaTTS-CL. This solver refers to and relies on the type predicates generated from a calendar definition in CaTTS-DL. The core of this solver is a *conversion constraint* that is applied whenever values of different calendric types are related in a CaTTS-CL constraint such as X *during* Y where X has type "day" and Y has type "teaching term". CaTTS-CL's conversion constraint has the following advantages.

1. It makes constraint solving with domain variables referring to different (calendric) types possible.
2. It makes search space restrictions possible that would not be possible if the calendar and temporal notion would be specified in a generic formalism such as first-order logic and processed with generic reasoning methods such as first-order logic theorem provers.

Apart from that, CaTTS-CL's constraint solver is based on an extension of the constraint system of finite domains, called *typed finite domains* including CaTTS' novel conversion constraint. The constraint reasoner solves (metric) multi-calendar appointment scheduling problems over time intervals which are expressed in CaTTS-CL.

1.4 Thesis' Outline

This thesis consists of seven chapters and three appendices. The chapters are built upon each other. Whenever necessary, references to other chapters are made explicit. The appendices are related to specific chapters: Appendix A summarizes the syntax of the type language CaTTS, including the languages CaTTS-DL and CaTTS-CL as well as some syntactic and closure restrictions made with CaTTS' prototype implementation. This appendix belongs to Chapter 4 which introduces into the language CaTTS. Appendix B belongs to Chapter 5 which specifies a constraint reasoner to solve multi-calendar temporal reasoning problems, in particular, appointment scheduling problems; its associated Appendix B summarizes a Sicstus Prolog-based CHR (Constraint Handling Rules) [FA97] implementation of this constraint solver. Finally, Appendix C contains a Haskell-based [Tho99] implementation of three algorithms implementing approaches to type checking with calendric data and constraints: first, an algorithm that tests well-formedness of CaTTS-DL calendar specifications. Second, a type checking algorithm for CaTTS-CL programs that refers to a subtype checker.[3] Finally, a transformation algorithm that transforms CaTTS-CL programs into programs of the language CL_{catts}. This transformation is based on a coercion semantics for subtyping in CaTTS-CL. CL_{catts} is used by CaTTS' multi-calendar constraint solver. This constraint solver thus defines an operational semantics for the constraint language CaTTS-CL. Appendix C belongs to Chapter 6. This chapter defines and discusses these typing and subtyping algorithms in terms of a declarative type system.

1.4.1 Introduction to the Thesis

Chapter 1 is this introduction. The introduction makes two assertions: first, calendars and calendric data are more conveniently expressed with dedicated language constructs using types and type checking approaches than in a set-theoretic or logic-based formalism such as an ontology, referred to as *"calendars as types"*. Second, calendars and calendric data are more efficiently processed with dedicated reasoning methods than with "axiomatic reasoning", referred to as *"theory reasoning"*. The necessity of a dedicated modeling and reasoning language that is specific for the application domain of time and calendars for information systems, and, in particular, for Web and Semantic Web applications and Web services is pointed out: the cultural diversity inherent to calendars and time incorporates a multitude of different time and calendar expressions and numerous varying interpretations underlying them. Dealing with such a diversity is unavoidable in a distributed and extremely heterogenous environment such as the Web and the Semantic Web. The internationalization and localization efforts of the Web community are pointing out this issue. Unfortunately, time and calendars are still rather neglected in this community. The introduction illustrates three scenarios – appointment scheduling, planning, and budgeting – involving arbitrary temporal and calendric data. Those scenarios point out that what one might call *"multi-calendar temporal data modeling and reasoning"* is likely to become much more important in the future than it has been in the past.

1.4.2 Background

Chapter 2 provides with a general background in temporal data modeling and reasoning approaches, overviewing important and well known approaches to time and calendars in the

[3]Type checking in CaTTS-CL refers to and relies on calendric types defined in CaTTS-DL.

Artificial Intelligence, the Database Systems, and the Web and Semantic Web communities. This chapter serves as an orientation guide and for further background reading in temporal data modeling and reasoning. It carefully relates the programming language approach CaTTS and its two language processors, a multi-calendar constraint reasoner and a type checker, to the considered approaches from research. Additionally, some applications of time and calendar formalisms, in particular, to Web services, to versioning and change management of Web resources, and to adaptation on the Web are addressed in Chapter 2. Sections on

- approaches to temporal knowledge representation and reasoning mainly from research in Artificial Intelligence and Database Systems,
- calendric computations (in most cases neglected in the approaches to temporal knowledge representation and reasoning), and
- Web and Semantic Web formalisms relevant for calendric data modeling and reasoning

are provided. Note that in contrast to the approaches from the Artificial Intelligence and Database Systems communities, the approaches from the Web and Semantic Web community are rather generic, i.e. not specific to the application domain of time and calendars.

Chapter 2 consists of four sections. The main aspects of each of these sections with regard to the contents are described in the following.

Approaches to Temporal Knowledge Representation and Reasoning. Section 2.1 overviews a large amount of work done in temporal knowledge representation and reasoning from research in Artificial Intelligence and Database Systems. Temporal knowledge representation refers to the symbolic representation of time (and calendars) which allows for describing temporal objects, i.e. objects which are somehow related to and/or in time, relations between such objects as well as changes of the reality modeled in some (dynamic) system. Usually, such knowledge representation systems for time (and calendars) are obtained by extending a language (in most cases a logic-based language) with means to represent temporal data. To process such data, temporal reasoning systems, usually depending on some intended application are defined for such representation systems. Those reasoners might be incorporated into a more general reasoning system.

Implicit time models focus on time-dependent entities indicating that a change in time has occurred. Examples of implicit time models surveyed in this section are the Situation Calculus [MH87, SG88, Web90, LS95, Ram00, McC02, MPP02], the Event Calculus [KS86, Kow92, Sha90, CMP93, CCM95, Sha95, SK95], and Dynamic Logic [Pra76, Har79, Har84, Moo85]. Implicit time models are particularly developed and used to model events, changes, and actions and to specify the effects of actions and to acknowledge and to react on changes in some dynamic system. Since CaTTS is rather designed to model time and calendars themselves and not to specify actions and reactions in some dynamic system, implicit time models are merely addressed for the sake of completeness.

Explicit time models consider time itself independent of anything that could happen in it. Such time models represent the flow of time by defining (at least) one time line with time primitives, in particular (time) points and/or (time) intervals. Time lines may be related either in parallel or vertically. A time model with parallel time lines leads to a hierarchical model to represent calendar units, so-called *time granularities*. Vertically related time lines provide means to define temporal histories, for example to model the transaction times of objects in a system like a data warehouse. In principle, explicit time models are either point-based [Bru72, KG77, McD82, Sho87] or interval-based [Ham72, Dow79, NS80, All83, HA89]. In the former, reasoning is performed on *points*. In the latter, reasoning is performed on *intervals*. The differences in point-based and interval-based time models refer to their applicability, their complexity concerning temporal reasoning tasks performed on points or intervals as well as their underlying common-sense understanding of time. Some proposals have been made to combine point-based and interval-based time models [Vil82, Boc90, Gal90, VS96]. Such combined time models are rather artifical, raising several problems. Some of those problems are discussed with the design of a time model underlying CaTTS in Chapter 3. Furthermore, time models for generalized intervals (i.e. intervals which are not necessarily convex) exist [LMF86, Lad87, DC91, NS92, Lig98].

Approaches to temporal reasoning in explicit time models based on constraint solving techniques refer to explicit time model based on points and/or intervals. Two principle temporal constraint formalisms exist: (1) *qualitative* temporal constraints, in particular in Allen's interval calculus [All83, vBC90, VKvB90, KL91, Mei96], and (2) *metric* temporal constraints, mainly based on disjunctive linear relations between time points [KL91, DMP91, Kou92, Mei96] to manipulate dates and durations. An example of a qualitative temporal reasoning problem is the following.

The meeting ran non-stop the whole day. The meeting began while Ms Jones was present. It finishes while Mr Smith was present who arrived after the meeting has begun. Mr White who arrived after Ms Jones has left has talked to Mr Smith. Could possibly Ms Jones and Mr Smith have talked during the meeting?

That is, qualitative temporal constraints refer to the temporal relations that hold between different objects. A constraint solver propagates new relations that can be inferred from such a problem. Thus, reasoning in qualitative constraint formalisms is performed by propagating constraints (in terms of temporal relations) that hold between different objects. In contrary, metric temporal constraints refer to dates and durations between dates. Reasoning in metric constraint formalisms is (usually) performed by propagating distances between objects to infer possible dates such objects might hold at. An example of a metric time constraint is the following.

A patient requires three medical exams, each followed within 12 hours. The exams require specific resources available from the 8th to the 12th and from the 20th to the 21st of the month.

To summarize, metric and qualitative constraints serve to model inherently different temporal reasoning problems. Furthermore, to solve metric and qualitative temporal reasoning problems, different constraint programming techniques are applied: metric constraints can be modeled by disjunctive linear relations in a DLR Horn framework, solved using *backtracking* and *bounds and/or arc consistency* techniques [Apt03] while qualitative constraints are modeled by the (transitivity property of) temporal relations themselves, solved using *path consistency* techniques [Apt03].

Various approaches to explicit time models with approaches to model different calendar units in terms of so-called *time granularities* [Mon96, Je98, BJW00, Euz01] exist. Time granularities are discretizations of a totally ordered (continuous or discrete) time line. A time granularity is a (partial) partition[4] of such a time line into a (infinite) countable set of aggregates of points (of such a time line), so-called *granules*, isomorphic to the integers. Examples of granules are "02.01.2005" of time granularity day and "the first week in the year 2005" of time granularity week. The granules of each time granularity are non-overlapping, and they preserve the order of the underlying time line. Numerous different formalizations of the concept of time granularity, various relationships between them, and several temporal operations on them have been proposed. The formalizations follow set-theoretic [MMCR92, Fre92, CEMP93, Euz93, Her94, GPP95, BCDE00, And83, CR87, WJL91, WJS95, Wan95, Sno95, BJW00, NWJ02], logic-based [Hob85, GM89, FM94, CFP04, CEMP93, Mon96], and automata-based [Wij00, LM01] approaches. Well-known approaches in each of those traditions are surveyed. Furthermore, an approach to temporal reasoning with time granularities according to [BJW00] is addressed. This approach is inherently metric and point-based. It is defined in the Horn DLR constraint framework, extended with time granularity: metric temporal reasoning problems are expressed in terms of time points and metric distances (i.e. integer intervals) between two points. A pair of variables only satisfies such a distance constraint, if (up and down) conversions between each of the points and the distance's time granularity exist. That means for example, if a point refers to weeks and another to months, then a conversion does not exist, and the constraint between those two points cannot be propagated. One could express constraints like "X and Y should fall into the same day". But it is neither possible to express a constraint like "X should be 3 days before Y" nor "X and Y should overlap with January 2005".

Calendric Computations. Section 2.2 refers to calendric computations. It addresses the main work on this aspect of time and calendars made by Reingold and Dershowitz in [DR01]. This section briefly reviews computational aspects of temporal and calendric data, often disregarded in formalisms for temporal knowledge representation such as those which

[4]in the mathematical sense of a partition of a set

are reviewed in Section 2.1. Note that although CaTTS allows for importing externally defined functions as well as that CaTTS provides with arithmetic operations for calendric computations, if required for some modeling problem, computation with calendric data is not a primary concern in CaTTS.

Web and Semantic Web Formalisms and Applications. Section 2.3 introduces the main formalisms to annotate semi-structured data using general-purpose data type declaration languages as well as ontology languages. Those formalisms are mainly developed to enrich data and information in Web pages and Web sites, making such data machine-readable and machine-processable. XML [W3C04a], a Web formalism for data and knowledge modeling is the basic principle all further languages developed for Web and Semantic Web applications and Web services directed by the World Wide Web Consortium (W3C) (cf. `http://www.w3.org`) depend on, for example XML Schema [W3C01], RDF [W3C04c], and OWL [W3C04b].

XML Schema is a language for restricting the structure of such XML documents. XML Schema extends XML not only with datatypes but also with a considerably large set of predefined time and date types dedicated to the Gregorian calendar to specify element contents, i.e. the data represented in some Web page or Web site. However, XML Schema provides with no means to specify user-defined temporal data types. Since XML Schema and similar languages fall short in issues concerning the semantics of documents needed, for example, to process the content of information, ontology languages such as OWL and RDF have been developed to overcome those problems. Ontology languages are expressive enough to formalize (at least some) time models. Some time ontologies are addressed in this section. The main drawback concerning ontologies is their computational complexity: existing reasoners for OWL such as FaCT [Hor98] are EXPTIME-complete [Tob01].

In addition to such Web formalisms, this section addresses the importance of internationalization and localization in the Web and Semantic Web since temporal and calendric data are an important matter of internationalization and localization due to their cultural, legal, business, and/or locational contexts. Beyond, Web and Semantic Web applications concerning time and calendars, in particular, Web services and temporal and active Web systems are (briefly) reviewed.

In Comparison with CaTTS. Section 2.4 compares CaTTS with the reviewed approaches to temporal knowledge representation and reasoning, calendric calculations, and the addressed Web and Semantic Web formalisms. The main statements of this comparison are summarized in the following.

CaTTS' underlying explicit time model is linear, interval-based and discrete. The time model incorporates the concept of time granularities. It is a simple, however, expressive time model that carefully selects and defines a minimum set of operations between time granularities. Furthermore, the time model provides a means to time granularity conversion between *any* pair of time granularities based on the definition of a slightly weaker *join* than an ordinary lattice join.

CaTTS' multi-calendar constraint reasoner operates on metric interval constraints. Rather than in a DLR Horn framework (as it is commonly used in metric temporal reasoning), CaTTS' constraints are formalized in the constraint system Finite Domains (FD) [FA97]. Variables represent time intervals with time granularity. Several temporal relations, in particular, the basic interval relations [All83] and several metric relations may hold between those variables. Variables in any time granularity can be compared. The reasoner allows for constraints such as "a visit of 5 working days must overlap with the meeting two working days". Thus, CaTTS' reasoner goes beyond proposals to temporal reasoning with time granularity made in the literature [BJW00]. Furthermore, in contrast to [BJW00], CaTTS' reasoner provides with a complete solution to the problem of time granularity conversion (addressed in [FM01]).

CaTTS departs from both set-theoretic and logic-based time models with time granularities as well as from time ontologies for the following reasons: CaTTS considerably simplifies the modeling of specificities of cultural calendars (such as leap years, sun-based cycles like Gregorian years, or lunar-based cycles like Hebrew months) as well as the modeling of professional calendars often involving "gaps" in time (e.g. "working-day"), "gapped" data items (e.g. "working-week"), and periodic events (e.g. "CS123-lecture"). The well-known advantages of statically typed languages such as error detection, language safety, efficiency, abstraction, and documentation whereas the latter two obtain particular interest due to overloaded semantics of calendric data apply to CaTTS, as well. Beyond this, CaTTS' static type checker provides both meta-type checking of predicate subtype definitions in CaTTS-DL and type checking of constraints in CaTTS-CL, obtaining the semantics of different time granularities even for reasoning with their granules.

In contrast to time ontologies, CaTTS comes along with a constraint solver dedicated to calendar definitions in CaTTS-DL; this dedication makes considerable search space restrictions, hence gains in efficiency, possible. While (time) ontologies follow the (automated reasoning) approach of "axiomatic reasoning", CaTTS is based on a (specific) form of "theory reasoning", an approach well-known through paramodulation. Like paramodulation ensures efficient processing of equality in resolution theorem proving, CaTTS provides the user with convenient constructs for calendric types and efficient processing of data and constraints over those types.

CaTTS complements data type definition languages and data modeling and reasoning methods for the Semantic Web such as XML Schema [W3C01], RDF [W3C04c], and OWL [W3C04b]: XML Schema provides a considerably large set of predefined time and date data types dedicated to the Gregorian calendar whereas CaTTS enables user-defined data types dedicated to any calendar. RDF and OWL are designed for *generic* Semantic Web applications. In contrast, CaTTS provides with methods *specific* to particular application domains, that of calendars and time.

1.4.3 A Time Model for Calendric Data, Types, and Constraints

Chapter 3 introduces the formal time model underlying the calendar type language CaTTS.

CaTTS is based on a *discrete*, *linear* and *interval-based* time model that incorporates the concept of *time granularities* [Mon96, Je98, BJW00, Euz01].

CaTTS is not intended for expressing possible futures, hence it is not based on a "branching time". Most common-sense applications can be conveniently modeled in a linear time framework. Temporal data of every kind have a duration. This reflects a widespread common-sense understanding of time according to which one mostly refer to time intervals, not to time points. Considering only time intervals and no time points significantly simplifies data modeling and data processing. However, CaTTS can deal with time point-like data like the beginning of a week or whether a day d falls into a week w or not, as well.

Discreteness results from the way time granularities are defined: a time granularity is a mapping from the integers into subsets of a totally ordered (continuous) time line isomorphic to the real numbers. The elements of such a time granularity are called *granules*. Examples of time granularities are days and weeks with granules such as "20.05.2005" and "the last week in summer". Note that by definition, each granule of any time granularity has a duration. CaTTS does not only allow for referring to granules, but also to finite sequences of granules, i.e. to (time) intervals in a time granularity.

CaTTS provides with language constructs, in particular, predicate types to define time granularities as subtypes of other already defined types (time granularities). Those subtypes are defined in terms of either *inclusion* or *aggregation* of time granularities. For example, a type "working-day" is an inclusion (in the common set-theoretic sense) of the type "day" since the set of working days is a subset of the set of days; the type "week" is an aggregation (in constructive set-theory) of the type "day" since each week can be defined as a time interval of days. Those two relations are sufficient to define most calendar units and calendars in use today as the examples in this thesis show.

A solution to the problem of time granularity conversion [FM01] is provided: a *calendar* (or an aligned set of calendars) is defined by a finite set of partially ordered types (time granularities) according to a composition of the aggregation and inclusion relations, referred to as *subtype relation*, just expressing the fact that a time granularity is always either (somehow) coarser or (somehow) finer than another time granularity. A *conversion* between time granularities ordered in this subtype relation is defined. Built upon this subtype relation (and its conversion), a concept of a slightly weaker *join* than the ordinary lattice join of time granularities is defined. This join ensures time granularity conversion between pairs of time granularities in a calendar which are not ordered in the subtype relation: the join of the two types (time granularities) is computed and both are converted into the join (type). For example, types (time granularities) "week" and "month" are not related with respect to the subtype relation. The join of "week" and "month" is "day" (because "week" and "month" are both subtypes of "day"). Values of both "week" and "month" can be then unambiguously represented in terms of days.

1.4.4 The Language CaTTS

Chapter 4 introduces the calendar type language CaTTS. CaTTS is a declarative modeling language to specify calendars and calendric data and constraints. It provides with a (rather small) set of intuitive, thus, user-friendly language constructs to define time granularities as types. Thanks to types and type checking approaches, such calendar specifications can be maintained and reused. Furthermore, the semantics of the different calendric types is maintained even for reasoning with calendric data and constraints. This chapter comes with three sections. The main aspects of each of the sections are illustrated by example in the following.

CaTTS-DL: Definition Language. Section 4.1 introduces into the type definition language CaTTS-DL to specify in a rather simple manner more or less complex, cultural and/or professional calendars. An example calendar that can be modeled in CaTTS-DL is given in the following.

```
calendar JapaneseGregorian =
 cal
   type second;
   type minute = aggregate 60 second @ second(1);
   type hour = aggregate 60 minute @ minute(1);
   type day = aggregate 24 hour @ hour(1);
   type week = aggregate 7 day @ day(1);
   type month = aggregate
     31 day named january,
     alternate month(i)
      | (i div 12) mod 4 == 0 &&
        ((i div 12) mod 400 != 100 &&
         (i div 12) mod 400 != 200 &&
         (i div 12) mod 400 != 300)     -> 29 day
      | otherwise                       -> 28 day
     end named february,
     31 day named march,
     30 day named april,
     31 day named may,
     30 day named june,
     31 day named july,
     31 day named august,
     30 day named september,
     31 day named october,
     30 day named november,
     31 day named december
    @ day(1);
   type year = aggregate 12 month @ month(1);
```

```
    group holiday = with select day(i) where P
(* onset of spring: *)
      type setsubun for P = relative i in february >= 3 &&
                            relative i in february <= 4
(* emperor's birthday: *)
      type showa for P = relative i in select april(i) where
         april(i) after "1926" && april(i) before "1989"   == 29
      type heisei for P = relative i in select december(i) where
         december(i) after "1988" == 23
   end

   group festival = with select day(i) where P
(* star festival: *)
      type tanabata for P = relative i in july == 7
(* children festival: *)
      type shichigosan for P = relative i in november == 15
   end
 end
```

A CaTTS calendar specification begins with the keyword `calendar` and ends with the keyword `end`. The above calendar specification binds a calendar to the identifier `JapaneseGregorian` (i.e. the name of the calendar that can be referred to). This CaTTS-DL calendar specification consists of a set of calendric type definitions (each identified by the keyword `type` followed by some user-defined name) and two group definitions (identified by the keyword `group` followed by some user-defined name). Groups can be used to collect a set of type definitions having some intended relationship. The first type defined is `second`. It has no further properties. The CaTTS system interprets such a type definition simply as an indexed set isomorphic to the integers. If the programmer wants to relate his/her type definition for seconds to a real flow of time, he/she can use CaTTS' predefined type `reference`. `reference` is the calendar unit "second" of Unix (UTC seconds with midnight at the onset of Thursday, January 1 of year 1970 (Gregorian) as fixed point indexed by 1). The type definition of the type `minute` is derived from that of the type `second`. The CaTTS language processor interprets this type definition as an *aggregation subtype* of the type `second` such that each of its elements comprises 60 seconds[5] (denoted `aggregate 60 second`) and that the minute that has number 1, i.e. `minute(1)` comprises all seconds between `second(1)` and `second(60)` (`minute(2)` those between `second(61)` and `second(120)`, etc. (denoted `@ second(1)`). Such a type definition allows the CaTTS language processor to compute the seconds contained in a specific minute, i.e. to *convert* values from one type to those of another. `aggregate 60 second @ second(1)` is called the type's *predicate*. The conversion between minutes and seconds is generated from such a predicate. Any further type definition follows the same pattern. The definitions are

[5]In CaTTS-DL, it is possible to define a type `minute` that considers leap seconds, as well (cf. Chapter 4).

straightforward following the rules of the Gregorian calendar [DR01]. Since Gregorian months have different lengths, a CaTTS type `month` is defined with a repeating pattern of the twelve months. The month February which is one day longer in each Gregorian leap year is defined by an additional pattern which specifies the leap year rule for the Gregorian calendar using the CaTTS language construct `alternate...end`. The above calendar additionally specifies the groups `holiday` and `festival.` The later defines some important Japanese festivals, in particular Tanabata (the star festival) and Shichigosan.[6] For the type definition in these two groups, CaTTS relative construct is used. The particularities are considered in the following: since onset of spring is celebrated on February 3 and 4, two relative expressions are conjoined (denoted `relative i in february >= 3 && relative i in february <= 4`). Another public holiday is the birthday of the emperor (tennō no tanjōbi). However, this notion naturally changes with the emperors. From 1926 until 1989, when the Shōwa tennō passed away, the 29^{th} April was celebrated as the emperor's birthday[7], but from then on changed to 23^{rd} December, the birthday of the new Heisei tennō. These birthdays are defined by finite types selecting those 29^{th} Aprils between 1926 an 1989 (denoted `relative i in select april(i) where april(i) after "1926"`&& `april(i) before "1989"== 29`) for the era named Shōwa, and selecting those 23^{rd} Decembers which are after 1988 for the era named Heisei (denoted `relative i in select december(i) where december(i) after "1988"== 23`). Note that `select december(i) where december(i) after "1988"` is a *local* CaTTS-DL type definition without binding an identifier to the type.

Note that as in the different Western calendars, there are also holidays with variable dates in the Japanese calendar, for example the *seijin no hi* (the day of Coming of Age). All young people who turn twenty years old in that year are celebrated always on the second Monday in January. Such holidays can be expressed in CaTTS-DL, as well. Additional aspects of the Japanese calendar are discussed in [BHRS05]. Further note that particularities like time zones can be easily expressed in a CaTTS-DL calendar specification. Calendar specifications of other cultural calendars in CaTTS-DL, in particular the Islamic and Hebrew calendars are given in [BRS05, BS04].

Each of the CaTTS-DL language constructs of which some have been used in the example given above, called *(predicate) type constructors* (e.g. `aggregate 7 day @ day(1)` and `select december(i) where december(i) after "1988"== 23`), defining some calendric type as a (inclusion or aggregation) subtype of another calendric type are introduced and exemplified in Section 4.1. CaTTS-DL's means to specify date formats for the values of such types are illustrated by example, as well. The module concepts, *calendar*, *calendar signature* (i.e. the "type" of a calendar itself), and *calendar function* are defined and exemplified.

[6] "Shichi Go San" means "Seven Five Three". On Shichigosan girls of age three and seven and boys of age three and five are celebrated and it is prayed for their good health and growth.

[7] Note that each old emperor's birthday continues to be a public holiday under a different name. For example, Shōwa tennō's birthday in today's calendar is referred to as the Green Day (midori no hi).

CaTTS-CL: Constraint Language. Section 4.2 introduces into the constraint language CaTTS-CL. Using CaTTS-CL one can express a wide range of temporal constraints over activities. Such activities are typed after calendar(s) specified in the definition language CaTTS-DL. An example multi-calendar appointment scheduling problem that can be expressed in CaTTS-CL is given in the following: assume that someone plans a trip (of 10 days) to Japan within July 2005 such that it contains the Tanabata festival. Furthermore, the trip must overlap 2 days with a one week meeting that is sometime after 10^{th} July 2005. In CaTTS-CL, this problem is expressed as follows, assuming that the used formats are specified in some CaTTS-FDL catalog:

```
program JapanTrip
 prog
   use_calendar JapanGregorian;

   Trip is 10 day && Trip within "07.2007" &&
   X is 1 tanabata && Trip contains X &&
   Meeting is 1 week && Meeting after "10.07.2005" &&
   (shift (begin_of Meeting) forward 2 day) equals (end_of Trip)
 end
```

The program identifier `JapanTrip` is bound to the CaTTS-CL program within the keywords `prog` and `end`. The program refers to the CaTTS-DL calendar specification `JapaneseGregorian` that is given above. The calendar is referred to by the `use_calendar` construct. The constraint `Trip is 10 day` associates the variable `Trip` with a domain of intervals of 10 days. The condition that the trip should be within July 2005 and that it must contain the Tanabata festival are expressed by the constraints `Trip within "07.2007"` and `Trip contains X` where `X` is a variable that is associated with a domain of Tanabatas. && denotes a conjunction of constraints. The variable `Meeting` is associated with a domain of weeks by the constraint `Meeting is 1 week`. The constraint `Meeting after "10.07.2005"` expresses that the Meeting must be some time after 10^{th} July 2005. Finally, the constraint `(shift (begin_of Meeting) backward 2 day) equals end_of Trip` expresses the condition that the trip must overlap with the meeting for 2 days as follows: the day that results from shifting the starting time of the meeting forward 2 days in time must equal the day that ends the trip to Japan. If this constraint holds, the trip overlaps with the meeting for 2 days.

The different calendar constraints of which some are used in the example given above, in particular, *calendar domain constraints* (e.g. `Trip is 10 day`) and *time constraints* (e.g. `Trip contains X`) that can be used in CaTTS-CL along with the notions of *answers* and *solutions* to multi-calendar appointment scheduling problems that can be expressed in a CaTTS-CL *program* are defined and exemplified.

Section 4.3 illustrates a (larger) example, demonstrating CaTTS' means to model different cultural and professional calendars as well as a multi-calendar appointment scheduling problem.

The complete syntax of CaTTS, separately defined for CaTTS-DL and for CaTTS-CL is given in this chapter's associated Appendix A. Appendix A additionally summarizes syntactic restrictions enforced for CaTTS' prototype implementation. Note that CaTTS' current prototype implements a (rather small) subset of CaTTS. The prototype is merely for proof of concepts, demonstrating the basic principles of the language.

1.4.5 Constraint Reasoning with Calendric Data

Chapter 5 introduces the language processor of CaTTS-CL, a multi-calendar temporal reasoner. This reasoner is defined and implemented using constraint-solving techniques. In the context of CaTTS, multi-calendar temporal reasoning addresses the problem of finding free time slots for related *activities*, i.e. objects with a temporal extend such as a meeting in the presence of various (temporal) conditions. Such problems are called *appointment scheduling problems*. In CaTTS, activities may refer to arbitrary (cultural and/or professional) calendars. In particular, CaTTS' multi-calendar temporal reasoner can solve appointment scheduling problems expressed in CaTTS-CL. On the implementation side of CaTTS' reasoner (which is the contents of Chapter 5), multi-calendar appointment scheduling problems can be conveniently modeled in the constraint system Finite Domains (FD) [FA97, MS98]. In such constraint systems, variables range over (finite) domains. Those domains are usually taken from a number set like integers or reals. In CaTTS, such domains are used to represent finite, convex (time) intervals. Those intervals are represented by two associated finite domain constraints, one represents the possible starting times of the interval and the other the interval's possible durations. It is specific to CaTTS' approach to solve multi-calendar temporal reasoning problems, and, in particular, appointment scheduling problems that can be expressed in a finite domain framework that those problems involve different calendric types. That is, each domain that represents an interval has a calendric type attached. Therefore, the constraint system FD is extended to a typed version of it, called *typedFD*. Furthermore, CaTTS' provides with a novel finite domain constraint, called *conversion constraint*. This constraint defines a means to relate domains represent in different calendric types to each other. The conversion constraint relies on the conversion functions generated from the types' predicates defined in some CaTTS-DL calendar specification. The conversion constraint is the core of CaTTS' constraint solver. It allows for efficient (in fact linear in the number of the constraints and variables w.r.t. the domains' sizes) constraint solving over different domains while maintaining the semantics of the different calendric types. Furthermore, CaTTS' multi-calendar constraint solver over the constraint system typedFD including the novel conversion constraint is proved to be complete.

Chapter 5 consists of six sections: Section 5.1 recalls basics in Constraint (Logic) Programming. Section 5.2 illustrates the working of CaTTS' constraint solver on a multi-calendar appointment scheduling problem. The constraint system within which this solver is defined and implemented is given in Section 5.3. The various constraints of the solver are introduced in Section 5.4. Section 5.5 defines the constraint solver by a finite set of inference rules (at least) one for each of the supported constraints. Finally, Section 5.6 analyzes the complexity of the algorithm. Appendix B is associated with Chapter 5: this appendix summarizes the Sicstus Prolog-based CHR (Constraint Handling Rules) [FA97] implementation of CaTTS' multi-calendar constraint solver. In this appendix, completeness of the multi-calendar constraint solver is shown.

The working of CaTTS' multi-calendar constraint solver is illustrated by example in the following.

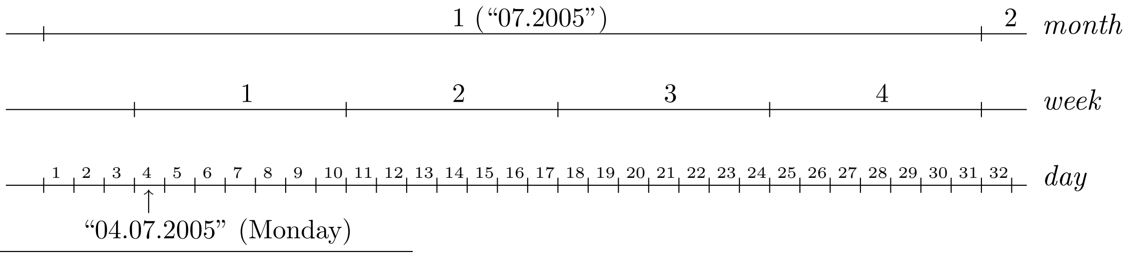
Multi-Calendar Constraint Reasoning in CaTTS. CaTTS' constraint solver relies on and refers to the type predicates generated from a calendar definition in CaTTS-DL. Those type predicates define functions to convert values of one calendric type to corresponding values in another calendric type. Those conversion functions generated from the predicates of the calendric types are accessed by the core of CaTTS' constraint solver, its conversion constraint. The conversion constraint provides a means to relate a set of values represented in one calendric type to an equivalent set of values represented in another calendric type. This relationship is maintained during constraint solving of a multi-calendar appointment scheduling problem expressed in CaTTS-CL. The conversion constraint is necessary to solve a time constraint like `X during Y` if `X` and `Y` refer to different calendric types.

Let us consider an example: assume the following CaTTS-CL program with referred calendric types and data formats defined in CaTTS-DL:

```
...
Trip is 2 week && Trip during "07.2005"
...
```

This program formulates the constraint, that a trip of 2 weeks must be during July 2005.

To solve this problem using CaTTS' constraint solver, the program is transformed into an equivalent program that contains conversion constraints for variables not expressed in the same calendric type. The type within which the variables[8] need to be represented as well as the transformation into constraints that can be solved by CaTTS' constraint solver are inferred by CaTTS' type checker (cf. Chapter 6). This should not concern us any further in order to understand the working of CaTTS' constraint solver in the following. We simply assume that this transformation has already been done. For a better understanding of CaTTS' constraint solver which operates on the integer indices of the values of the different calendric types, the following indexing for types `day`, `week`, and `month` is assumed.



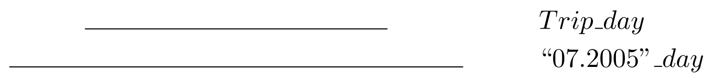
[8]Since `Trip` has type interval of `week` and `"07.2005"` has type `month`, a good candidate for a common representation is the type interval of `day`.

The variable `Trip` is represented in terms of weeks and in terms of days and the constant `"07.2005"` is represented in terms of months and days as follows according to the illustration given above: the variable `Trip` and its associated domains in terms of weeks, written $Trip :: -\infty..\infty + 2..2, week$ and in terms of days, written $Trip\_day :: -\infty..\infty + 14..14, day$. The *calendar domain constraint* $Trip :: -\infty..\infty + 2..2, week$ is read as "the variable `Trip` is associated to the domain $-\infty..\infty + 2..2, week$ that represents intervals of 2 weeks where the possible starting time of this interval is any integer (denoted $-\infty..\infty$) and the possible duration is in minimum and in maximum 2 (denoted 2..2)". Intervals that are represented by this domain constraint are for example the interval from (week) 1 to (week) 2 and the interval from (week) 2 to (week) 3. `Trip_day` is represented analogously. Since the duration of one week in terms of days is 7, the duration of two weeks in terms of days is 14. The interval from (day) 11 to (day) 24 is represented by this domain, for example. (This interval corresponds to the (equivalent) interval from (week) 2 to (week) 3). Similarly, the constant `"07.2005"` and its associated domains in terms of months, written "07.2005":: $1..1+1..1, month$ and in terms of days, written "07.2005"_day:: $1..1+31..31, day$ is represented by calendar domain constraints. In addition to those four calendar domain constraints, the constraint solver contains the time constraints, *Trip_day during "07.2005"_day*, i.e. the 2-week trip must be during July 2005, both expressed in terms of days. To ensure that for the trip only "real" weeks are considered and for the month July only this month, the constraint solver additionally contains the following conversion constraints: $Trip\_day \simeq Trip$ and "07.2005"_day $\simeq$ "07.2005". The conversion constraint $Trip\_day \simeq Trip$ relates the trip represented in weeks to an equivalent amount of time that is represented in days and analogously the conversion constraint "07.2005"_day $\simeq$ "07.2005".

To summarize, the constraint problem given above in CaTTS-CL has the following (internal) representation:

$Trip :: -\infty..\infty + 2..2, week$, $Trip\_day :: -\infty..\infty + 14..14, day$,
"07.2005" :: $1..1 + 1..1, month$, "07.2005"*_day* :: $1..1 + 31..31, day$;
Trip_day during "07.2005"_day, $Trip\_day \simeq Trip$, "07.2005"*_day* $\simeq$ "07.2005"

Constraint propagation that is performed by CaTTS' constraint solver, i.e. the application of a constraint like *Trip_day during "07.2005"_day* is all about reducing the domains associated to the variables *Trip_day* and *"07.2005"_day*. The constraint *during* states the following:



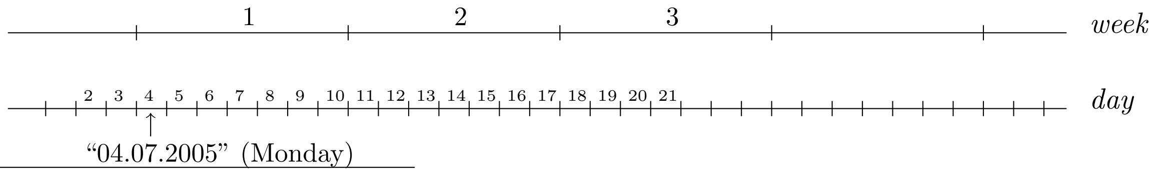
In words, July 2005 must start before the trip starts and it must end after the trip ends. Now, we apply this constraint on our problem:

$$\begin{array}{ll}
 & \mathit{Trip\_day}\ \mathit{during}\ \text{“}\mathit{07.2005}\text{”}\mathit{\_day},\ \mathit{Trip\_day} \simeq \mathit{Trip},\ \text{“07.2005”}\mathit{\_day} \simeq \text{“07.2005”};\\
 & \mathit{Trip} :: -\infty..\infty + 2..2, \mathit{week},\ \mathit{Trip\_day} :: -\infty..\infty + 14..14, \mathit{day},\\
 & \text{“07.2005”} :: 1..1 + 1..1, \mathit{month},\ \text{“07.2005”}\mathit{\_day} :: 1..1 + 31..31, \mathit{day}\\
 & \\
\mapsto_{\mathit{during}} & \mathit{Trip\_day}\ \mathit{during}\ \text{“}\mathit{07.2005}\text{”}\mathit{\_day},\ \mathit{Trip\_day} \simeq \mathit{Trip},\ \text{“07.2005”}\mathit{\_day} \simeq \text{“07.2005”};\\
 & \mathit{Trip} :: -\infty..\infty + 2..2, \mathit{week},\ \mathit{Trip\_day} :: 2..17 + 14..14, \mathit{day},\\
 & \text{“07.2005”} :: 1..1 + 1..1, \mathit{month},\ \text{“07.2005”}\mathit{\_day} :: 1..1 + 31..31, \mathit{day}
\end{array}$$

What has happened: the time constraint $\mathit{Trip\_day}\ \mathit{during}\ \text{“}\mathit{07.2005}\text{”}\mathit{\_day}$ demands that the trip starts after July 2005 starts. Thus, the minimal possible starting time of the trip is (day) 2 as denoted in the constraint $\mathit{Trip\_day} :: 2..17 + 14..14, \mathit{day}$. Furthermore, the time constraint $\mathit{Trip\_day}\ \mathit{during}\ \text{“}\mathit{07.2005}\text{”}\mathit{\_day}$ demands that the trip ends before July 2005 ends. Thus, its maximal ending time of trip is (day) 30. The duration of $\mathit{Trip\_day}$ is 14 (days). Since the maximal ending time is (day) 30 and the maximal duration is 14 (days), the maximal starting time of $\mathit{Trip\_day}$ is (day) 17 (i.e. $30 - 14 + 1 = 17$)[9]. Now the conversion constraint $\mathit{Trip\_day} \simeq \mathit{Trip}$ is applied since Trip and $\mathit{Trip\_day}$ no longer represent the same amount of time:

$$\begin{array}{ll}
 & \mathit{Trip\_day}\ \mathit{during}\ \text{“}\mathit{07.2005}\text{”}\mathit{\_day},\ \mathit{Trip\_day} \simeq \mathit{Trip},\ \text{“07.2005”}\mathit{\_day} \simeq \text{“07.2005”};\\
 & \mathit{Trip} :: -\infty..\infty + 2..2, \mathit{week},\ \mathit{Trip\_day} :: -\infty..\infty + 14..14, \mathit{day},\\
 & \text{“07.2005”} :: 1..1 + 1..1, \mathit{month},\ \text{“07.2005”}\mathit{\_day} :: 1..1 + 31..31, \mathit{day}\\
 & \\
\mapsto_{\mathit{during}} & \mathit{Trip\_day}\ \mathit{during}\ \text{“}\mathit{07.2005}\text{”}\mathit{\_day},\ \mathit{Trip\_day} \simeq \mathit{Trip},\ \text{“07.2005”}\mathit{\_day} \simeq \text{“07.2005”};\\
 & \mathit{Trip} :: -\infty..\infty + 2..2, \mathit{week},\ \mathit{Trip\_day} :: 2..17 + 14..14, \mathit{day},\\
 & \text{“07.2005”} :: 1..1 + 1..1, \mathit{month},\ \text{“07.2005”}\mathit{\_day} :: 1..1 + 31..31, \mathit{day}\\
 & \\
\mapsto_{\mathit{Trip\_day} \simeq \mathit{Trip}} & \mathit{Trip\_day}\ \mathit{during}\ \text{“}\mathit{07.2005}\text{”}\mathit{\_day},\ \mathit{Trip\_day} \simeq \mathit{Trip},\ \text{“07.2005”}\mathit{\_day} \simeq \text{“07.2005”};\\
 & \mathit{Trip} :: 1..2 + 2..2, \mathit{week},\ \mathit{Trip\_day} :: 4..11 + 14..14, \mathit{day},\\
 & \text{“07.2005”} :: 1..1 + 1..1, \mathit{month},\ \text{“07.2005”}\mathit{\_day} :: 1..1 + 31..31, \mathit{day}
\end{array}$$

What has happened: the constraint $\mathit{Trip\_day} \simeq \mathit{Trip}$ tells us that the amount of time that is represented by $\mathit{Trip\_day}$ in terms of days must be equivalent to the amount of time that is represented by Trip in terms of weeks. After applying the constraint *during*, $\mathit{Trip\_day}$ represents those intervals of 14 days that start earliest at day 2 and latest at day 17. The constraint $\mathit{Trip\_day} \simeq \mathit{Trip}$ finds the weeks started *within* those day intervals, and, in turn, find the days that start those selected weeks. Let us consider this by the illustration of the indices:



[9]The addition by 1 results from the fact that a single day has a duration of 1 by itself.

Before applying the conversion constraint $Trip\_day \simeq Trip$, the calendar constraints for the different representation of the trip are: $Trip\_day :: 2..17 + 14..14, day$ and $Trip :: -\infty..\infty + 2..2, week$. Applying $Trip\_day \simeq Trip$, enforces $Trip\_day$ and $Trip$ to represent the same amount of time (however in different types). So, we have to ask the following questions:

- Does (day) 2 start a week? – No.
- What is the *next* week that starts after (day) 2? – (week) 1; thus, the minimal starting time for $Trip$ is (week) 1.
- Which day starts this week? – (day) 4; thus, the minimal starting time of $Trip\_day$ is (day) 4.
- Does (day) 17 start a week? – No.
- Which week contains this day? – (week) 2; thus, the maximal starting time of $Trip$ is (week) 2.
- Which day starts this week? – (day) 11; thus, the maximal starting time of $Trip\_day$ (day) 11.

That is, after applying the conversion constraint $Trip\_day \simeq Trip$, the calendar constraints for the different representations of the trip are: $Trip\_day :: 4..11 + 14..14, day$ and $Trip :: 1..2 + 2..2, week$.

This example clarifies that multi-calendar constraint reasoning based on conversion constraints has the following advantages: first, the semantics of the different calendric types are maintained during constraint solving. Second, the domains of the variables can be much more reduced (e.g. the representation of $Trip\_day$ before and after applying the conversion constraint $Trip\_day \simeq Trip$); thus, multi-calendar constraint solving becomes more efficient. Finally, each pair of calendric types, even those which are not aligned such as months and weeks can be compared; thus, each multi-calendar constraint problem that is expressed in CaTTS-CL and that refers to calendric types defined in CaTTS-DL can be solved.

1.4.6 An Approach to Predicate Subtyping with Calendric Types

Chapter 6 introduces the language processor of CaTTS-DL. This language processor includes "meta type checking" approaches for CaTTS-DL and type checking approaches for the constraint language CaTTS-CL. The "meta type checker" verifies well-formedness of CaTTS-DL calendar specifications. Furthermore, it generates conversion functions from the calendric types' predicates specified in a calendar specification. A conversion function "casts" values from one calendric type (e.g. working weeks) into values of another calendric type (e.g. days). Those conversion functions are accessed by CaTTS-CL's multi-calendar constraint reasoner. Type checking CaTTS-CL programs after CaTTS-DL calendar specifications is based on an approach to subtyping with calendric types. CaTTS-CL's subtyping semantics is defined by *implicit coercion* [BTCGS91]. Coercion obtains a way to evaluate CaTTS-CL programs using CaTTS' multi-calendar constraint reasoner such that arbitrary calendric types defined in a CaTTS-DL calendar specification can be referred to.

Chapter 6 consists of seven (full) sections. Section 6.1 recalls basics in subtyping for those readers not familiar to. Advantages of CaTTS' programming language approach to calendric data modeling and reasoning using types and type checking approaches are gathered in Section 6.2. The concept of predicate subtypes as it is used in CaTTS is introduced in Section 6.3. Section 6.4 contains definitions for conversion function generation from CaTTS' type predicates for those language constructs which are considered in the current prototype implementation. A type system to check well-formedness of CaTTS-DL calendar specifications is formalized in Section 6.5. Finally, Section 6.7 defines typing and subtyping relations for CaTTS-CL programs typed after CaTTS-DL calendar specifications as well as a coercion semantics for subtyping in CaTTS-CL. Coherence for this coercion is recovered. This chapter comes with additional sections on the equivalence of calendric type definitions (Section 6.6) and typing of CaTTS-DL calendar specifications themselves (Section 6.9). Appendix C is associated with Chapter 6. This appendix summarizes the Haskell-based [Tho99] implementation of CaTTS' type checkers testing well-formedness of CaTTS-DL calendar specifications, type checking CaTTS-CL programs, and transforming CaTTS-CL programs into programs in the language that can be handled by CaTTS' constraint reasoner.

Both the working of CaTTS' meta type checker that verifies well-formedness of a CaTTS-DL calendar specification and the translation of CaTTS-CL programs in terms of implicit coercion are illustrated by example in the following.

Well-formedness of CaTTS-DL Calendar Specifications. For programming languages, types and type checking ensure the absence of certain undesirable behaviors during program execution. Execution is not a primary concern of the calendar definition language CaTTS-DL. In fact, CaTTS-DL merely provides with means to define for example what a day, a week, or a working week is. But type checking can still serve to reject calendar specifications that are erroneous or undesirable in some way. Let us consider the following CaTTS-DL calendar specification which defines types for days, weeks, Saturdays, and Sundays common in the Gregorian calendar.

```
...
   type day;
   type week = aggregate 7 day @ day(1);
   type saturday = select day(i) where relative i in week == 6;
   type sunday = select day(i) where relative i in week == 7;
...
```

This CaTTS-DL calendar specification is well-formed, because (1) the syntactic forms of CaTTS-DL are correctly used, (2) each type definition refers to some other type already defined, and (3) each of the types defines a reasonable set of values. Adding the following type declaration to the calendar given above, the calendar remains well-formed.

```
...
   type weekend_day = saturday | sunday;
...
```

The type `weekend_day` defines those days which are Saturdays or Sundays, denoted `saturday | sunday`. CaTTS-DL's type checker verifies that this type definition is well-formed, because the type `weekend_day` is constructed using the CaTTS-DL *or* type constructor denoted | which is applied to two calendric types `saturday` and `sunday`, already defined and, in turn, well-formed. Furthermore, conjuncting Saturdays and Sundays to a larger set of weekend days is reasonable, because Saturdays and Sundays are (specific) days.

The following type definition would however be rejected by CaTTS-DL's type checker.

```
...
   type sunday_week = week | sunday;
...
```

Defining a set that conjuncts the set of Sundays with the set of weeks is not at all reasonable: Sundays are specific days and weeks are intervals of seven days. In particular, `sunday` is an inclusion of days while `week` is an aggregation of days. Thus, CaTTS-DL's type checker rejects this type declaration, because the type constructor | can only be applied to calendric types which are inclusions of a common calendric type. For example, `sunday` and `saturday` are both inclusions of `day` while such a type cannot be inferred for types `sunday` and `week`.

Coercion Semantics for Subtyping in CaTTS-CL. Let us reconsider the CaTTS-CL program addressed above.

```
...
Trip is 2 week && Trip during "07.2005"
...
```

In this example, the program has been evaluated using CaTTS' constraint reasoner. We have merely mentioned that this program can be transformed such that it becomes reasonable for the solver. In what follows, it is illustrated how this transformation is performed in terms of coercion. At first, CaTTS-CL's type checker infers that `"07.2005"` is of type `month`, written `"07.2005":month`, and that the variable `Trip` has type `week`* (read as "interval of weeks"), written `Trip:week`*, as defined with the constraint `Trip is 2 week`. Since `Trip during "07.2005"` is a constraint, it has type boolean, written `Trip during "07.2005"`:$\mathbb{B}$ which is also inferred by the type checker. To evaluate this CaTTS-CL constraint using CaTTS' constraint solver, `Trip` and `"07.2005"` must have the same type. Not to charge the programmer with this requirement, subtyping is used; it says: if there is a common (super)type within which `Trip` and `"07.2005"` can be both represented, the constraint `Trip during "07.2005"` is well-typed. This is achieved using subtyping.

With subtyping CaTTS-CL's type checker infers that `Trip` and `"07.2005"` can be both represented in the type of intervals of days ($\texttt{day}^*$). The subtype checker than verifies the subtyping statements, i.e. whether $\texttt{week}^*$ is a subtype of $\texttt{day}^*$, written $\texttt{week}^*\leq\texttt{day}^*$, and whether `month` is a subtype of $\texttt{day}^*$, written $\texttt{month}\leq\texttt{day}^*$. Having verified those typing and subtyping statements, we know *that* `Trip during "07.2005"` is typeable and that $\texttt{week}^*$ and `month` are both subtypes of $\texttt{day}^*$. However, to evalute this program using CaTTS' constraint reasoner, we need to know *why* this is the case, because the constraint reasoner is based on the conversion constraints for values expressed in terms of $\texttt{week}^*$ and $\texttt{day}^*$ as well as for values expressed in terms of `month` and $\texttt{day}^*$. That is, the reasoner must know how to represent the values in the different types. This is achieved using a coercion semantics for subtyping in CaTTS-CL: the constraint `Trip during "07.2005"` is not only type checked and the subtyping statements $\texttt{week}^*\leq\texttt{day}^*$ and $\texttt{month}\leq\texttt{day}^*$ are not only verified. The constraint is also transformed to a constraint `X during Y` where the variables `X` and `Y` both have type $\texttt{day}^*$. Furthermore, the subtyping statements $\texttt{week}^*\leq\texttt{day}^*$ and $\texttt{month}\leq\texttt{day}^*$ are transformed into conversion constraints (of the constraint solver) that relate `X` (of type $\texttt{day}^*$) to `Trip` (of type $\texttt{week}^*$), written `X` $\simeq$ `Trip`, and `Y` (of type $\texttt{day}^*$) to `"07.2005"` (of type `month`), written `Y` $\simeq$ `"07.2005"`. The transformed program

`X during Y`, `X` $\simeq$ `Trip`, `Y` $\simeq$ `"07.2005"`

can be now evaluated using CaTTS' constraint solver as demonstrated in the previous section.

1.4.7 Conclusion of the Thesis

Chapter 7 concludes the thesis. This chapter consists of two sections. In Section 7.1 the results of the work reported about in this thesis are summarized. In particular, this section shows that the basic principles assumed in the introduction to this thesis are verified by documentary evidence of the calendar type language CaTTS: CaTTS provides the user with user-friendly language constructs to model cultural and professional calendars including various irregularities and regulations. CaTTS makes calendric data and calendars machine readable and processable by virtue of calendric types and type checking approaches. Types and type checking enables to incorporate CaTTS' calendar type definintion language, CaTTS-DL (in principle) into *any* programming or modeling language. This has been shown by incorporating CaTTS-DL into the constraint language CaTTS-CL using type checking approaches dedicated to constraint reasoning. CaTTS-CL comes along with a multi-calendar temporal reasoner that allows for expressing and solving appointment scheduling problems that refer to arbitrary calendric types defined in CaTTS-DL. CaTTS' reasoner implements an extension of the constraint system finite domains by means of *typed finite domains*, i.e. each domain is associated with a calendric type. Furthermore, the constraint solver implements a novel constraint, called *conversion constraint*, that allows for relating domains expressed in terms of different calendric types. This reasoner is proved to be efficient; in fact linear in the number of constraints and in the number of variables according to the size of the domains of the variables. Furthermore, the constraint solver is proved to be complete. Section 7.2 discusses perserpectives for future work that comes in two parts: at first, possible extensions and improvements of CaTTS, including CaTTS-DL and CaTTS-CL and the corresponding language processors are addressed. Second, an application of CaTTS' basic principles to other "concrete theories", in particular, to data modeling and reasoning with topologies using dedicated language constructs and reasoning methods is discussed.

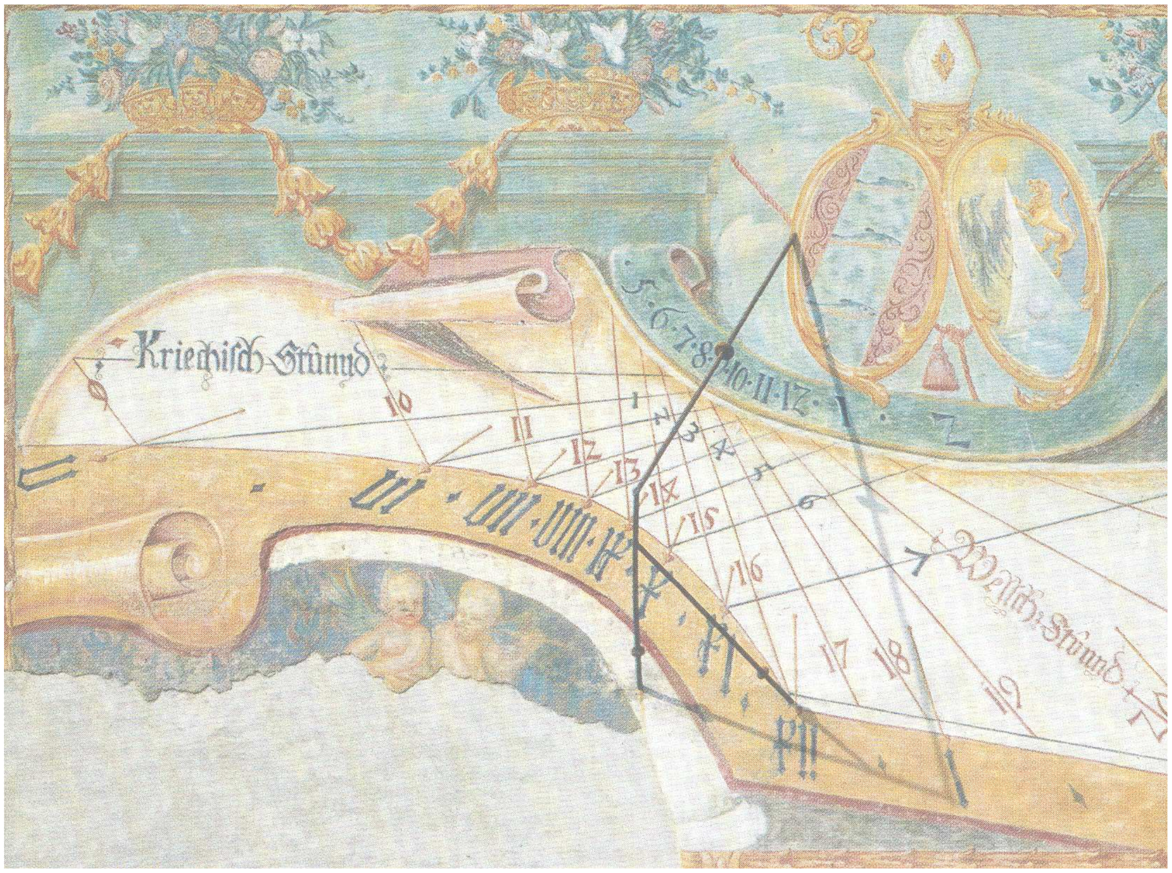


Figure 1.1: Sundial with East, Central, and West European Time. Fresco Cloister Osiach, Carinth

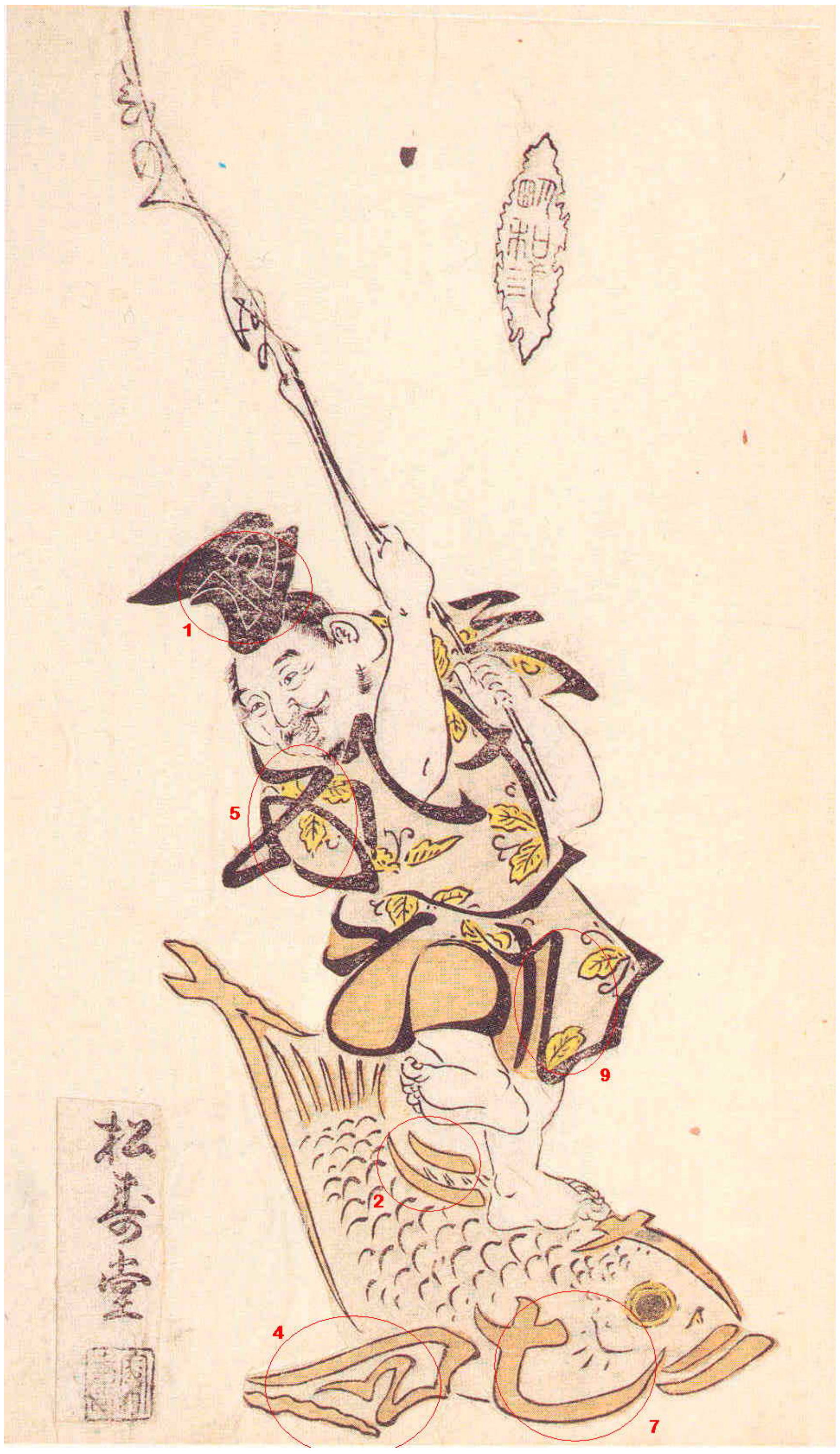


Figure 1.2: A Japanese calendar illustrations [be96], p.105.

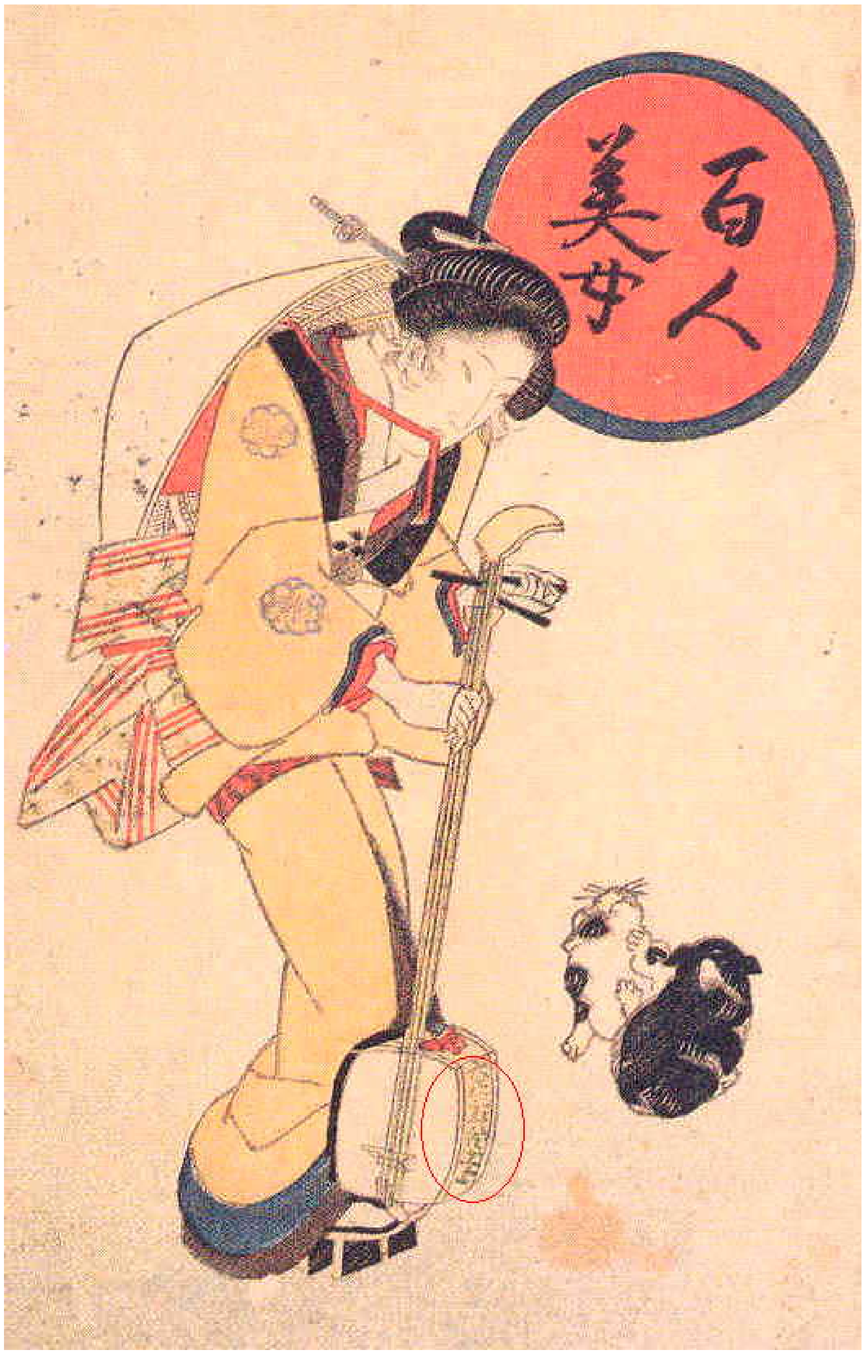


Figure 1.3: A Japanese calendar illustrations [be96], p.102.

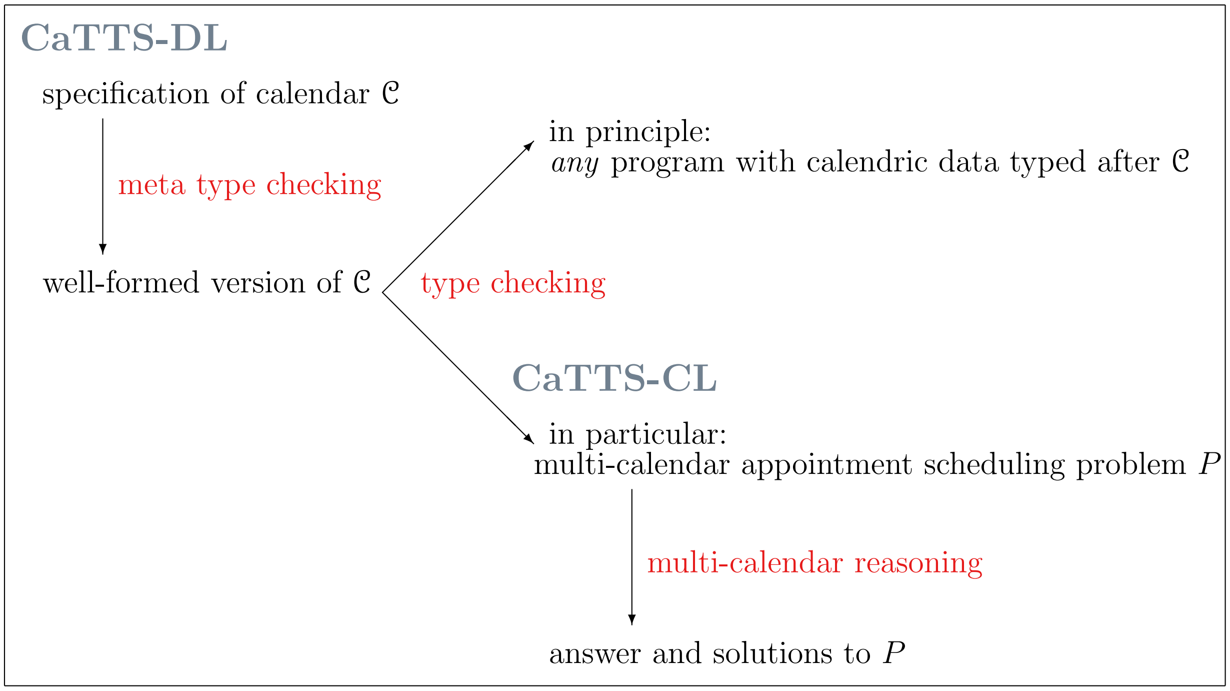

CaTTS-DL
specification of calendar $\mathcal{C}$
meta type checking
well-formed version of $\mathcal{C}$
type checking
in principle:
any program with calendric data typed after $\mathcal{C}$
CaTTS-CL
in particular:
multi-calendar appointment scheduling problem P
multi-calendar reasoning
answer and solutions to P

Figure 1.4: The system CaTTS.

Chapter 2

Background: Temporal Knowledge Representation and Reasoning for Information Systems

"quid est ergo tempus?
si nemo ex me quaerat, scio;
si quaerenti explicare velim, nescio."
(Aurelius Augustinus, 354 – 430, Confessiones, XI 14)

Temporal knowledge representation and reasoning is a major research field in Artificial Intelligence, in Database Systems, and in the Web and the Semantic Web. The ability to model and process time and calendar data is essential for many applications like appointment scheduling, Web services, temporal and active database systems, and so-called adaptive Web applications as well as mobile computing applications.

An exhaustive overview of all the approaches, results, and applications on temporal knowledge representation and reasoning would probably require an entire book by itself. This chapter has a less ambitious goal: first, it provides the reader with a general background in temporal data modeling and reasoning approaches, serving as an orientation guide. Second, it provides with links to further specific reading and application to temporal knowledge representation and reasoning. Third, it provides with a clear classification of important and well known approaches to time and calendars in Artificial Intelligence, Database Systems, and the Web and Semantic Web. Finally, it addresses some applications of time and calendar formalisms, in particular, to Web services, to versioning and change management of Web resources, and to adaptation on the Web. This chapter includes additional sections on calendric calculations and internationalization and localization efforts in the Web and the Semantic Web.

This chapter provides with a comprehensive comparison of CaTTS to the reviewed approaches to temporal knowledge representation and reasoning. CaTTS is a programming language approach to temporal and calendric data modeling specifying calendars as types with constraint-based reasoning approaches on calendric data referring to such calendric types.

2.1 Approaches to Temporal Knowledge Representation and Reasoning

Knowledge is the *symbolic representation* of aspects of some *discourse universe* such as time and calendars. An example for temporal knowledge represented in natural language is "John plays tennis every Saturday between 10 a.m. and 11 a.m.". Modeling formalisms and computation mechanisms to manipulate the knowledge of some discourse universe, to create solutions, and to formulate new problems have been developed. Such formalisms have been proposed as tools to define knowledge in terms of symbols that may be manipulated by computer programs.

In principle, symbolic representation of temporal knowledge allows for describing changes of the reality modeled in some system, for example a database. A computer program that allows for manipulating such knowledge usually provides with inference strategies to reason about time-dependent objects such as schedules, plans, actions, and changes. These and further considerations have given birth to a large field of research that can be summarized as the development of *time models*. Essentially two different approaches to represent time, *implicit time models* and *explicit time models*, exist. The computation mechanisms to manipulate temporal knowledge that is defined in such time models are often called *temporal reasoning methods*.

Usually, knowledge about time (and calendars) is obtained by extending a language with means to represent temporal data and to define a temporal reasoning system, i.e. a technique for reasoning about assertions formulated in the extended language. Such assertions usually depend on some intended application (e.g. scheduling, planning, changes and actions), determining which reasoning technique(s) might be applied. A temporal reasoner is usually part of a more general reasoning system.

Following approaches to temporal data modeling and reasoning are surveyed in this section.

1. Implicit time models which focus on time-dependent entities indicating that a change in time has occurred.

2. Explicit time models which consider time itself independent of anything that could happen in it.

3. Approaches to temporal reasoning in explicit time models based on constraint solving techniques.

4. Explicit time models with approaches to model different calendar units in terms of so-called *time granularities* as well as approaches to temporal reasoning with time granularities.

2.1.1 Implicit Time Models

In principle, implicit time models consider time dependent entities like *events* and *actions* indicating that a *change* has occurred in the modeled world, for example a database of facts. With such systems, time is not made explicit. Instead, time is implicitly specified in terms of the changes that have occurred in the past or that might occur in the future.

Implicit time models are particularly used to model events and actions that result in changes in the context of some application. Implicit time models are used to specify the effects of actions and to acknowledge and to react on changes in such systems.

Almost all implicit time formalisms provide a simplistic notion of time, however useful for simple problem-solving tasks: a state describes the world at an instantaneous time, i.e. at a time point. Actions are modeled as functions over states. Well-known implicit time models are the Situation Calculus [MH87] and the Event Calculus [KS86, Kow92]. In the field of program verification[1], Dynamic Logic [Pra76, Har79, Har84, Moo85] is a well-known implicit time formalism which is also applied to problems in temporal knowledge representation and reasoning. Implicit time models are widespreadly applied to active database systems, and recently, to model and process changes and actions in active Web and Semantic Web systems (cf. Section 2.3.5).

2.1.1.1 Situation Calculus

The Situation Calculus has been introduced by McCarthy and Hayes in 1969 [MH87]. This calculus has long been a foundation for temporal knowledge representation and reasoning in Artificial Intelligence. The Situation Calculus has been modified and/or extended with several different means [SG88, Web90, LS95, Ram00, McC02, MPP02]. In addition to research in knowledge representation and reasoning on actions and changes [GLR91, Lif91], the Situation Calculus is particularly used for practical work in planning [FN71, FHN72, L96́, GLL97], active database systems [Rei92, Rei95, BAF98], and in agent programming and robotics [BCF$^+$98, LLR99, BRST00, MSZ01, ABH$^+$02].

In the Situation Calculus, temporal knowledge is represented as a sequence of so-called *situations,* each being a description of the world at an instantaneous time, i.e. at a time point. The Situation Calculus models a *changing world* in terms of a totally ordered and discrete sequence of situations, i.e. the Situation Calculus models a set of points isomorphic to the integers. The changing world is formalized by a many-sorted predicate calculus with some reserved predicate and function symbols. The sorts of this calculus are *situations, actions,* and *objects. Fluents* are specific relations and functions defined over the sort of situations. Fluents are used to describe the world in each situation. Action performances, action preconditions, and histories are non-decomposable functions from one situation to another.

The Situation Calculus is feasible only in domains where merely one event can occur

[1]Note that research on program verification which inherently differs from research on temporal knowledge representation and reasoning for information systems is not considered in this survey.

at a time. Furthermore, no concept of an event taking time is supported. The Situation Calculus does not provide a notion of persistence: an event that is true at one situation needs to be explicitly verified at any succeeding situation.

Various axiomatizations of the Situation Calculus in some first-order language [Bak91, PR93, Rei93, LPR98, Rei01] as well as some formalizations of the Situation Calculus in terms of logic programs [KS94, BDD95] are proposed.

2.1.1.2 Event Calculus

The Event Calculus has been introduced by Kowalski and Sergot in 1986 [KS86, Kow92]. Various modifications and/or extensions of this calculus exist [Sha90, CMP93, CCM95, Sha95, SK95]. The most notable variation is the so-called Simplified Event Calculus [Kow92] that is aware of some of the problems appearing with the original calculus. Those problems are discussed in [Mis91, DMB92]. This simplified calculus has been applied to problems in planning [DMB92, YS02] and in active database systems [Kow92, FWP97].

The Event Calculus is a formalism to reason about *events.* It is based on a many-sorted first-order predicate calculus or the Horn clause subset of such a calculus. In most formalizations of the Event Calculus, a linear, point-based time line is assumed. Events initiate and terminate time intervals over which *fluents* hold. Fluents are properties that can have different values which are manipulated by the occurrences of events at different instantaneous times, i.e. at different time points.

Various axiomatizations of the Event Calculus and of the Simplified Event Calculus are proposed [Esh88, Sha89, Sha90, Mis91, Kow92, SK95, MS99].

2.1.1.3 Dynamic Logic

Dynamic Logic is based on classical logics, in particular, propositional and predicate logics and modal logics which are combined with an algebra of regular events. Dynamic Logic enables verification of imperative programs and program specifications. It has been proposed by Pratt [Pra76] (with an emphasis on the modal nature of program interactions), Harel [Har79, Har84], and Moore [Moo85].

The basic idea of Dynamic Logic is to model *programs* by modal operations. Such programs change the values of variables causing changes of the truth values of some formula. Among the numerous formalisms for (formal) program reasoning, Dynamic Logic enjoys the singular advantage being strongly related to classical logics. Therefore, Dynamic Logics gains from the advantages of classical logics compared to (often more complex) logics frequently applied to reason about programs.

Although Dynamic Logic has been initially introduced as a formalism for program verification, it has been turned out to be a formalism for reasoning about some actions in (natural or artifical) systems. In Artificial Intelligence, Dynamic Logic has been adapted for tasks such as the description of actions in some common-sense world and for specifying particular reasoning systems [SWM95, SGdMM96, Mey99].

2.1.2 Explicit Time Models

Explicit time models consider *time* itself, independently of any event or action that might occur. The flow of time is explicitly represented, defining a *time model* that specifies one (or more) time lines with time primitives, in particular, *points* and/or *intervals*. Some time models specify more than one time line which are either related in parallel or vertically. The former leads to *hierarchical time models*, in particular used to define calendar units like "day", "week", or "working year", so-called *time granularities*. The latter leads to *multi-dimensional time models*, in particular used to define temporal histories, for example of transactions of objects and/or validities of some properties of those objects stored in a system like a database.

Explicit time models are either point-based [Bru72, KG77, McD82, Sho87] or interval-based [Ham72, Dow79, NS80, All83, HA89]. In the former, reasoning is performed on *points*. In the latter, reasoning is performed on *intervals*. Some proposals have been made to combine point-based and interval-based time models [Vil82, Boc90, Gal90, VS96]. Such combined time models are rather artifical, raising several problems (cf. Section 3). Furthermore, time models for generalized intervals (i.e. intervals which are not necessarily convex) have been proposed [LMF86, Lad87, DC91, NS92, Lig98].[2]

Explicit time models have been widely applied in temporal reasoning [Vil82, All83, vBC90, VKvB90, KL91, DMP91, Kou92, Mei96], temporal database systems [ÖS95, Sno95, Gan99], planning and scheduling [RS90, All91], and (time) ontology design for Web and Semantic Web applications and Web services [DAR02, PH04].

2.1.2.1 Point-based Models

Points are modeled as duration-less portions of time, i.e. their beginning and ending times are identical. Point-based time models have been influenced by physics [New36] where it is common to model time as an unbounded, ordered continuum of (partially) ordered points isomorphic to the real numbers. In Artifical Intelligence, research in this tradition can be found in [Bru72, KG77, McD82, Sho87].

A point-based time model is defined on a structure $(P, <_P)$, where P denotes a set of points and $<_p$ an ordering of P.

A point-based model defines at least a partial (i.e. irreflexive, antisymmetric, and transitive) order on points. Let $p, q, r, s, t \in P$ points.

[2]Note that combined time models and time models based on generalized intervals are not surveyed in detail for the following reasons: CaTTS is based on a purely interval-based time model (cf. Chapter 3). CaTTS does not provide any means to deal with generalized intervals. Investigating means to deal with generalized intervals in CaTTS would probably require an entire thesis by itself. Further reading on generalized intervals can be found in [Spr02].

An **ordering** over $(P, <_P)$ can be defined as follows:

(irreflexivity)	$\neg(p <_P p)$
(antisymmetry)	$((p <_P q) \wedge (q <_P p)) \Rightarrow (p = q)$
(transitivity)	$((p <_P q) \wedge (q <_P r)) \Rightarrow (p <_P r)$

Additionally, it may have (some of) the following properties:

- **left-linearity or linearity:**

(left-linearity)	$((q <_P p) \wedge (r <_P p)) \Rightarrow ((q <_P r) \vee (q = r) \vee (r <_P q))$
(linearity)	$(p <_P q) \vee (p = q) \vee (q <_P p)$

- **boundness or unboundness:**

(right-boundness)	$\forall p \exists q.(q <_P p)$
(left-boundness)	$\forall p \exists q.(p <_P q)$

- **discreteness or density:**

(discreteness)	$\forall p, q.(p <_P q) \Rightarrow \exists r.((p <_P r) \wedge \neg\exists s(p <_P s <_P r))$
	$\forall p, q.(p <_P q) \Rightarrow \exists r.((r <_P q) \wedge \neg\exists s(r <_P s <_P q))$
(density)	$\forall p, q.(p <_P q) \Rightarrow \exists r.(p <_P r <_P q)$

Dense and continuous point-based models provide a means to model continuous changes such that any extend of time can always be partitioned into subintervals. This property is necessary for example for modeling planning problems where tasks are frequently decomposed into subtasks. In dense and continuous models it is not possible to refer to some next (resp. previous) time. This is however possible in discrete models. Any finite strict partial order (i.e. if p and q are points, then $p < q$ or $q < p$) is automatically discrete.

Note: Including a concept of (metric) *durations* into a time model, i.e. specific amounts of time with a given length but without begin and end points into a point-based time model, a distinction between continuous and discrete time becomes important: if the time line is continuous, a point has no duration, but in discrete time, it always has some duration.

The properties introduced above are sufficient to achieve a certain level of completeness. Two theories are known to be syntactically complete [vB91]: the unbounded dense linear theory and the unbounded discrete linear theory.

Point-based models provide with a simplified concept useful in simple problem solving tasks such as maintaining different states of a system like a database. Points describe the modeled world, for example in a database of facts at an instantaneous time where events are represented by dates. Actions are modeled in such systems as functions mapping between points. For example, if dates are represented by integers, then the temporal order between two dates could be simply computed using numerical applications. Point-based time models are predominately applied in temporal databases [ÖS95, Sno95, Gan99]. In such systems, facts are indexed by dates, each represented by a time point. The temporal ordering between dates can be computed by some simple operations.

2.1.2.2 Interval-Based Models

In several applications, events simply cannot be assigned to a precise date, for example, to model the fact that two events cannot happen at the same time. Furthermore, in many applications events appear that must be decomposed into subparts, for example in planning tasks where an event might consist of several sub-events. Recall that points cannot be decomposed. Although some events appear to be instantaneous, for example, one might argue that "step inside the house" is instantaneous, it also appears that such events might be either decomposed or varied in their "precision" (e.g. from minute to second). Thus, point-based time models are neither useful nor expressive enough for many applications.

Starting from a point-based time model, intervals might be implemented into this approach. In a time model of totally ordered points, an interval can be represented by an ordered pair of points with the first point less than the second. To ensure that such intervals can meet (e.g. "the light was on" and the "the light is off"), i.e. only having one endpoint in common, the endpoints of the intervals must be *less*, i.e. for an interval i with endpoints i^- and i^+, i.e. $i^- < i^+$. This can be achieved when the intervals are closed in their starting points and open on their ending points (or vice versa). However, this requirement points out that a time model based on points does not correspond with our intuitive notion of time. Furthermore, there are reasons why the modeling of intervals in terms of endpoints is still inconvenient: the modeling of temporal knowledge is significantly complicated, because both, points and intervals have to be considered. Furthermore, having for example an event e holding sometime during an interval i which in turn is during an interval i', then e holds during i', as well. Thus, relations between intervals and events may be "carried forward" such that reasoning can be kept local. It is not clear to maintain those properties when considering intervals modeled by endpoints in a point-based time model.

To overcome the problems of point-based time models, purely *intervals-based* time models have been introduced [Ham72, Dow79, NS80, All83, HA89] where intervals are the time model's only primitives. An interval-based time modeled well-known and widely applied in Artificial Intelligence, Knowledge Representation, and Information Systems is Allen's interval calculus [All83] exclusively based on intervals and the 13 basic relations between pairs of them. The thirteen basic interval relations are illustrated in Figure 2.1.

Allen [All83] initially takes a structure $(I, \mathcal{R})$ where I denotes a set of intervals and and $\mathcal{R}$ the set of the 13 relations between pairs of elements of I. $\mathcal{R}$ is informally specified by the following axiom schemas:

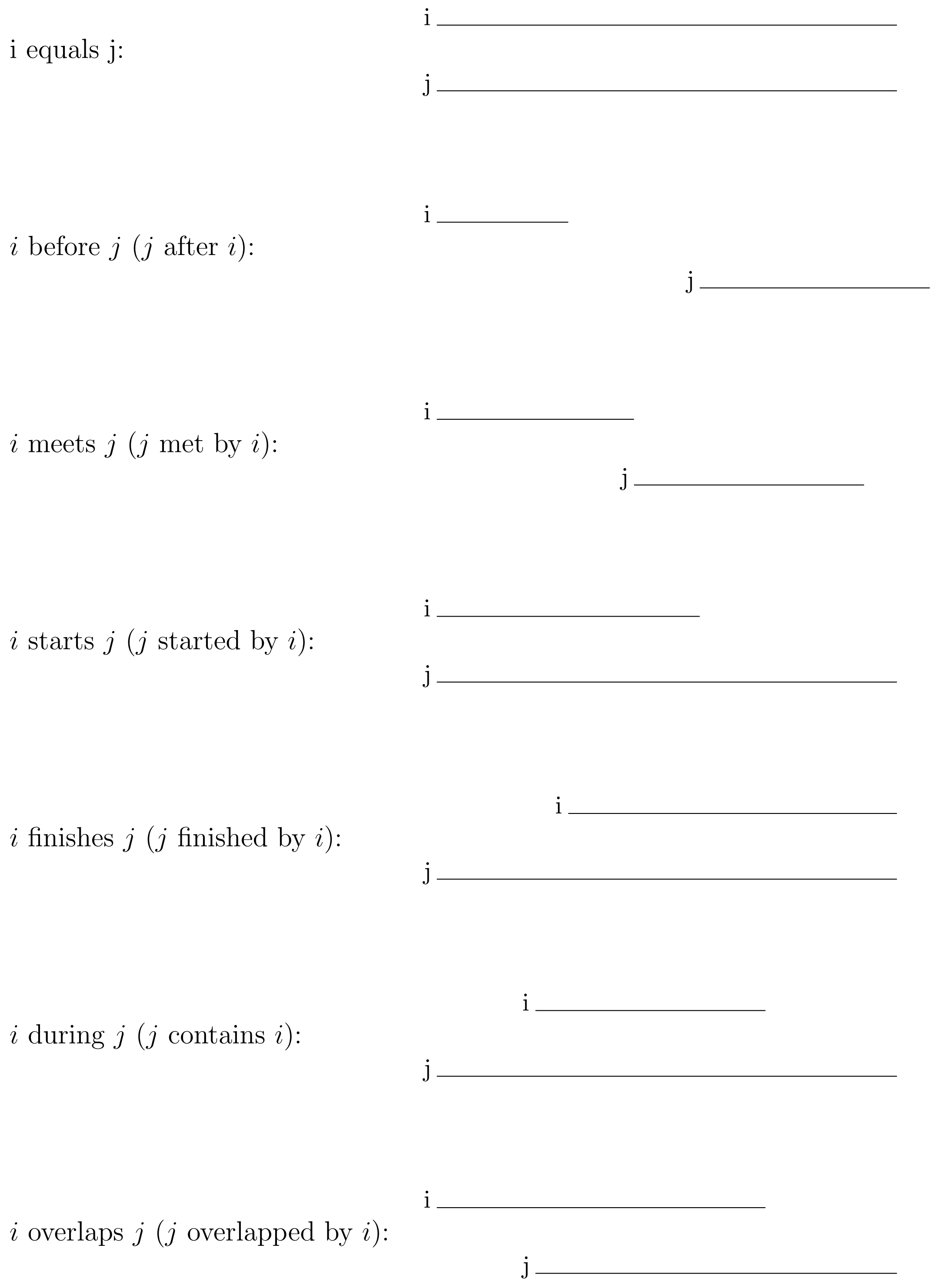

i equals j:
i
j
i before *j* (*j* after *i*):
i
j
i meets *j* (*j* met by *i*):
i
j
i starts *j* (*j* started by *i*):
i
j
i finishes *j* (*j* finished by *i*):
i
j
i during *j* (*j* contains *i*):
i
j
i overlaps *j* (*j* overlapped by *i*):
i
j

Figure 2.1: The 13 basic relations between two intervals i and j.

1. For all $i \in I$, there exists an interval $j \in I$, $i \neq j$ with $R(i,j)$ for $R \in \mathcal{R}$.

2. The relationships in $\mathcal{R}$ are mutually exclusive:
   $\forall i,j \in I, R \in \mathcal{R}.\forall R' \in \mathcal{R}.(R(i,j) \Rightarrow \neg R'(i,j)$

3. The relationships have a transitivity behavior. For example, if $i\ before\ j$ and $j\ meets\ k$, then $i\ before\ k$. The transitivities between all pairs of the 13 basic relations are given in a transitivity table in [All83].

The relationships between intervals have also been studied by other researchers: van Benthem [vB91] has introduced two basic relations, *precedes* and *contained-in* which are consistent with the structure of intervals over the rational numbers. Ladkin [Lad87] has defined each of the 13 basic interval relations in terms of the relation *precedes*. Allen and Hayes [HA89] have re-defined the interval calculus in terms of the relation *meet*. Characteristics, axiomatizations, comparisons, and expressiveness of those three interval calculi have been thoroughly investigated by Ladkin in [Lad87].

2.1.2.3 Combined and Generalized Models

In addition to purely point-based and purely interval-based time models, models integrating points and intervals [Vil82, Boc90, Gal90, VS96] as well as models to represent intervals which are not necessarily convex, referred to as *generalized intervals* [LMF86, Lad87, DC91, NS92, Lig98], have been proposed for different purposes. However, note that most of these approaches are computationally rather complex, raising several (artificial) problems.

Points and Intervals Combined. Differentiating between events which are instantaneous, i.e. duration-less events and events which have a temporal extend, i.e. continuous events has been motivated by accomplishing events like "to close the door" and continuous changes like a "100-meter sprint". Although this argumentation is not convincing (cf. Chapter 3), various time models integrating points and intervals have been proposed [Vil82, Boc90, Gal90, VS96].

The pioneer work in this tradition by Vilain [Vil82] has combined Allen's interval calculus [All83] with points and (primitive) relations between points and between points and intervals, yielding in 26 basic relations holding between points and/or intervals.

Vila and Schwalb [VS96], whose work has been inspired by the works of Bochman [Boc90] and Galton [Gal90], have defined points from intervals such that an interval is defined by an ordered pair of ending points. A many-sorted first-order axiomatization is proposed for this calculus [VS96] with sorts for points and intervals and three primitive relations "<" (less) over intervals and "*begin*" and "*end*" over intervals and points. Dense and discrete time is differentiated by adding the denseness axiom over points.

Generalized Intervals. Motivated by planning problems [LMF86, NS92, TC95, KM99, Ter00] that require an abstraction from the number of times an event repeats, or from the number of times an event occurs in a temporal relation, for example scheduling one's office hours, time models for not necessarily convex intervals, so-called *generalized intervals* have been proposed [LMF86, Lad87, DC91, NS92, AK94, Lig98]. Generalized intervals are intended to represent

- periodic events such as "John has breakfast every morning",
- interrupted events such as "John interrupts his breakfast to answer the phone", and
- events that consist of several related sub-events such as activities related to a planned event.

Since generalized time intervals are nothing more than mere collections of (possibly infinite many) intervals, most of the proposed models [LMF86, Lad87, DC91, NS92, AK94, Lig98] are based on Allen's interval algebra [All83]. Those approach basically define additional (to the basic interval relations which might be conjuncted) specific relations between such interval collections. The proposals differ in the description of the generalized intervals as well as in the basic relations defined between pairs of them: Ladkin [Lad87] has proposed a general taxonomy of the set of all relations between any pair of generalized time intervals in terms of specific relationships between such intervals. The suggested relations are derived from applications for task descriptions and management and process and action algebras. Ligozat [Lig98] has further extended this work. Khatib's [AK94] framework bases on matrices that represent binary relations between sequences of time intervals. Those matrices contain the time interval relations between any pair of convex components of the considered generalized time intervals.

2.1.3 Temporal Constraints

In the following, formalisms for *Temporal Constraint Satisfaction Problems (TCSPs)* based upon relating points and/or intervals are surveyed. For all representations in this section, time is assumed being linear, dense, and unbounded which is common with most proposals. Two principle temporal constraint formalisms have been introduced in the literature:

1. *qualitative* temporal constraints, in particular in Allen's interval calculus [All83, vBC90, VKvB90, KL91, Mei96], and
2. *metric* temporal constraints, mainly based on disjunctive linear relations between time points [KL91, DMP91, Kou92, Mei96] to manipulate dates and durations.

Combined (qualitative and/or quantitative) formalisms, based on points and/or intervals, and binary relations between them are proposed in [Vil82, Mei96].

Recall that a *Constraint Satisfaction Problem (CSP)* is defined by a finite sequence of variables $\mathcal{X} := X_1, \ldots, X_n$ where $n > 0$, with respective domains $\mathcal{D} := D_1, \ldots, D_n$, i.e. variable X_i ranges of the domain D_i together with a finite set $\mathcal{C}$ of constraints, each defined on a subsequence of $\mathcal{X}$. A *constraint* C on $\mathcal{X}$ is a subset of $D_1 \times \ldots \times D_n$. If C

equals $D_1 \times \ldots \times D_n$, then C is *solved.* A CSP is *solved,* if all its constraints are solved and no domain of it is empty, and *failed,* if it either contains the false constraint, usually denoted $\perp$, or some of its domains are empty. In a TCSP, variables represent points and/or intervals and constraints represent temporal relations between them.

A *solution* to a CSP is a sequence of legal values $(d_1, \ldots, d_n) \in D_1 \times \ldots \times D_n$ if for every constraint $C \in \mathcal{C}$ on the variables $X_{i_1}, \ldots, X_{i_m}$, $(d_{i_1}, \ldots, d_{i_m}) \in C$, i.e. if for all variables of the CSP all constraints are satisfied. If a CSP has a solution, it is *consistent*, otherwise *inconsistent.*

Most approaches to TSCPs have focused on satisfiability problems, i.e. *consistency*, the problem of deciding whether a set of temporal formula (modeled by a set of constraints) has a model or not. In the context of Allen's calculus, the problem of *entailed* relations is also considered.

Consider the following temporal reasoning problem.

Example 2.1 *John, Mary, and Tom work in the same bank, but in different affiliations. They plan a phone conference. They have decided to phone in the morning before the bank's affiliations open at 9:00. John leaves home between 7:50 and 8:00. It takes him 15 to 20 minutes to reach the office. When he arrives, he can wait 30 minutes, at most 40 minutes. Mary can phone John 5 to 10 minutes after she arrives at the office or between 8:40 and 8:45. Tom arrives before the bank's affiliation opens.*

Reasoning about such information is the principal task of temporal reasoning systems. One wants to answer queries like "is the contained temporal information consistent" (i.e. if there is no conflict between the temporal information modeled), or "what is the earliest/latest possible time for the phone conference to take place".

Properly analyze of such problems leeds to an abstract analyze of *events* (i.e. activities that take time) like being present at work, walking to the office, receiving a phone call, etc. Such events can be identified with (different) time primitives: *points*, e.g. `8:20`, the time when Tom is soonest at the office, or the time, Mary leaves home, and *(generalized) intervals*, `[7:50,8:00]` the time when John leaves home, the time it takes John between leaving his home and arriving at the office, or the time Mary is able to phone John.

Such events may be either related by *metric* constraints or by *qualitative* constraints. Metric (time) constraints are dates and durations between two events, placing absolute bounds on events, for example 9:00, the time the bank opens according to Example 2.1. Qualitative (time) constraints are relative positions of events, for example John's presence at the office overlaps with Mary's presence at the office or Tom is at the office before the affiliation opens according to Example 2.1.

Note: Metric and qualitative constraints serve to model inherently different temporal reasoning problems. To solve metric and qualitative temporal reasoning problems, different constraint programming techniques are applied. In particular, metric constraints can be modeled by disjunctive linear relations, solved using *backtracking* (and *bounds/arc* *consistency*) techniques [Apt03] while qualitative constraints are modeled by the (transitivity property of) temporal relations themselves, solved using *path consistency* techniques [Apt03].

2.1.3.1 Metric Temporal Constraints

In the simplest case, temporal information is available in terms of dates, durations, or other precise numeric form. In this case, metric temporal constraints can be formulated, referring to absolute distances between points. Temporal information can be therefore represented numerically such that it may be easily computed, merely subtracting numeric values. Such temporal information can be modeled in so-called *acyclic directed graphs.* In such a graph, (temporal) events are represented by nodes, and the distances between these events are represented by the graph's edges, obtaining a partial order on temporal events. A known event occurrence may then be represented by a constraint. Such an event constraint is represented by a pair of its earliest and latest appearance in time. Those times are computed by adding up distances between the known event and some other (known) event. If the graph contains different possible paths between (unknown) events, its earliest (resp. latest) appearance are modeled by minimal (resp. maximal) path distances.

In the common (more complex) case, the precise numeric information is not available and/or information about distances is not precise, but expressed as a range of possible distances. These distances may be either precise distances or completely qualitative distances like ranges.

Dechter, Meiri, and Pearl [DMP91] have applied the technique of CSPs to model metric temporal information, aiming at a TCSP where each variable represents a point in time. Two types of constraints are introduced:

- *unary constraints* to represent point to date information, i.e. a point t is represented by a finite domain $\{i_1, ..., i_n\}$ with $n \geq 1$ and $t \in i_1 \vee ... \vee t \in i_n$, and
- *binary constraints* to represent point to point information, i.e. the distance between two points t_1, t_2 is represented by a finite domain $\{i_1, ..., i_n\}$ with $n \geq 1$ and $t_2 - t_1 \in i_1 \vee ... \vee t_2 - t_1 \in i_n$.

Turning attention back to the introducing Example 2.1, the temporal event that John leaves home between `7:50` and `8:00` may be formulated by an unary constraint t:

$$t \in \{(50, 60)\}$$

where `7:00` is chosen as relative origin in time and minute is the chosen time unit.

The temporal event 'Mary has time to phone John' may be specified by a binary temporal constraint $t_2 - t_1$:

$$t_2 - t_1 \in \{(65, 70), (100, 105)\}$$

where `7:00` is chosen as relative origin in time expressed in the time unit minute, as well.

Let t and s be two metric constraints with finite domains i and j. Then two operations over the set of metric temporal constraints are defined as follows:

- *Intersection:* The intersection of two metric temporal constraints t and s, denoted $t \bigoplus s$, is the set-theoretic intersection of their domains:

$$t \bigoplus s = \{x | x \in i \wedge x \in j\}$$

- *Composition:* The composition of two metric temporal constraints t and s, denoted $t \bigotimes s$, is defined as distance addition on their domains:

$$t \bigotimes s = \{z | \exists x \in i, \exists y \in j.x + y = z\}$$

Most of the algorithms proposed to manipulate metric temporal constraints are based on *Disjunctive Linear Relations (DLRs)*, and in particular, most tractable metric temporal formalisms [KL91, DMP91, Kou92] are subsumed by the Horn DLR constraint framework [Apt03].

Known tractable formalisms for metric temporal constraints to express statements such as "an event e happens 5 time units before event e" which are neither expressible in Allen's (qualitative) interval calculus [All83] nor subsumed by the Horn DLR framework are discussed in [Mei96, DJ97].

2.1.3.2 Qualitative Temporal Constraints

Given two temporal primitives, i.e. points and/or intervals p and q, a *qualitative temporal constraint* $(p, q) : R$ says that p and q are supposed to stand in *one* of the relations $R \in \mathcal{R}$, where $\mathcal{R}$ is a finite set of *basic temporal relations* that may hold between the two temporal primitives p and q. The set $\mathcal{R}$ of basic temporal relations between points and/or intervals is given in the following:

- basic *point to point* relations [McD82, Vil82] possibly holding between two points, illustrated in Table 2.1,
- basic *interval to interval* relations [All83] possibly holding between two intervals, illustrated in Table 2.2, and
- basic *point to interval* and *interval to point* relations [Vil82, Mei96] possibly holding between a point and an interval, illustrated in Table 2.3.

Relation	**Inverse**	**Relation on Endpoints**
t before s	s after t	$t < s$
t equals s		$t = s$

Table 2.1: The basic binary relations between two points t and s.

Relation	**Inverse**	**Relation on Endpoints**
i equals j		$i^- = j^-, i^+ = j^+$
i before j	j after i	$i^+ < j^-$
i during j	j contains i	$j^- < i^-, i^+ < j^+$
i starts j	j started_by i	$i^- = j^-, i^+ < j^+$
i finishes j	j finished_by i	$j^- < i^-, i^+ = j^+$
i meets j	j met_by i	$i^+ = j^-$
i overlaps j	j overlapped_by i	$i^- < j^- < i^+ < j^+$

Table 2.2: The basic binary relations between two intervals $i = [i^-, i^+]$ and $j = [j^-, j^+]$.

Relation	**Inverse**	**Relation on Endpoints**
t before i	i after t	$t < i^-$
t after i	i before t	$t > i^-$
t during i	i contains t	$i^- < t < i^+$
t starts i	i started_by t	$t = i^-$
t finishes i	i finished_by t	$t = i^+$

Table 2.3: The basic binary relations between a point t and an intervals $i = [i^-, i^+]$.

Let us turn attention back to the introducing Example 2.1. Let further p and q be two temporal primitives. An example of a qualitative temporal constraint between p and q, describing the temporal relation of the fact when Mary and John arrive at the office is:

$$(p, q) : \{before, meets, overlaps\}$$

This qualitative temporal constraint means *p before q* or *p meets q* or *p overlaps q*, i.e. a qualitative temporal constraint describes a disjunction of the possible basic temporal relations between a pair of temporal primitives.

In the point algebra, 2^3 subsets of possible combinations of basic temporal relations (i.e. qualitative temporal constraints) exist, in the interval algebra, 2^{13}, and in the point-interval and interval-point algebra 2^5, each, qualitative temporal constraints exist. Two binary operations are defined over the set of these qualitative temporal constraints:

- *Intersection:* the intersection of two qualitative temporal constraints R_1 and R_2, denoted $R_1 \bigoplus R_2$, is the set-theoretic intersection $R_1 \bigcap R_2$.
- *Composition:* the composition of two qualitative temporal constraints R_1 and R_2, denoted $R_1 \bigotimes R_2$, is defined by a transitivity table [All83], i.e. the 3-elemental transitivity: from $(p, q) : R'$ and $(q, r) : R''$, $R' \bigotimes R''$ is the least restrictive relation between p and r.

Since the constraint propagation algorithm for computing the closure of a set of propositions (i.e. to check for consistency) in the interval algebra is sound but not complete, time intervals are represented by their ending points in many qualitative temporal reasoning applications, in particular when points and intervals are "first-class citizens" of the respective time model, benefiting from the computational advantages of the point algebra [Mei96].

Note that only a part of Allen's interval algebra [All83], that of convex intervals, may be expressed in the point algebra, using the three point relations *before, equals,* and *after.* A complete classification of the computational complexity of Allen's interval calculus and its tractable subclasses (in particular those subclasses of the interval algebra which are subsets of the ORD-Horn algebra) is presented in [KJJ03]. Note further that integrating point-to-point, interval-to-point, and point-to-interval relations into tractable subclasses of Allen's interval calculus does not change its computational complexity [Mei96] in a qualitative temporal reasoning framework.

2.1.3.3 Metric and Qualitative Constraints Combined

The two kinds of temporal constraints, qualitative and metric, have been combined into a single representation scheme accepting both kinds of temporal information. Mainly two different approaches exist:

1. The qualitative and metric temporal constraints are kept and processed separately. Subsequently the individual parts are composed to a global solution. This approach has been followed by Kautz and Ladkin [KL91].

2. The qualitative and metric temporal constraints are integrated into a single *general temporal network* where temporal variables represent points (resp. intervals) along with a set of unary and binary constraints which may be qualitative or metric. Algorithms solving such temporal networks involve different constraint satisfaction techniques [Apt03], in particular *path consistency* and *backtracking*. This approach has been followed by Meiri [Mei96].

Different algorithms for TCSPs in which the relationships among temporal constraints can be modeled by directed graphs have been developed. Algorithms that operate over such so-called *constraint networks* are for example question answering and consistency checking. In those algorithms, constraint propagation is performed on a network of intervals and/or points. The nodes represent intervals and/or points, and the arcs are labeled with sets of binary relations (cf. Table 2.1, Table 2.2 and Table 2.3) on intervals and/or points. The labels represent constraints on the relations among the nodes. A good survey on constraint propagation algorithms for TCSPs is given in [FGV05].

2.1.4 Time Granularity Systems

The importance of data modeling and reasoning with data referring to different calendar units like “day”, “week”, or “teaching term” such as the widespread Gregorian times and dates has been widely recognized in the research areas of Artificial Intelligence and Database Systems. For example, a person being in Munch from Tuesday to Thursday and in London from Friday to Sunday will meet another person in London on Friday at 8 p.m. One might question how to merge the apparently contradictory information: the informations concerning the first person have to be precised in terms of hours. If the reasoner has some additional information about the temporal extend of the trip from Munich to London, one might reason about the time the first person have to leave Munich at the latest to be in London in time.

In the research areas of Artificial Intelligence and Database Systems one of the most promising approaches handling such problems has been the introduction of the concept of *time granularity* [Mon96, Je98, BJW00, Euz01]. Time granularities may be used to specify either the occurrence time of different classes of events (modeled for example by points or intervals) with time granularities or the temporal relations such as Allen’s interval relations between such events. For example, the temporal event of a flight departure may be given in the time granularity “minute”, and the temporal event of a student’s examination in that of “day”, or the temporal relation “during” may be given in the time granularity of “day”, describing the stay of some person in London. Thus, time granularities can be used to specify both metric and qualitative temporal reasoning frameworks. To apply time granularities in such frameworks, not only the granularities themselves, but also meaningful relations between them are needed: if in some application events are expressed in different time granularities, the system must "compare" them in a meaningful manner to reason on relations formulated over such "differently grained" events.

With a common understanding [Mon96, Je98, BJW00, Euz01], time granularities are discretizations of a totally ordered (continuous) time line isomorphic to the real numbers. A time granularity is a (partial) partition[3] of such a time line into a (infinite) countable set of aggregates of points (of such a time line), so-called *granules*, isomorphic to the integers. Examples of granules are "02.01.2005" of time granularity day and "the first week in the year 2005" of time granularity week. The granules of each time granularity are non-overlapping, and they preserve the order of the underlying time line. Granules may be considered either as points or as intervals.

Numerous different formalizations of the concept of time granularity, various relationships between them, and several temporal operations on them have been proposed. The formalizations follow set-theoretic, logic-based, and automata-based approaches. Well-known approaches in each of those traditions are surveyed in the following.

Applications of systems based on time granularities might be planning, scheduling, medical monitoring, temporal databases including temporal data models and temporal query languages, data mining, and time management in data warehouses and work-flow systems. Newer applications for time granularities could be found in the Web and Semantic Web, in Web services, so-called adaptive Web systems, and mobile computing applications. However, beyond the temporal database query language TSQL2 [Sno95], practical applications, in particular, implementations of time granularity systems hardly exist.

2.1.4.1 Set-theoretic Time Granularity Systems

A lot of work on set-theoretic approaches to time granularities has been done in Artificial Intelligence [MMCR92, Fre92, CEMP93, Euz93, Her94, GPP95, BCDE00] and in Database Systems [And83, CR87, WJL91, WJS95, Wan95, Sno95, BJW00, NWJ02].

In Artificial Intelligence, proposals for metric time granularities based on a hierarchy of strictly constrained time granularities with upward and downward conversions on points and intervals have been made [MMCR92, CEMP93]. In [Euz93], a less-constrained metric formalization has been proposed. Qualitative time granularities have been proposed in [Fre92, Her94, GPP95]. In this approach, a separate set of relationships is defined at any granularity level. A qualitative, interval-based framework that uses the same representation formalism at each granularity level is proposed in [Euz93]. In [Euz93], conversions between time granularities are defined on the relations rather than on objects between those relations might hold. Note that these qualitative proposals can only reason in homogeneous networks where all relations are given in the same time granularity. The network can be then converted (into another time granularity) only as a whole. A qualitative formalization for time granularities where (point-based) objects are converted is proposed in [BCDE00].

[3]in the mathematical sense of a partition of a set

In Database Systems, a large amount of work on time granularities exists. This approach to time granularities is inherently point-based and metric: conversions which introduce the (unsolved) problem of indeterminacy when converting to a finer time granularity is performed on point-based objects, i.e. granules are considered as points with time granularity. This approach suffers from some not only theoretically but also practically relevant problems such as the equivalence problem of time granularities. Furthermore, it only partially works out the problem of time granularity conversions. This problem of the metric, point-based approach is discussed in [FM01]. A comprehensive presentation of this metric, point-based approach to time granularities is given in [BJW00]. In what follows, a brief summary of the historically development of this approach is given.

Initially in [And83], the need to support different time granularities in temporal databases has been addressed. Based on Anderson's work [And83], Clifford and Rao [CR87] have proposed a theoretical model of time granularities underlying a total order which introduces most of the features of the set-theoretic approach. They introduce granularity conversions along a *finer than* relation on this total order (e.g. hours are finer than days), defining a time granularity in terms of its next finer time granularity. Wiederhold, Jajodia, and Litwin [WJL91] have advanced this model by adding some specific semantics. These semantics allow for temporal comparisons regarding to the complete ordering of time granularities. Their model allows for handling mixed time granularities (i.e. temporal aspects stated in more than one time granularity). In sequencing papers, Wang and others [WJS95, Wan95] have generalized this totally ordered set of time granularities to a partially ordered one allowing for finer and coarser relations between the time granularities comprised in a particular finite set of partially ordered time granularities. In addition to this finer than relationship, several further relationships between time granularities have been thoroughly investigated. A collection of such relations is given in [BJW00]. Furthermore, a calendar algebra [NWJ02] has been proposed to symbolically represent time granularities as expressions form other (previously defined time granularities) using algebraic operators.

Time Granularities. According to the temporal structure proposed for time granularities in [CR87] which has been generalized in [BJW00], a time granularity G is a function from an index set into the power set over a linear (dense or discrete) time line. This function is order preserving and conservating. The elements of a time granularity do not overlap. The *origin* of a time granularity is the smallest element according to the order relation and it is index by 0. The *image* of a time granularity is the union of its granules, and the *extend* of a time granularity is the smallest interval of the underlying time line that contains the time granularity's image.

For time granularities G and H, Clifford and Rao [CR87] have introduced a *finer than* relation as follows: a time granularity G is finer than a time granularity H iff $\forall i \in \mathbb{Z} \exists j \in \mathbb{Z}$ such that $G(i)$ is a subset of $H(j)$. In [BJW00], several additional relations between time granularities have been suggested:

$G \trianglelefteq H$ iff $\forall j \exists S \;\; H(j) = \bigcup_{i \in S} G(i)$	*groups into*
$G \sqsubseteq H$ iff $\forall i \exists j$ such that $G(i) = H(i)$	*sub-granularity of*
$G \trianglelefteq H$ and $G \preceq H$	*partitions*
$G \;\hat{}\!\subseteq H$ iff the image of G is contained in the image of H	*covered by*
$G \trianglelefteq H$ and their exists a periodic repetition of the grouping pattern of granules of G into granules of H	*groups periodically into*
$G \leftrightarrow H$ iff $\exists k \forall i$ such that $G(i) = H(i+k)$	*shift-equivalent*

Apart from the relation for shift-equivalence, all those relations define time granularities in a way such that G is a "more precise" time granularity than H.

Calendars are defined in this framework as sets of time granularities that contain a time granularity G such that each of the time granularities groups into the time granularity G [BJW00].

In [BJW00], specific sets of time granularities are defined within which the granules of one time granularity can be converted into granules of another time granularity. Note that in this framework, conversions are not always defined: in particular, a conversion from a granule into a granule of a coarser time granularity is only defined, if the corresponding time granularities are related according to the "finer than" relationship [CR87]. A conversion from a granule into granules of a finer time granularity is only defined, if the corresponding time granularities are related according to the "group into" relationship [BJW00]. Thus, a universally valid solution to the problem of time granularity conversion is not provided. The problem of time granularity conversion in the metric, point-based approach to time granularities is discussed in [FM01]. In particular, in [BJW00], conversions are only supported for aligned time granularities, i.e. time granularities which are related according to the "finer than" relationship. Non-aligned time granularity systems have been considered for example in [Sno95].

In [NWJ02], a formal algebra to always define coarser time granularities from finer ones is proposed. The suggested algebraic operators directly refer to and relay on the afore mentioned relations between time granularities. The afore mentioned problem of time granularity conversion also appears within this calendar algebra.

Constraint Solving and Query Answering. An extension of the relational data model of relational database systems to handle time granularities is proposed in [WJS95, Wan95]. In principle, the proposed data model is an extension of the relational data model where each tuple is timestamped under a time granularity. The goal of this work is to take into account possible time granularity mismatch in the context of federated databases and data warehouses. In [BJW00] this work has been further developed for answering queries in relational databases with time granularities.

In [BJW00], a metric, point-based Temporal Constraint Satisfaction Problem (TCSP) under time granularities is defined in the Horn DLR constraint framework, extended with time granularity. In this TCSP (that can be modeled by a directed graph), the nodes (i.e. the variables) model time points and the arcs are labeled by integer intervals and time granularities. An arc denotes the metric (temporal) distance between two points. A pair of variables only satisfies a constraint (i.e. the arc between those variables), if (up and down) conversions between each of the points and the arc's time granularity exist. In particular, this approach only applies for aligned (according to the "finer than" relationship) systems of time granularities which is not always the case, as the "week/month" example illustrates it. In [BJW00], the authors provide an arc-consistent algorithm for consistency checking when the time granularities are periodic with respect to some finer time granularity. An incomplete algorithm is provided by propagating constraints expressed under the same time granularity and then converting the new values into the other time granularities.

Qualitative Time Granularity. Time granularity operators for qualitative time representation have been first proposed in [Euz93, Euz95]. Those time granularity operators are defined in the context of relational algebras, applicable to both point and interval algebras. In principle, a qualitative algebra [All83, VKvB90] is initially augmented with a neighborhood structure [Fre92]. Subsequently, an interval algebra with qualitative granularity in terms of conversions of the relations according to some coarser and/or finer granularity is constructed [Hir96]. In [Euz93, Euz95], Euzenat provides with a framework where situations are described by a set of possible relationships holding between temporal primitives (i.e. points or intervals). In particular, each layer of the qualitative and relational language represents a situation in the unalternated language (of the chosen) relational algebra. The qualitative time granularities are defined by a set of operators. The operators are used to convert the representation of a (whole) situation into a finer or coarser representation of the same situation (as a whole). The operators apply to the relationships that might hold between the temporal primitives (in some situation). The operators transform each of the relationships that hold in a situation into a coarser (using some upward conversion) or a finer (using some downward conversion) time granularity.

One of the important aspects of the work by Euzenat [Euz01] is the definition of a set of (generic) constraints which should be satisfied by any system of granularity conversion operators:

formal definition	**name**
$r \in \rightarrow r$	*self-conversion*
$\forall r, \forall r', r'', \exists r_1, \dots r_n :$	*neighborhood compatibility*
$r_1 = r', r_n = r''$ and $\forall i \in \{1,..,n-1\} N(r_i, r_{i+1})$	
$\rightarrow r^{-1} = (\rightarrow r)^{-1}$	*conversion reciprocity distributivity*
$r \in \bigcap_{r' \in \uparrow r} \downarrow r$ and $r \in \bigcap_{r' \in \downarrow r} \uparrow r$	*inverse compatibility*
$\uparrow\uparrow r = \uparrow r$ and $\downarrow\downarrow r = \downarrow r$	*idempotence*
$\rightarrow r = \Leftarrow\rightarrow\Rightarrow r$ and $\rightarrow r = \Rightarrow\rightarrow\Leftarrow r$	*representation independence*

where $\uparrow$ denotes upward conversion, $\downarrow$ denotes downward conversion, $\rightarrow$ denotes a neutral operator (if a time granularity change between two layers is addressed, but it is not necessary to know which one is the coarser), $\Leftarrow$ and $\Rightarrow$ denotes an interval relational space,

x and y denote time primitives, and r denotes a relation between time primitives. *Self-conversion* means that, independed of the conversion, a relationship must belong to its own conversion. *Neighborhood compatibility* contains the conversion of a relation to form a conceptual neighborhood. *Conversion reciprocity distributivity* denotes symmetry. It states that the conversion of the relation between a first object and a second object must be the reciprocal of the conversion of the relation between the second one and the first one. *Inverse compatibility* states that the conversion operators are consistent with each other. Finally, *representation independence* states that the conversion must not dependent on the representation of the temporal primitive, i.e. whether a relation between points or intervals is converted.

From these constraints, the possible conversion operators for a particular relational algebra for points and/or intervals such as the algebras proposed in [Vil82, All83, VKvB90] can be generated. Examples of such relational algebras with conversion operators can be found in [Euz01].

2.1.4.2 Logic-based Time Granularity Systems

Hobbs [Hob85] has introduced a very general logic-based concept of granularity. He has considered predicates extracted from a global theory (given in some logic formalism) which are relevant to some present situation described at a specific granularity. Extracting relevant predicates leads to an indistinguishability relation: variables are indistinguishable to the situation at hand if no relevant predicates distinguish between them. In this concept of granularity, a granularity is characterized *after* some theory has been defined. This work has been enhanced by Greer and McCalla [GM89].

Different frameworks to deal with time granularities in Linear Temporal Logics (LTL) has been suggested: for example, in [FM94], Fiadeiro and Maibaum represent each granularity in the same (classical) temporal logic. Conversions are defined between these different representations. Note that representations with mixed time granularities are not possible in this framework. Combi, Franceschet and Peron [CFP04] have defined time granularities as models of LTL formulas using appropriate propositional symbols to mark endpoints of granules. Demri [Dem04] has generalized the framework of Combi et al. and the automata-based approach (cf. Section 2.1.4.3) of Montanari and Dal Lago [LM01] to a first-order language. This language allows for defining time granularities by means of periodic constraints specified in a first-order logic.

In [CEMP93, Mon96], Montanari and others have proposed a metric temporal logic with granularities. The authors of this work show how to extend syntax and semantics of temporal logics to cope with metric temporal properties possibly expressed at different time granularity. This work results in a metric and layered temporal logic [Mon96]. This layered logic is based on the idea of a logic of positions, i.e. a topological or metric logic [RG68, RU71]. Such a metric logic is then generalized to a many-layered metric temporal logic, embedding the notion of time granularity. The main features of the logic proposed in [Mon96] are three different operators: a contextual operator (to associate different granularities with different formulas), a displacement operator (to move within a given granularity), and a projection operator (to move across granularities).

Alternative temporal logics with time granularities are proposed in [Mon96, MP96, MPP99]. The logics introduced in those works define various theories of time granularity as extensions of monadic second-order theories. They provide with suitable restrictions to such languages for time granularities to get decidable temporal logics with time granularities.

Yet another alternative has been proposed in [OG98]. The authors of this work have integrated set-theoretic operators for defining time granularities in terms of predicates into a modal first-order language with modality operators like "next" with time granularity.

2.1.4.3 Automata-based Time Granularity Systems

Wijsen [Wij00] has proposed a string-based approach to represent time granularities, in particular, to represent infinite periodic time granularities like days or weeks. Such time granularities are modeled as (infinite) words over an alphabet that consists of three symbols: ■ denoting fillers, □ denoting gaps, and ≀ denoting separators. A granule is constructed form fillers and gaps and delimited by a separator. A finite set of granules is used to describe the repeating pattern of a time granularity. Periodic time granularities can be identified with ultimately periodic strings, and they can be finitely represented by specifying a (empty) finite prefix and a finite repeating pattern in terms of granules modeled by combination of the three symbols ■, □,and ≀.

This string-based approach has been further extended by Montanari and Dal Lago [LM01] to an automata-based approach: a specific subclass of Büchi automata has been introduced that recognizes languages only consisting of periodic words (described by the afore mentioned symbols of Wijsen's string-based approach).

Note: Common to all the afore mentioned approaches to time granularities is that they disregard irregularities specific to calendar systems like leap seconds and moon and sun cycles, i.e. no means to handle characteristics of "real-life" calendars within a framework for time granularities are provided. Comparing problems of time granularities of different calendars, for example when integrating the Gregorian and the Islamic calendars where Islamic days begin at sunset[4] while Gregorian days begin at midnight are not sufficiently solved, as well. Furthermore, none of the afore mentioned approaches define a way for specifying arbitrary date formats for the granules of any time granularity.

Furthermore, except for the temporal database query language TSQL2 [Sno95] (which is based on the metric, point-based time granularity model) none of the previously surveyed approaches to time granularities have been applied and implemented in an information systems.

[4]Note that sunset cannot be defined in terms of a fixed hour; instead, sunset depends on the day in the year.

2.2 Calendric Computations

The afore surveyed time formalisms (possibly embedding a concept of time granularity) provide with symbolic approaches to time and calendars based on logics, relational algebras, or set-theory. A rather practical approach to real-time calendar systems is made in [DR01]. In their work, Dershowitz and Reingold provide with a large collection and implementation of 15 calendars in use today (or which have been used in the past); among others, the Gregorian, the Islamic, the Hebrew, the old and modern Hindu, and the Mayan calendars along with both all their particularities like leap years and moon cycles and relevant holidays related.

As an intermediate device for converting from one calendar to another, Reingold and Dershowitz have chosen a day numbering: Julian day numbers are used to specify days in any calendar with midnight at the onset of Monday, January 1 in year 1 (Gregorian) as fixed date indexed by 1, counting forward day by day from there. Note that this date corresponds to Monday, January 3 in year 1 in the Julian calendar. For each calendar C, a function `fixed_form_C(`$C - date$`)` is implemented that converts the date $C - date$ of the calendar C to its corresponding Julian day number. The function `C_form_fixed(`$date$`)` does the inverse computation. These two functions are the backbones for conversions between dates specified in any of the calendars implemented.

Calendar dates are represented by lists of components (in descending order significance) having in most cases the form "year month day" where each component is represented by an integer. Each component can be selected using the corresponding function.

An implementation of the calendars along with the conversions between them presented in the book of Dershowitz and Reingold [DR01] are provided as a Java-Applet which is available at `http://emr.cs.iit.edu/home/reingold/calendar-book/Calendrica.html`.

Further, in deep, algorithms for leap year calculations, including generalizations can be found in [HR04] and algorithms for calculating Hebrew dates in [DR04], completing the work made in [DR01].

This short section on calendric computations is provided within this chapter to remind the reader that particularities and irregularities of real-life calendar systems as they appear with Hebrew months or leap second insertions is an important aspects that should be considered in approaches to data modeling and reasoning with calendars.

2.3 Web and Semantic Web Formalisms and Applications

Section 2.1 has reviewed specific methods and formalisms on temporal knowledge representation and reasoning proposed in the research areas of Artificial Intelligence and Database Systems.

Today, several Web and Semantic Web languages exist, developed to enrich data and information in Web pages and Web sites, making such data machine-readable and machine-processable. XML [W3C04a], a Web formalism for data and knowledge modeling is the basic principle all further languages developed for Web and Semantic Web applications and Web services directed by the World Wide Web Consortium (W3C) (cf. `http://www.w3.org`) depend on such as XML Schema [W3C01], RDF [W3C04c], and OWL [W3C04b].

This section reviews (generic) data modeling languages for the Web and the Semantic Web. In particular, this section investigates to what extend data modeling and reasoning with time and calendars is supported by existing Web and Semantic Web languages. Applications of calendar and time data modeling and reasoning using languages for calendar Web services, calendar computation systems, and temporal and active Web systems are briefly surveyed. Additionally, a section on internationalization and localization efforts on the Web and the Semantic Web is given.

2.3.1 Data Type Definition Languages

Essentially two different data type definition languages exist for XML, XML-DTD [W3C04a] and XML Schema [W3C01]: XML and XML-DTD provide a surface syntax for structured documents, but imposes no semantic constraints on the meaning of these documents. XML Schema is a language for restricting the structure of XML documents. XML Schema extends XML with datatypes.

2.3.1.1 XML DTD

Generic markup languages such as SGML (Standard Generalize Markup Language), conceived for modeling texts and XML (eXtensible Markup Language) [W3C04a], conceived for modeling data and knowledge on the (Semantic) Web are used to specify structures of documents and data. Such markup languages allow for data modeling independent from its usage and from possible presentation devices. Thus, it is possible to meaningfully model the structure of data needed for some specific applications. Layout particularities for the structural elements specified, for example in XML, can be defined using so-called XML-based *style-sheet* languages like CSS or XSL-FO.

Such generic markup languages are based on so-called *semi-structured* data models [ABS00]. In particular, they are much richer than the relational data model [Cod70], because they allow optional elements.

A simple framework for typing XML is provided with XML-DTD [W3C04a]. In principle, XML-DTD is a semantically weak formalisms for declaring structural types for XML documents. No support for data types of element contents exists. In particular, XML-DTD does not support any means for typing time and calendar data to appear in XML documents.

2.3.1.2 XML Schema

XML Schema [W3C01] is a much richer data type definition language for XML than XML-DTD. Beyond a rich set of structural types, it provides a means to define structure, content and semantics of XML documents. And what is relevant for modeling time and calendars, XML Schema provides predefined data types for specifying element contents, i.e. the data represented in some Web page or Web site. In the following, XML Schema's possibilities for representing times and dates are summarized.

XML Schema provides a considerably large set of predefined time and date types dedicated to the Gregorian calendar. The supported date format is ISO-8601 standard [ISO00]. Time zones and leap years are taken into account. XML Schema provide a means to represent qualitative and metric points and intervals which may be either dates and times like "2005-02-10" or periodic like "day". The data type `dateTime` is intended to specify points, and the data type `duration` is intended to specify intervals. Any point or interval is represented by specifying Gregorian `dateTime` coordinates. For example, a point represented by a Gregorian date is specified by its year, month, and day coordinates. XML Schema allows for adding some duration to a Gregorian time or date. Several data types are supported to represent sub-parts of a time or date format, for example the year coordinate from a Gregorian date. Furthermore, data types are provided to specify annually, monthly, etc. periodic points. Note that the concept of time granularity (cf. Section 2.1.4) is inherent to any of the calendric data types supported in XML Schema, i.e. XML Schema provide no means for abstracting dates and times form their related time granularity.

XML Schema does neither support time and calendar data types that go beyond the Gregorian calendar nor date formats other than those in ISO-8601 standard. Furthermore, no means for specifying user-defined temporal data types are supported.

Note: In XML Schema user-defined data type definitions are restricted such that the user can specify a restricted pattern for a pre-defined XML Schema primitive data type in terms of a regular expressions as illustrated in Example 2.2.

Example 2.2 *In XML Schema, a datatype could be defined that corresponds to the SQL datatype Year-Month interval that requires a four digit year field and a two digit month field but required all other fields to be unspecified. This datatype could be defined as follows, restricting XML Schema's primitive data type* `duration` *by specifying a regular expression.*

```
<simpleType name='SQL-Year-Month-Interval'>
    <restriction base='duration'>
      <pattern value='P\p{Nd}{4}Y\p{Nd}{2}M'/>
    </restriction>
</simpleType>
```

2.3.2 Ontology Languages

Data modeling languages such as XML are used to syntactically structure Web documents. XML Schema is a language for restricting the structure of such XML documents. XML Schema additionally extends XML with datatypes. But such languages fall short in issues concerning the semantics of documents needed, for example, to process the content of information. To overcome such kinds of problems, ontologies designed in (Web-based) ontology languages are used to give an explicit and unambiguous description and a common understanding to a certain domain of knowledge such as time, calendars, and location. Ontologies seam a promising approach to support communication between humans as well as machine-based processing of data. Therefore, ontology languages became more and more popular in the last decades in research areas such as knowledge representation and reasoning, natural language processing and information retrieval and extraction, and, nowadays, in particular, the Semantic Web.

In [Gru93], ontologies are defined as "explicit specifications of a conceptualization" where conceptualization means a (abstract) model of the world. Since relevant concepts in such a model are defined including their related properties and instances, Description Logics [BCM03] are the favorite formal languages to design ontology languages. The basic entities of Description Logics are unary predicates, called *concepts*, and binary predicates, called *roles*. A specific Description Logic is mainly characterized by the provided *constructors* to build compound concepts and roles from atomic ones. Usually, a Description Logic additionally has a so-called *T-Box*. A T-Box is a finite set of (named) concepts, roles, and constructors. Description Logic systems mainly provide two different reasoning tasks: *subsumption* and *satisfiability*; the former, to determine subconcept-superconcept relationships, thus the taxonomy of some T-Box considered, and the latter, to test whether a given concept can ever be instantiated or not. Depending on the expressiveness of the underlying Description Logic, the complexity of these algorithms vary. Most of these algorithms are tableau-based reasoning algorithms.

In the Semantic Web, ontologies are intended to defined terminological consensus for various domains such as Web services. For this purpose, ontology languages such as RDF [W3C04c], and OWL [W3C04b] have been developed. Those languages are essentially based on Description Logic systems with corresponding reasoning methods. OWL facilitates greater machine-based processing of Web content than that supported by XML and XML Schema as well as that supported by RDF. OWL is based on the Description Logic $\mathcal{SHIQ}$ [HST99]. $\mathcal{SHIQ}$ is an extension of the basic Description Logic $\mathcal{ALC}$ with means such as transitive and inverse roles, number restrictions, and role inclusions. Existing reasoners for $\mathcal{SHIQ}$ are FaCT [Hor98] and RACER [HM01]. Both are EXPTIME-complete [Tob01].

2.3.2.1 RDF: Resource Description Framework

RDF [W3C04c] is an XML-based data model with a simple semantics of directed binary graphs to identify resources and relations between them.

RDF can be used to model information about Web resources and other entities that should be identified such as meta data about Web resources, informations about Web documents or about specifications, items available from some online shopping facility, and some user's preferences. RDF is particularly intend for situations in which such information needs to be processed and/or exchanged by computer programs or applications.

RDF follows up ideas for conceptual graphs, logic-based knowledge representation, frames, and relational databases. In particular, RDF bases on the idea of identifying entities like Web documents or resources using Web identifiers, so-called *Uniform Resource Identifiers (URIs)*. URIs might also refer to entities not directly retrievable on the Web. Such entities, denoted as *subjects*, are described by properties, denoted as *predicates*, which might have values, denoted as *objects*. That is, RDF is used to make statements about entities like Web documents by determining (1) what is described by the statement, i.e. the subject, (2) what is the property of the entity the statement describes, i.e. the predicate, and (3) what is the value of this property for the entity described by the statement, i.e. the object. To make such statements machine-processable, each subject, predicate, and object is identified by a URI; more precisely by a URI reference, i.e. a URI and a fragment at the end of the URI. XML is used to represent these statements and to exchange them between machines.

RDF statements can be represented by a simple graph model: subjects and objects are represented by nodes and predicates are represented by directed arcs, each from a subject node to its corresponding object node. Groups of statements are represented by corresponding groups of nodes and arcs. Note that RDF statements are similar to some other (simple) formats for recording information, in particular:

- entries in a simple record or catalog listing, describing the resource in a data processing system,
- rows in a simple relational database, and
- simple assertions in formal logic.

To give certain additional information to literals such as a price or the age of a person in an RDF statement, RDF provides with some means to associate a datatype with such a literal. This is achieved by pairing a string with a URI reference that identifies a particular datatype, resulting in a single (typed) literal node in the corresponding RDF graph. RDF does not provide any built-in set of datatypes. It rather defines a means to relate an externally defined datatype to the corresponding RDF node. Not any kind of datatype is allowed in this framework: RDF datatype concepts are based on a conceptual framework from XML Schema datatypes including some additional restrictions. In particular, the structure of the elements of such data types are limited to character strings that can be described by regular expressions. Example 2.3 illustrates the use of RDF (using typed literals) to specify the creation date of a Web page.

Example 2.3 *The RDF/XML fragment*

```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [<!ENTITY xsd "http://www.w3.org/2001/XMLSchema#">]>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:exterms="http://www.example.org/terms/">

  <rdf:Description rdf:about="http://www.example.org/index.html">
```

```
    <exterms:creation-date rdf:datatype="&xsd;date">2005-06-26
    </exterms:creation-date>
  </rdf:Description>

</rdf:RDF>
```

illustrates how the Web document with URI ***http://www.example.org/index.html*** *can be annotated with a creation date whose value is "June 26, 2005". This information is modeled by the following RDF triple with a typed literal (of XML Schema type* `date`*):*

```
ex:index.html  exterms:creation-date  "2005-06-26"^^xsd:date
```

where `ex:index.html` *is the RDF subject,* `exterms:creation-date` *the RDF predicate, and* `"2005-06-26"^^xsd:date` *the RDF object.*

Beyond the rather limited RDF datatype concept, RDF has some further limitations: RDF only recognizes full URI references, i.e. RDF neither has knowledge about their structure nor assumes any relationships between them. RDF directly represents only binary relationships. Explicitly modeling n-ary relationships thus becomes rather complicate in RDF.

Additionally to RDF, RDF Schema is provided. RDF Schema is a vocabulary for describing properties and classes of RDF resources. RDF Schema has a semantics for generalization-hierarchies of such properties and classes. RDF Schema provides the facilities needed to define the vocabularies (terms) which are intend to be used in RDF statements. Concepts are provided to indicate that such vocabularies are used to describe specific classes of resources and that those classes will be described by specific properties. RDF Schema provides means to indicate which classes and properties are expected to be used together.

2.3.2.2 OWL: Ontology Web Language

OWL [W3C04b] is more expressive than RDF Schema. OWL is an ontology language to formally describe the meaning of terminology used in Web documents such that useful reasoning tasks can be performed on such documents. OWL provides additional language constructs to those provided with RDF Schema. The language constructs are used to describe properties and classes, for example relations between classes (e.g. disjointness), cardinality (e.g. "exactly one"), equality, richer typing of properties, characteristics of properties (e.g. symmetry), and enumerated classes. OWL provides with a formal semantics. OWL facilitates greater machine interpretability of Web content than that supported by XML, RDF, and RDF Schema.

OWL has three decreasingly-expressive sub-languages: OWL Full, OWL DL, and OWL Lite. OWL Full is upward compatible with RDF, i.e. every valid OWL Full document is an RDF document. OWL Full is intractable and undecidable. OWL Full provides with meta-modeling facilities of RDF Schema. OWL Full can be viewed as an extension of

RDF, while OWL Lite and OWL DL can be viewed as extensions of a restricted view of RDF. OWL DL is closely related to Description Logics, in particular to the logic $\mathcal{SHIQ}$ [HST99]. It is computationally complete and decidable. OWL Lite's intended use concerns situations where only a simple class hierarchy and simple constraints are needed. Due to its restricted expressiveness, OWL Lite provides with less time consumption reasoning support than OWL DL[5]. The development of OWL has been influenced by DAML+Oil, Description Logics, frame paradigms, and RDF/RDF-Schema.

OWL has three components: *classes*, *individuals* (also called instants), and *properties* (called roles, relations, or attributes in other formalisms). Classes are used to describe sets of individuals that belong together and share some properties. Classes can be declared to be disjoint from each other or by Boolean combination in terms of union, intersection, or complement. OWL Full and OWL DL additionally allow for describing a class by enumeration of the individuals that make up the class. An individual is an instance of a class. Classes may be declared as equivalent to each other and/or in a subclass-superclass hierarchy. Individuals may be declared as equivalent or not equivalent to each other. Note that OWL Full allows for declaring complex class descriptions consisting of enumerated classes, property restrictions, and Boolean combinations. Properties are binary relations to state relationships between individuals or from individuals to data values that can be, as it is the case for classes, equivalent to each other and/or in a subproperty-superproperty hierarchy. Properties may have a domain that limits the individuals to which the property can be applied and a range that limits the individuals that the property may have as its value. Properties may be characterized being inverse, transitive, symmetric, functional (i.e. unique value), and inverse functional to each other. Furthermore, OWL allows for specifying data type properties in accordance with XML Schema data types. In particular, OWL uses RDF mechanisms for data values, the OWL built-in datatypes are taken form the XML Schema datatypes. Example 2.4 illustrates the use of XML Schema data types in OWL.

In OWL Lite cardinality is restricted to the values 0 and 1.

Example 2.4 *OWL distinguishes properties according to whether they relate individuals to individuals (object properties) or individuals to datatypes (datatype properties). Datatype properties may range over RDF literals or simple types defined in accordance with XML Schema datatypes, e.g. the OWL class* `VintageYear` *comes with a datatype property ranging over year values:*

```
<owl:Class rdf:ID="VintageYear" />

<owl:DatatypeProperty rdf:ID="yearValue">
  <rdfs:domain rdf:resource="#VintageYear" />
  <rdfs:range  rdf:resource="&xsd;gYear"/>
</owl:DatatypeProperty>
```

[5]Recall that reasoners like RACER or FaCT provided for OWL DL are ExpTime-complete.

At least two reasoners, FaCT [Hor98] and RACER [HM01] (which are both EXPTIME-complete [Tob01]), are implemented with OWL. Both support with a subsumption algorithm (determining subclass-superclass relationships), and satisfiability checks.

2.3.2.3 Applications: Time Ontologies

Most existing and emerging time formalisms for Web and Semantic Web applications are ontology-based. Such time formalisms are based on a generic modeling and reasoning mechanism: they use a general modeling and reasoning mechanism for temporal and calendric data which is based on description logic systems, i.e. they are modeled using an ontology language such as OWL.

Various time ontologies, implemented in different ontology languages are available both from academic (e.g. the Knowledge Interchange Format KIF [GF92], Ontolingua [Kno]) and industrial sources (e.g. Cycorp (cf. `http://www.cyc.com`)), varying in expressive power, formality, and/or coherence. Well-known time ontologies for the Web and the Semantic Web are the DAML time ontology [DAR02] and time in OWL-S [PH04]. Furthermore, as time is a necessary notion to deal with in several domains like locations, events, enterprises and organizations, and natural language text, time has been modeled as a part of various common-sense ontologies like the Cyc ontology [Cyc97].

Time Ontology in KIF. The time ontology in KIF is developed at the Stanford Knowledge Systems Laboratory (cf. `http://www.ksl-svc.stanford.edu`). This temporal ontology is based on a simple point-based time model. Intervals are modeled by their ending points. Thus, they differ between open, half-open, and closed intervals. The basic interval to interval relations are modeled, for example a time interval ?tr1 preceeds a time interval ?tr2 on ending points (which are themselves considered as intervals):

```
(<= (before ?ti1 ?ti2)
     (and (t-interval.limits ?ti1 ?start1 ?end1)
          (t-interval.limits ?ti2 ?start2 ?end2)
          (t<  ?end1 ?start2)))
```

The time ontology is dedicated to the Gregorian calendar: different “layers” are used to model the various Gregorian calendar units in terms of KIF classes. For example, “Calendar-Day-1” through “Calendar-Day-31” is an instance of “Calendar-Month-January”, or “Wednesday” is an instance of “Regular-Non-Convex-Time-Interval”. Additionally, a class “Time-Quantity” to represent durations in time is modeled with a relation “Magnitude” to convert between different time quantities. However, problems such as converting the duration of a (Gregorian) month to that of days, yielding in ambiguity is not solved. Facts about the time quantities of Gregorian calendar units are stored in an additional library.

DAML time ontology. The DAML time ontology [DAR02] has been designed for expressing time-related properties of Web services. The different calendar units of the Gregorian calendar and the different time zones have been axiomatized, i.e. as it is the case for the previously addressed time ontology in KIF, the DAML time ontology is restricted to calendar units of the Gregorian calendar. For example weekdays and weekend days are modeled as follows in the DAML time ontology:

```
<axiom id = "4.3-12">
    weekday(y,x) <-> [Monday(y,x) v Tuesday(y,x) v Wednesday(y,x) v
                      Thursday(y,x) v Friday(y,x)]
</axiom>
<axiom id = "4.3-13">
    weekendday(y,x) <-> [Saturday(y,x) v Sunday(y,x)]
</axiom>
```

The DAML time ontology is based on a point-based time model, axiomatizing temporal relations on points and intervals which are modeled by their ending points. Metric axioms are additionally provided. The axiomatization of this ontology is done actually for the same point-based time model as that of the KIF time ontology.

Note: The DAML time ontology constitutes a barrier to its use due to its complexity.

Time in OWL-S. OWL-S is an OWL-based Web service ontology to provide Web services with constructs for unambiguously describing properties corresponding to time and calendars as well as capabilities of Web services. Since most real-world Web services involve some notion of time, a subset of the DAML time ontology has been integrated into OWL-S. OWL-S allows for integrating XML Schema data types.

Time in OWL-S [PH04] is designed as a sub-ontology of the DAML time ontology, axiomatizing basic temporal concepts and relations, in particular, points, intervals (which are modeled by their ending points), durations, events, dates and times. As it is the case for the DAML time ontology, time in OWL-S is dedicated to Gregorian calendar units and time zones. It is essentially a subset of the complete DAML time ontology, in particular to overcome the computational complexity of this time ontology.

Socio-Cultural Time Ontology. Within the research unit the work present in this thesis has been developed, a socio-cultural time ontology [Oez04] has been designed. This ontology provides with an approach to extend existing time ontologies like time in OWL-S [PH04] and the DAML time ontology [DAR02] dedicated merely to the Gregorian calendar with ontological concepts dedicated to cultural aspects of time like the different notions of weekend days in Germany, Turkey, and Israel. This ontology provides with several classes of time periods, for example according to nation (e.g. US, Turkey) or to business (e.g. education). A class such as “time periods of education and business shorter than one day” has for example individuals such as “meeting hour”, “exhibition dates”, or “office hours” which might have properties such as “periodic”.

This socio-cultural time ontology has been modeled in OWL.

2.3.3 Internationalization

Recently, the World Wide Web Consortium W3C (cf. `http://www.w3.org`) has started the so-called “Internationalization Activity” (cf. `http://www.w3.org/International`). This activity is intended to ensure that Web and Semantic Web technologies like formats, languages, and protocols, and, in particular, those technologies developed by the W3C are usable world-wide in all languages and in all writing systems. The W3C Internationalization Activity aims at:

1. means for using Web technologies world-wide according to languages, scripts, regions, and various aspects of cultures,

2. making internationalization aspects of Web technologies better understood, and widely and consistently used, and

3. providing with a set of elements and attributes for data type definition languages like XML Schema to support internationalization and localization of Web documents, Web and Semantic Web applications, and Web services.

These goals require work on different (technical) aspects such as character sets and character encoding, language, resource identifiers, document structure and meta data, and locale specific data according to the layout as well as the content of Web and Semantic Web documents and applications. Locale specific data also addresses problems concerning the display (and interpretation) of date formats according to its cultural context. For example, the date “12/02/2005” is interpreted in France as 12th February 2005 while it is interpreted as 2nd December 2005 in the US. Yet, no uniform solution to such kinds of problems is provided. However, the internationalization activity recommends to use the HTTP Accept-Language header to determine the locale (which contains date and time information) of the user.

Note: Locale specific data does not contain information on temporal and calendric data that goes beyond date format specifications. Problems that have varying common-sense understanding such as “weekend days” which refer to Saturdays and Sundays in (western) European countries while they refer to Fridays and Saturdays in Israel. Such cultural contexts of temporal and calendric data go far beyond date and time specifications. However, such data frequently appear in Web documents and Web and Semantic Web applications, thus, such temporal and calendric contexts are worth to be considered, aiming at both, internationalization and localization in the (Semantic) Web.

A practical impact of internationalization and localization efforts addresses the way internationalization options are exposed in Web services regarding their definitions, descriptions, messages, and discovery mechanisms [W3C04d].

2.3.4 Web Services for Calendric Data

This short section addresses applicability of ontology markup for Web services involving, in particular, calendric and temporal data.

In general, a Web service is a (agent-based) software systems to support system interoperability across the Web. A Web service has a machine-processable interface and interacts with other systems using specific messages. XML-based formalisms like WSDL [W3C05] for modeling and describing such interfaces and SOAP [W3C03] for exchanging structured and typed information, i.e. messages between peers have been developed. Agents usually read and process data annotated by some ontology described in a language like RDF or OWL.

In what follows, the calendar agent RCal [PSS02, SSP02, Sin03] and the calendar Web server WebCal [Ohl03] are briefly considered as examples of Web services operating with calendric and temporal data.

2.3.4.1 Web-based Meeting Scheduler

Research on agent programming and robotics has yield several proposals for automated meeting schedulers in the 1990ies [KM93, MCF$^+$94, GLS96, SD98]. Meeting and appointment scheduling is a well-known temporal reasoning problem that is often time consuming, involving several parties and (time) constraints that must be fulfilled. Such systems are usually realized as multi-agent systems. In such systems, any user has a personal agent that can cooperate with other agents. However, most of these systems suffer from a lack of flexibility and power since most of them are stand-alone systems.

Within the Semantic Web, research on Web services implemented as multi-agent systems has gained in popularity. An example of a Web-based multi-agent system that supports such appointment scheduling tasks is the RETSINA (Reusable Environment for Task-Structured Intelligent Network Agents) Calendar Agent, short RCal, proposed in [PSS02, SSP02, Sin03]. RCal works symbiotically with Microsoft Outlook 2000 and the Semantics Web (as long as events and schedules are annotated by some ontology markup). It provides with a lot of features on appointments and schedules, referring to resources on Web pages or using URIs.

Let us turn attention to the reasoning capabilities of RCal: RCal is designed as a distributed meeting scheduler to gather and reason about appointments and schedules. Schedules are collections of such appointments. In addition to several attributes like location, subject, and a list of required attendees, each appointment is associated with an event. Each such event is represented by a start time, an end time, and a duration. Each start time and end time, in turn, is represented by a (Gregorian) date and time in a twelve hour format with am/pm identifier and an optional time zone identifier. An event's duration is represented in terms of minutes. Events may be declared as either singular or daily (resp. weekly, monthly, yearly) recurring. The two temporal queries that can be answered over such events are (1) checking whether a particular time is free, and (2) finding a free time slot of a given duration. Those queries take, of course, account of recurring events. However, both the representation of temporal and calendric data and the scheduling problems that can be solved by RCal agents are very limited. Both features are predefined within the system. No means for user-defined events and/or queries are provided.

RCal can browse and download functionality for schedules marked up using the Hybrid RDF Calendar Ontology, short iCal (cf. `http://ilrt.org/discovery/2001/06/schemas/icalfull/hybrid.rdf`). In order to browse events and schedules annotated using other ontologies, RCal is connected to the DMA2ICal markup translation agent [SSP02] which provides a translation service (based on DAML-S) for translating into markup using the iCal ontology.

RCal works synergistically with Microsoft Outlook 2000. It retrieves appointments form Outlook. The information from some user's Outlook is used while negotiating (based on the Contract Net Protocol [Smi80]) and reasoning about available meeting times with other RCal agents.

2.3.4.2 Calendar Web Server

Numerous calendar converters are provided on the Web. Most of this work, e.g. Kalender-Rechner (cf. `http://www.nabkal.de`), Calendar Converter (cf. `http://www.fourmilab.ch/documents/calendar`) as well as calendar conversions between particular calendars like the Western-Chinese Calendar Converter (cf. `http://www.mandarintools.com/calendar.html`) is very similar to the work that has been done by Dershowitz and Reingold [DR01], performing calendar date conversions using Julian day numbers. Those approaches mainly vary in the number of implemented calendars, additional features such as computing distances between dates and shifting dates by a number of days, algorithms computing Easter and moon cycles, week dates, and conversion to Unix Time. A very good survey on computation tools and information material on calendars and time on the Web is collected in the "Calendar and Time Database" (cf. `http://www.greatdreams.com/calndr2.htm`). This collection includes a lot of links concerning additional information and computation tools of calendars and time like time zones and holidays. Additionally, information and computation tools on astrology and astronomy basics are collected. However, none of these calendar converters provide any means for machine-based reading and processing of the calendric data, thus, none of these converters making calendar conversion usable for Web services that need to refer to such data and conversions.

A system that provides (function-based) calendric computations in addition to date conversion is WebCal [Ohl03]. WebCal is a C++ library that is conceived as a Web server. WebCal provides with calendric calculations on convex (fuzzy) intervals in the system's reference time unit. The system's reference time unit is second. The calendric calculations are performed by a variety of pre-defined functions over such intervals, among them Allen's interval relations [All83]. Since computations of such functions (which are applied to Gregorian dates and times) are performed on such intervals of seconds, the WebCal system converts each date and time into such a reference interval of seconds. WebCal provides merely with computations between pairs of intervals of seconds. The system, thus, does neither provide with means for temporal reasoning and/or scheduling nor for manipulating time granularities.

The supported calendars and conversions of each of their calendar units to the reference time unit are implemented in the system. In particular, the Gregorian calendar is implemented in the WebCal prototype. Thus, this system does not provide any means for user-defined calendars.

2.3.5 Temporal and Active Web Systems

Temporal Web Systems. Management and processing of time-dependent objects is a thoroughly investigated research issue of temporal database systems (cf. `http://www.scism.sbu.ac.uk/cios/paul/Research/tdb_links.html`, a link collection of research on temporal database systems). In such systems, time-dependent objects are usually associated with one or more temporal dimensions, in particular, transaction time, valid time, and/or event time [Je98]. The transaction time of an object is the time the object is current in the underlying system. The valid time of a proposition related to some stored object is the time the proposition is true in the underlying system. Event time is the time a particular event related to some stored object occurs. Event occurrences change the truth values of some propositions belonging to that stored object. Those temporal dimensions are the backbones of a formal representation of the evolution of objects over time. Such a formal representation is called an object's history. A history is modeled by a time line and time primitives (i.e. points or intervals) over the time line. Time-dependent objects are timestamped by those time primitives, i.e. the time primitives specify the temporal representation of an object of the real world or of an underlying system.

Similar problems to those investigated in the context of temporal database systems also apply to Web documents and other Web resources and Web and Semantic Web applications such as Web resource versioning, change management in Web documents, booking and reservation systems, news services, and XML warehouses. Several proposals concerning version management, change representation, and querying of changes in such Web resources and Web applications, as well as extensions of semi-structured data, in particular, to timestamp objects regarding their transaction times and/or valid times have been made in the last years [DBCK96, DBCK98, CTZ00, GM00, CTZ01, Web01, Dyr01, CAM02, Nør02, DLW04, NLM03].

Active Web Systems Automatically reacting on changes of time-dependent objects is a thoroughly investigated research issue of active database system (cf. `http://www.ifi.unizh.ch/dbtg/Links/adbs_sites.html`, a link collection of research on active database systems). An active system allows users to specify actions to be taken automatically when its state reaches a certain pre-defined condition. Automatic responses of active systems are usually declared by using the Event-Condition-Action (ECA) rules proposed in [DBB^{+}88]. The event belonging to such a rule may be either a primitive event or a composite event. Usually, primitive events are the basic operations such as insert and delete. A composite event is a compound set of basic events. Several approaches to model and detect composite events in active databases exists [GD92, GJS93, Hin03].

Similar problems to those investigated in the context of active database systems also apply to Web resources and Web and Semantic Web applications. For example, Web applications evolving automatically according to some user's actions and background (e.g. his/her flight bookings, or his/her medical history) are crucial in the context of the so-called "adaptive Web". In the last years, proposals have been made, applying the concepts of active database systems to the (Semantic) Web. In particular, research on adaptive Web applications and systems concentrates on user models, Web application models and adaptation models that may be based on ECA rules which exploited, for example, to incorporate reactive functionality in XML documents and to incorporate reactivity in ontology evolution [AYU01, BCP01, PPW03, ABB04, MAB04, AAM05]. Furthermore, a variety of data models are proposed for specifying adaptive hyper-media [dBHW99, KW02, CTB03].

2.4 In Comparison with CaTTS

A programming language or a query language to be used in an inter-cultural context such as the Web or the Semantic Web to model and to retrieve calendric data or to offer world-wide services to process such data should be aware of differences between cultural and/or professional calendars in use today [BHRS05]: for example, scheduling the phone conference of three persons in France, Greece, and Japan, the language used to infer a possible time slot has to consider the personal and professional time constraints of the persons. Furthermore, everyone's calendar data should be better expressed in the calendar and time zone the person is used to. Calendar data conversion, for example from weeks to months or between Japanese and Gregorian year numbering that must be performed for calculations should be invisible to the users.

So as to make it possible for every user to express calendar data in the calendar he/she is used to, the Calendar and Time Type System CaTTS [BS04, BRS05] has been developed. CaTTS consists of two languages, a *type definition language*, CaTTS-DL, and a *constraint language*, CaTTS-CL, of a (common) parser for both languages, and of a language processor for each language. In particular, CaTTS provides with *predicate subtyping* and *constraint solving* approaches to calendric data and constraints according to calendric types defined by some user in a CaTTS-calendar specification. CaTTS is based on a formal time model [BS03, BLS05] in accordance with the set-theoretic tradition of time granularity systems, however, purely interval-based.

2.4.1 Approaches to Temporal Knowledge Representation and Reasoning

CaTTS' notion of time is *linear*. CaTTS is not intended for expressing possible futures, hence it is not based on a "branching time". Most common-sense, Web and Semantic Web applications and many Web services can conveniently be modeled in a linear time framework.

CaTTS' notion of time is purely *interval-based*, i.e. temporal data of every kind have a duration. This reflects a widespread common-sense understanding of time according to which one mostly refer to interval, not to points. For example, one refers to "2004/10/05", a day, or to "1st week of October 2004", a week. Even point-like data such as 9:25 can be perceived as having a duration, possibly as small as one second or one millisecond. It has two advantages to consider intervals only:

1. It significantly simplifies data modeling, an advantage for CaTTS' users.
2. It simplifies data processing, i.e. type checking and constraint reasoning, an advantage for CaTTS' language processors.

However, CaTTS can deal with point-like data like the beginning of a week or whether a day d falls into a week w or not, as well.

CaTTS' notion of time is based on a system of *time granularities*, following this promising approach form Artificial Intelligence and Database Systems to model and reason with calendric data. However, CaTTS is a programming language approach defining time granularities as types. Furthermore, CaTTS gets along with only two (slightly modified) relations between time granularities and their compositions out of the large set of (possible) relationships between time granularities.

Implicit Time Models. CaTTS' time model inherently differs from change-based approaches such as the Situation Calculus and the Event Calculus following the tradition of implicit time models. In CaTTS, a notion of time is explicitly introduced. Implicit time models have been proposed for applications in active databases or active Web and Semantic Web whereas CaTTS is designed for modeling cultural and professional calendars and for solving multi-calendar appointment scheduling problems referring to such calendars.

Explicit Time Models. CaTTS is based on an explicit time model since CaTTS is a type language to define calendars and to reason over calendric data of different types. Since CaTTS' notion of time is purely interval-based, CaTTS' time model particularly bases on Allen's interval calculus [All83]. Interval-based time models follow the common-sense intuition of time always having a duration. Purely interval-based time models have several advantages compared to those which combine (durationless) points and intervals. Discussions on the advantages and disadvantages of purely interval-based time models compared to those combining points and intervals are provided in Section 2.1.2 and Chapter 3. CaTTS does not support generalized intervals.

Temporal Constraints. CaTTS provides with a constraint-based approach to solve multi-calendar appointment scheduling problems involving calendric data of different types (i.e. time granularities). For example, a person plans a meeting lasting 3 working days after 22nd April 2005 and before May 2005. A colleague's visit of 5 days must overlap with the planned meeting. CaTTS' solver tests for consistency and then calculates several (at least one) solutions such that all of the stated constraints are satisfied.

Multi-calendar appointment scheduling problems involve metric temporal constraints over intervals. Thus, metric temporal constraints, merely reasoning on points and distances between points are not sufficient. For example, the problem "the visit must overlap the meeting two working days" cannot be expressed in terms of metric, point-based temporal constraints in a Horn DLR framework. However, such a problem can be expressed in CaTTS. CaTTS' reasoner supports metric, interval-based temporal constraints in a finite domain constraint framework.[6] Furthermore, CaTTS allows for reasoning on such constraints where the domains of the variables possibly refer to different calendric types (time granularities).

Multi-calendar appointment scheduling problems inherently differ form qualitative temporal constraint problems: in a qualitative temporal constraint problem some (temporal) relations (e.g. some interval relations) are specified between objects. An example of a merely qualitative temporal reasoning problem is the following: different persons come and leave a bank under certain constraints which are expressed in terms of temporal relations. Then one might ask questions such as "Could possibly persons A and B *meet* at the bank's entrance?". Such kinds of questions does not appear frequently in appointment scheduling problems. Qualitative temporal reasoning thus performs reasoning on the temporal relations that hold between the different temporal objects whereas CaTTS' solver performs reasoning on the temporal objects (which refer to intervals with calendric type (time granularity)) themselves.

Although multi-calendar appointment scheduling is a rather metric temporal reasoning problem, it goes beyond metric temporal reasoning. In particular, CaTTS' reasoner goes beyond the "only existing constraint solver with different time granularities" (which is purely point-based and metric) as the authors of the work presented in [BJW00] claim. Using merely metric temporal reasoning frameworks (which provide with constraints to model points and distances between points) one could express temporal constraints such as "person A is at the bank", but neither "an event e happens during a task t" nor "an event e happens 5 time units before an event e"'.[7] Such constraints can be however expressed in CaTTS-CL.

Time Granularity Systems. CaTTS' time model basically refers to the set-theoretic approach to time granularities. However, CaTTS is a programming language approach to model time granularities and calendars as types. CaTTS provides with declarative

[6]The constraint system finite domain is the common choice for modeling scheduling problems in a constraint-based framework [FA97].

[7]Such constraints cannot be expressed in qualitative temporal reasoning frameworks, as well.

language constructs to define time granularities as types and with time constraints to reason over the elements of such types. CaTTS is particularly designed as a type system, in principle, integratable in *any* Web or Semantic Web language like XQuery, XSLT, RDF, or OWL. CaTTS provides with type checking approaches to temporal and calendric data. Furthermore, CaTTS provides with an application to constraint solving of multi-calendar scheduling problems with time granularity on intervals, whereas constraint solving in [BJW00] is inherently point-based, dedicated to metric constraints with time granularity only.

CaTTS yields a complete solution to the problem of time granularity conversion addressed in [FM01]. Actually, time granularity conversion is inherent to the operational semantics of the language CaTTS.

2.4.2 Calendric Computations

CaTTS is a declarative modeling and reasoning language to define time granularities and calendars as types in some CaTTS-calendar specification. Furthermore, CaTTS provides means to specify calendric constraints over the elements of such types to define multi-calendar appointment scheduling problems that can be solved by CaTTS' constraint solver. CaTTS is not intended as a full-fledged programming language such as Prolog, rather modeling of calendric types should be preserved as declarative and intuitive as possible. However, since calendars are often rather complex, involving several particularities and irregularities, CaTTS can be complemented with algorithms for calendric computations in the following ways.

1. Basically, CaTTS provides with language constructs to define *periodic exceptions* such as Gregorian leap year regulations in months within a CaTTS type declaration.
2. Additionally, more complex calendric calculations that are necessary to define calendric types such as Hebrew months and Gregorian or Orthodox Easter Sunday can be either specified
   (a) by defining CaTTS-*macros*, or
   (b) by *importing* externally defined functions.

The addressed examples are given below in CaTTS-syntax.

```
...
(* Gregorian leap year rule *)
type month = aggregate
      31 day named january ,
      alternate month(i)
          | (i div 12) mod 4 == 0 &&
            ((i div 12) mod 400 != 100 &&
             (i div 12) mod 400 != 200 &&
             (i div 12) mod 400 != 300)   -> 29 day
          | otherwise                     -> 28 day
      end named february,
      ...,
      31 day named december
 @ day(1);
...
(* Hebrew months *)
type month = aggregate
     30 day named nisan,
     29 day named iyyar,
     30 day named sivan,
     29 day named tammuz,
     30 day named av,
     29 day named elul,
     30 day named tishri,
     alternate month(i)
      | newYearDelay?(i) == 2 -> 30 day named long_marheshvan
      | otherwise              -> 29 day named short_marheshvan
     end named marheshvan,
     alternate month(i)
      | newYearDelay?(i) > 0 -> 30 day named long_kislev
      | otherwise             -> 29 day named short_kislev
     end named kislev,
     29 day named tevet,
     30 day named shevat,
     alternate month(i)
      | isLeapAdarRishon?(i) -> 30 day named adar_rishon
      | otherwise none
     end,
     29 day named adar_sheni @ day(-21) ~@ 133;
   type adar = adar_rishon | adar_sheni;
macro newYearDelay?(y) = if ny2 - ny1 == 356 then 2
                         else if ny1 - ny0 == 382 then 1
                         else 0 where
                              ny0 = y2ed?(y - 1),
                              ny1 = y2ed?(y),
                              ny2 = y2ed?(y + 1);
macro y2em?(y) = (235*y - 234) div 19;
macro m2y?(m) = if y2em?(y) == em then y
                else y + 1 where
                em = m + 70852,
                y  = (19*em + 234) div 235;
macro y2ed?(y) = if 3*(d + 1) mod 7 < 3 then d + 1
                 else d where
                      em = y2em?(y),
                      ep = 12084 + 13753*em,
                      d  = 29*em + (ep div 25920);
macro isLeapAdarRishon?(m) = isLeapAdarRishonInCycle?((m mod 235) + 1);
macro isLeapAdarRishonInCycle?(i) =
     i ==  36 || i ==  73 || i ==  98 || i == 135
  || i == 172 || i == 209 || i == 234;
macro isHebrewLeapYear?(y) = (7*y + 1) mod 19 < 7;
```

```
...
(* Orthodox Easter *)
macro shifted_epact(greg_y) = (14 + 11*(greg_y mod 19)) mod 30;
macro julian_year(greg_y) = if greg_y > 0 then greg_y
                            else greg_y - 1;
macro julian_april_19th_ndx(jul_y) = k where
    day(k) during Julian.year(jul_y) &&
    relative index day(k) in Julian.april == 19;
macro paschal_moon(greg_y) = day(k) where
    jul_y == julian_year(greg_y) &&
    k == julian_april_19th_ndx(jul_y) - shifted_epact(greg_y);

type orthodox_easter = with select day(i) where
    day(i) within year(y) && sunday(j) after paschal_moon(y) &&
    day(i) equals min(sunday(j));
...
(* import of an externally defined function *)
import greg_easter;

type gregorian_easter = with select C.day(i) where
    C.day(i) equals greg_easter(j);
```

2.4.3 Web and Semantic Web Formalisms and Applications

CaTTS is a programming language approach to data modeling and reasoning with time and calendars. It considerably simplifies the modeling of peculiarities of cultural calendars including particularities such as leap years, sun-based cycles like Gregorian years, or lunar-based cycles like Hebrew months as well as the modeling of professional calendars often involving "gaps" in time (e.g. "working-day"), "gapped" data items (e.g. data items of type "working-week"), and periodic events (e.g. "CS123-lecture") due to user-defined data types.

The well-known advantages of typed languages such as error detecting, language safety, gain in efficiency and consistency, abstraction, and documentation apply to CaTTS, as well.

Furthermore, CaTTS comes along with a constraint solver dedicated to (user-defined) calendar specifications; this dedication makes considerable search space restrictions, hence gains in efficiency, possible.

A discussion how CaTTS complements and/or differs from Web and Semantic Web formalisms and applications, in particular, those formalisms and applications surveyed in Section 2.3 follows.

Data Type Definition Languages. CaTTS inherently complements data type definition languages for the Web and the Semantic Web such as XML Schema: XML Schema provides a considerably large set of predefined time and date data types dedicated to the Gregorian calendar. In XML Schema, user-defined data type declarations are restricted to regular expressions over character sets. Whereas CaTTS provides with declarative language constructs for user-defined data types dedicated to any calendar.

Ontology Languages and Time Ontologies. CaTTS complements *DL-style* data modeling and reasoning methods for the Semantic Web such as RDF and OWL: DL-style languages are designed for *generic* Semantic Web applications. In contrast, CaTTS provides with methods *specific* to particular application domains, that of calendars and time. Furthermore, ontology languages such as OWL are based on description logics where temporal and calendric data is axiomatized in the logic and tableau-based reasoning algorithms are used to solve problems such as checking for consistency. In contrast, CaTTS is a programming language approach exploiting the advantages of data types, type checking approaches, and constraint-based reasoning methods.

CaTTS departs from time ontologies such as the KIF time ontology, the DAML time ontology, and time in OWL-S: while (time) ontologies follow the (automated reasoning) approach of "axiomatic reasoning", CaTTS is based on a (specific) form of "theory reasoning" [Sti85, BM05], an approach well-known through para-modulation [RW69]. Like para-modulation ensures efficient processing of quality in resolution theorem proving, CaTTS provides the user with convenient constructs for calendric types and efficient processing of data and constraints over those types.

Internationalization and Localization Efforts. CaTTS supports internationalization and localization efforts in the Web and the Semantic Web specialized on the domain of time and calendars. It goes far beyond locale specifications of times and dates for Web resources provided within the W3C Internationalization Activity.

CaTTS is intended to be, in principle, used to type check programs or specifications in *any* language (e.g. SQL, XQuery, Sparql, RDF, OWL) [BLOS03, BBL$^+$04], using temporal and calendric data enriched with type annotations after some calendar specified in CaTTS as well as to reason with multi-calendar appointment schedules involving arbitrary calendric data as long as the data is typed after calendric types defined in CaTTS.

Applications: Web Services and Temporal and Active Web. CaTTS complements Web services such as calendar agents and calendar Web servers with user-defined calendric types and with reasoning approaches to solve multi-calendar appointment scheduling problems over the elements of such data. CaTTS means for calculating scheduling problems go far beyond predefined scheduling problems over dates and times with (minute-based) durations. Furthermore, CaTTS is neither restricted to a finest time granularity nor does it claim conversions to this finest time granularity to perform any calculations. Furthermore, in CaTTS it is up to the user whether he/she uses a connection of calendric types defined in some CaTTS-calendar specification to real-time or not. For example, to infer a schedule of some student's examinations within a one-month time slot, a connection to real-time is useless.

However, CaTTS does not support fuzzy time intervals: primarily, solving appointment scheduling problems rather has to deal with temporal imprecision (e.g. that a time specified in terms of month becomes imprecise when it is expressed in terms of days) and user-preferences which might be solved by specifying so-called soft constraints rather than with fuzziness.

CaTTS might be used as a time and calendar specification language for the time-related objects modeled in temporal or active systems.

Note: Research on (annotation) languages for events and temporal expressions in natural text for Semantic Web applications such as TimeML [IJ04] as well as research on agent-based creation of temporal categories and concepts from natural language texts based on machine learning methods [Beu04] inherently differ from CaTTS: the main difference between CaTTS and languages like TimeML is that TimeML annotates with markup that can be further used for information extraction and question answering of natural language text *after* the used documents have been composed. CaTTS provides a means to annotate documents *before* composing them. Furthermore, CaTTS is a type language specialized in calendar and time modeling and reasoning, addressed to Semantic Web applications and Web Services, and, in particular, to multi-calendar appointment scheduling problems.

Chapter 3

A Time Model for Calendric Data, Types, and Constraints

"Zeit ist das, was man an der Uhr abliest."
(Albert Einstein, 1879–1955)

This chapter introduces into a time model to declaratively define calendric and temporal data, types, and constraints and to reason with such constructs. In particular, this time model formally introduces the temporal and calendric concepts underlying the type language CaTTS. The time model is convenient and intuitive to model real-life calendar expressions. It supports properties such as "gapped" calendric expressions like working days nevertheless being continuous and point-like calendar dates like "today" nevertheless having a duration which are inherent to cultural and professional calendars like the Gregorian and Hebrew calendars and the academic calendar of a university. The model is expressive enough to describe most common features of cultural and professional calendars like leap seconds, leap years, time zones, and "gaps" in time, for example between working days or within working years. In particular, the time model meets the requirements of the calendar type language CaTTS regarding its *(type) definition language* and its *constraint language* (cf. Chapter 4), its *multi-calendar constraint solver* (cf. Chapter 5), and its *type system* (cf. Chapter 6).

CaTTS' time model is *linear*. CaTTS is not intended for expressing possible futures, hence it is not based on a "branching time". Most common-sense, Web and Semantic Web, and many Web service applications can be conveniently modeled in a linear time framework like appointment scheduling systems.

CaTTS' time model is *discrete*. CaTTS is a language to specify in a rather simple manner more or less complex, cultural or professional calendars like the Hebrew calendar and the academic calendar of a University. People use calendars to give names to specific "portions" of time like days, weeks, or years according to some calendar (time) unit. Such portions of time can be naturally modeled by intervals over a continuous time line which can be referred to by an (integer) index.

CaTTS' time model is purely *interval-based,* i.e. temporal data of every kind have a duration. CaTTS has no notion of "time points" without duration. This reflects a widespread common-sense understanding of time according to which one mostly refers to time intervals, not to time points. For example, one refers to "2004/10/05", a day, or to "1st week of October 2004", a week. Even time point-like data such as "9:25" can be perceived as having a duration, i.e. one minute. Considering a time model containing only time intervals and no time points has two advantages. First, it significantly simplifies data modeling, an advantage for CaTTS' users. Second, it simplifies data processing, i.e. type checking and constraint reasoning, an advantage for CaTTS' language processors.

Since CaTTS' time model is purely interval-based and discrete, it is aware of the following well-known (rather theoretical) temporal modeling and reasoning problems [FGV05]:

1. Point-based events like "start moving".
2. Continuous changes involving point-based and interval-based events like "100-meter sprint".
3. The homogeneity of interval-based events, i.e. if an event e is true on interval i, then it must hold on any subinterval of i.
4. The concatenation of interval-based events, i.e. if an event e is true on two consecutive intervals i_1 and i_2, then e must also be true on the time "concatenating" i_1 and i_2.
5. Determining the truth value of an event e at point p in a time model considering points and intervals. For example, assume that e is true on interval i_1 and false at interval i_2. Assume further that i_1 meets i_2 at point p, then different possibilities appear to model the intervals i_1 and i_2:
   (a) e is true and false at p if the intervals are closed.
   (b) There is a "truth gap" at point p if the intervals are open.
   (c) The option to model intervals in such a framework are either right-open or left-open intervals of time points which is rather artificial.

   This problem is sometimes referred to as the "divided instant problem" [FGV05].

The first two issues formulated in the list given above introduce a notion of "imprecision" in terms of calendars, since a point-based event can only be specified as precisely as the underlying calendar (time) unit (e.g. millisecond, hour, and month) used. Problems three to five, formulated in the afore mentioned list does not appear in CaTTS, since CaTTS' notion of time is discrete and interval-based.

Note: Inherent to a discrete time model is the fact that each event has a duration, and that it is possible to refer to some "previous" or "next" event in time (cf. Section 2.1.2).

3.1 Base Time Line

In order to *formalize* CaTTS interval-based (see also [BS03], thus, time point-less time model, however, time points in a continuous time line (cf. Definition 3.1) have to be considered. This ensures a connection of (temporal) *activities*[1], in particular, *events* and *tasks*[2], to a continuous flow of time, in particular, to ensure homogeneity of such activities. Activities are objects with a temporal extend, defined over the time model that is introduced in this chapter.

Definition 3.1 (Base Time Line). *A* **base time line** *is a pair ($\mathcal{T}$,$<_\mathcal{T}$) where $\mathcal{T}$ is an infinite set (isomorphic to $\mathbb{R}$) and $<_\mathcal{T}$ is a total order on $\mathcal{T}$ such that $\mathcal{T}$ is not bounded for $<_\mathcal{T}$. An element $t \in \mathcal{T}$ is called* **time point***.*

The time points in the base time line are totally ordered, i.e. $<_\mathcal{T}$ is an irreflexive, antisymmetric, transitive, and linear order relation. The time points are further neither right-bounded nor left-bounded. Finally, the base time line is continuous, i.e. isomorphic to the real numbers.

3.2 "Discretization" of Time

Each *calendric type* defined in a CaTTS *calendar* specification creates a *discrete* image of the (continuous) base time line ($\mathcal{T}$,$<_\mathcal{T}$). Thus, CaTTS calendric types specify countable sets isomorphic to (subsets of) the integers. For example, calendric types "day" and "working-day" imply two (different) images of the base time line: days partition this time line, working days correspond to a portion of the day partition of this time line. Both images of the base time line are discrete, i.e. isomorphic to the integers.

3.2.1 Time Granularities

The notion of *time granularity* [Mon96, Je98, BJW00, Euz01] (cf. Section 2.1.4) formalizes such a "discretization" of a (continuous) base time line ($\mathcal{T}$,$<_\mathcal{T}$).

Definition 3.2 (Time Granularity). *Let ($\mathcal{T}$,$<_\mathcal{T}$) be a base time line. A* **time granularity** *is a (non-necessarily total) function $\mathcal{G}$ from $\mathbb{Z}$ into the power set of $\mathcal{T}$, $\mathcal{P}(\mathcal{T})$, such that for all $i, j \in \mathbb{Z}$ with $i < j$*

1. *if $\mathcal{G}(i) \neq \emptyset$ and $\mathcal{G}(j) \neq \emptyset$, then for all $t_i \in \mathcal{G}(i)$ and for all $t_j \in \mathcal{G}(j)$ $t_i <_\mathcal{T} t_j$.*
2. *If $\mathcal{G}(i) = \emptyset$, then $\mathcal{G}(j) = \emptyset$.*

[1]The notion "activity" is frequently used in Constraint Programming to describe objects that have a temporal extent [Apt03].

[2]The notions "event" and "task" are taken from research on "planning" and "scheduling", well-known kinds of Constraint Satisfaction Problems [FA97].

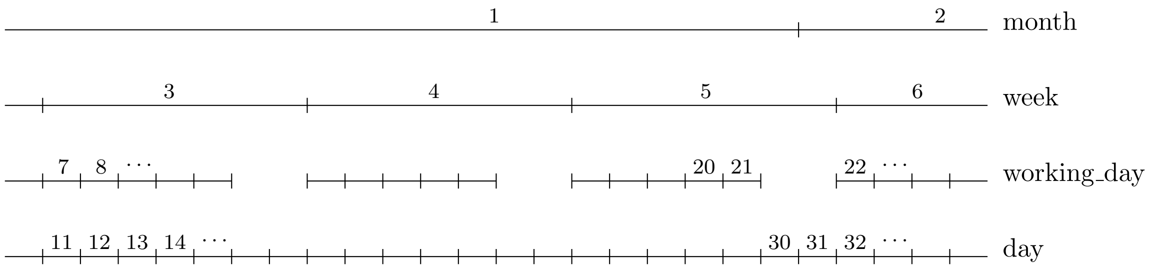


Figure 3.1: An illustration of some time granularities.

Examples of time granularities are day, working day, weekend, week, month, holiday, etc. Definition 3.2 accounts in addition to time granularities which are partitions of the base time line like "day" or "month", time granularities with non-continuous elements like "holiday", time granularities with non-convex elements like "working week", and bounded time granularities like "someone's exam days during his/her years of study". Figure 3.1 illustrates some time granularities.

According to Definition 3.2, two different elements of the same time granularity do not overlap. The first condition of Definition 3.2 induces from the ordering of the integers the common-sense ordering on elements of time granularities: for example, the day "10/25/2004" is after the day "10/24/2004"[3]. The second condition of Definition 3.2 is purely technical: it makes it possible to refer to the *infinite* set $\mathbb{Z}$ also for *finite* time granularities (e.g. someone's exam days during his/her years of study).

Note: Explicitly defining the "non-existing parts" by an empty set of a (finite) time granularity provides a means to refer to the inverse of a time granularity like "days except for exam days during someone's years of study", as well.

Each granule of every time granularity $\mathcal{G}$ is represented by an integer index (recall that $\mathcal{G}$ is isomorph to $\mathbb{Z}$) as illustrated in Figure 3.1. Since a time granularity $\mathcal{G}$ is defined as a mapping form integers into (generalized) intervals (i.e. intervals which are not necessarily continuous according to the base time line) over the base time line, each granule of every time granularity has a duration. Granules may be therefore divided into sub-sets (in some "finer" time granularity). The definition of time granularities does not enforce a restriction to a finest time granularity.

At a first glance, a time model considering only time intervals and not time points falls short in modeling activities like "start moving" if viewed as being instantaneous, i.e. having no duration. This might be true for a time model that merely defines a single time line

[3]If corresponding date formats for the integer indices of the elements are defined. Date format specifications for the granules of some time granularities is provided with CaTTS-FDL (cf. Section 4.1.3).

isomorphic to $\mathbb{R}$. But the time model introduced in this chapter defines multiple time lines in terms of time granularities isomorphic to $\mathbb{Z}$. Thus, each activity is defined according to a time granularity. That means that the afore mentioned problem is no longer applicable: an activity may only be viewed being instantaneous according to some time granularity. For example, the activity “start moving” can be expressed in terms of seconds. This kind of “imprecision” reflects measuring and specifying activities in terms of a time granularity: a proposition made about an instantaneous activity can only be as precise as the underlying time granularity.

3.2.1.1 Activities over Time Granularities

Although each element of any time granularity has a duration, point-like data can be represented in CaTTS, as well. For example, one can determine the beginning of the “44th week 2004” in terms of days, i.e. “25.10.2004”, the “time point” “25.10.2004” starts the “time interval” “44th week 2004”.

The following definitions introduce concepts, called *activities* that define different kinds of objects that take time in a time granularities.

Definition 3.3 (Activity). *Let $\mathcal{G}$ be a time granularity.*
An **activity** *A* **over** *$\mathcal{G}$ is either*

- *an element $\mathcal{G}(i)$ of $\mathcal{G}$, called* **event***, or*
- *a continuous sequence of elements $\mathcal{G}(i_1), \ldots \mathcal{G}(i_k)$, $1 \leq k$ of $\mathcal{G}$, called* **task***.*

According to Definition 3.3, an event can be understood as a (time) point with time granularity, and a task can be understood as a (time) interval with time granularity. Examples of events are “44th week 2004” in time granularity “week” and “25.10.2004” in time granularity “day”. “The last two weeks of July 2004” is an example of a task in time granularity “week”.

Definition 3.4 (Duration). *Let $\mathcal{G}$ be a time granularity.*
A **duration** *D* **over** *$\mathcal{G}$ is a number (expressed as an unsigned integer) of a continuous, finite sequence of $\mathcal{G}$.*

A duration can be informally understood as an (finite) interval with a time granularity that has a given length but no specific starting or ending time.

CaTTS’ constraint language, CaTTS-CL provides several relations over activities and durations of activities to reason about them (cf. Section 4.2).

Note: Since time granularities introduce a “discretization” of time, an event has a duration, i.e. 1 in terms of its associated time granularity. For example, the event “25.10.2004” in time granularity “day” has duration 1 (day), and the task “the last two weeks of July 2004” in time granularity “week” has duration 2 (week).

3.2.1.2 Time Granularities in CaTTS

CaTTS (cf. Chapter 4) provides with language constructs, so-called *type constructors*, to define time granularities as *(base) types*. Since time granularities are nothing but (infinite) sets isomorphic to the integers, they can be defined by *predicates* in a higher-order logic. Such predicates are used as the type constructors of the language CaTTS. Thus, a (base) type in CaTTS specifies (in terms of a predicate) a set, referred to as "predicate set". The usual set-theoretic operations (e.g. $\cup$) can be applied to predicate sets. The elements of such base types in CaTTS are events as defined in Definition 3.3. Events refer to (time) points with time granularity.

In Chapter 4, type constructors not only for base types, but also for *composed types*, in particular, for types of tasks, i.e. interval types and types of durations, i.e. duration types are introduced for CaTTS. In CaTTS, points, intervals, and durations are denoted as different *kinds* of calendric types (time granularity).

3.2.2 Relations between Time Granularities

In CaTTS, calendric types (time granularities) are defined either in terms of *inclusion* or in terms of *aggregation* of other calendric types (time granularities) such that the resulting type always defines a specified *subset* of another calendric type (time granularity). Those subsets are referred to as *subtypes* in CaTTS. For example, the calendric type (time granularity) "working-day" is an inclusion subset (in the common set-theoretical sense) of "day", since the set of working days is a subset of the set of days. "Week" is an aggregation subset (in the sense of constructing sets from other sets as illustrated in Figure 3.2) of the time granularity "day", since each week can be unambiguously specified by an interval of days. Thus, the calendric types (time granularities) "working-day" and "week" are both subtypes (the former in terms of inclusion and the latter in terms of aggregation) of the calendric type (time granularity) "day". Definitions of inclusion and aggregation of time granularities are given in the following.

3.2.2.1 Aggregations

Set-theoretically, some time granularity can be constructed from another time granularity by aggregating intervals of elements of a "finer" time granularity to a single element of a "coarser" (i.e. aggregated) time granularity. A hierarchy of set-aggregations of time granularities is illustrated in Figure 3.2.

Definition 3.5 (Aggregation). *Let $\mathcal{G}$ and $\mathcal{H}$ be time granularities.*
$\mathcal{G}$ is an **aggregation of** *$\mathcal{H}$, denoted $\mathcal{G} \preceq \mathcal{H}$, if for all $i, j \in \mathbb{Z}$ $\mathcal{H}(j) \subseteq \mathcal{G}(i)$, i.e. if every element of $\mathcal{G}$ is an interval over $\mathcal{H}$ and every element of $\mathcal{H}$ is included in (exactly) one element of $\mathcal{G}$.*

With Definition 3.5, day $\preceq$ day, week $\preceq$ day, month $\preceq$ day, and working week $\preceq$ working day, but neither week $\preceq$ working day (a week cannot be defined by an interval of working

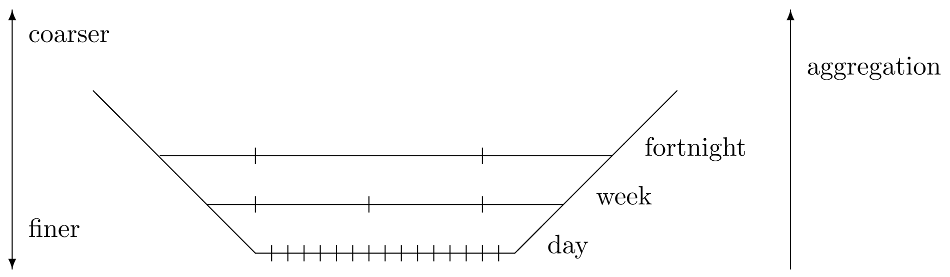


Figure 3.2: An illustration of aggregations of time granularities.

days, i.e. all days which are neither weekend days nor holidays) nor month $\preceq$ week (a month cannot be defined by an interval of weeks).

Note that the aggregation $\mathcal{G}$ of a time granularity $\mathcal{H}$ is equivalent to the union of the intervals over $\mathcal{H}$, each aggregated by a set of elements of $\mathcal{G}$.

3.2.2.2 Inclusions

Set-theoretically, some time granularity can be constructed form another time granularity by defining particular subsets such that the (new) time granularity is an inclusion (subset) of the original time granularity.

Definition 3.6 (Inclusion). *Let $\mathcal{G}$ and $\mathcal{H}$ be time granularities.*
$\mathcal{G}$ is an **inclusion of** *$\mathcal{H}$, denoted $\mathcal{G} \subseteq \mathcal{H}$, if for all $i \in \mathbb{Z}$ exists $j \in \mathbb{Z}$ such that $\mathcal{H}(j) = \mathcal{G}(i)$, i.e. if every element of $\mathcal{G}$ is an element of $\mathcal{H}$.*

With Definition 3.6, day $\subseteq$ day, weekend day $\subseteq$ day, and Sunday $\subseteq$ weekend day, but not day $\subseteq$ working week.

The two relations, inclusion of and aggregation of, are corner stones of CaTTS. As the examples given in Chapter 4 show, they are very useful in modeling calendars, in particular, they are sufficient to model any calendar in use today. Indeed, they reflect widespread forms of common-sense modeling and reasoning with calendric data.

3.3 Calendars

In CaTTS, *calendars* are specific (finite) sets of calendric types (time granularities) where each type contained is defined either by aggregation or inclusion of another calendric type (time granularity). The *composition* $\preceq \otimes \subseteq$, denoted $\leq$, of the relations aggregation of and inclusion of defines a partial order over calendric types (time granularities) in a calendar $\mathcal{C}$ such that each type (time granularity) is somehow "finer" or "coarser" than another type (time granularity). For time granularities $\mathcal{G}$ and $\mathcal{H}$ the composition $\mathcal{G} \leq \mathcal{H}$ is defined as follows:

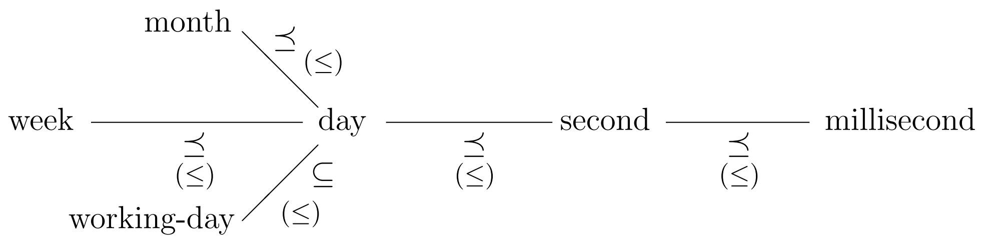


Figure 3.3: An illustration of a calendar.

Definition 3.7 (Subtype Relation.) *Let $\mathcal{G}$ and $\mathcal{H}$ time granularities. Then $\mathcal{G}$ is a* **subtype of** *$\mathcal{H}$, denoted $\mathcal{G} \leq \mathcal{H}$, if $\forall i \in \mathbb{Z}\ \exists j \in \mathbb{Z}.\ \mathcal{H}(j) \subseteq \mathcal{G}(i)$.*

That is, $\mathcal{G}$ is an "aggregated subset" of $\mathcal{H}$. Thus, a subtype is always coarser than its supertype. For example, if "week" $\preceq$ "day" and "summer week" $\subseteq$ "week", then "summer week" $\leq$ "day". In particular, if "week" $\preceq$ "day", then "week" $\leq$ "day", and if "working day" $\subseteq$ "day", then "working day" $\leq$ "day".

The subtype relation $\leq$ defines a partial order over time granularities in a calendar. The following formalization of the notion of *calendar* reflects the central role that subtypes play in CaTTS.

Definition 3.8 (Calendar). *Let $\mathcal{G}_1, \ldots, \mathcal{G}_n$ be time granularities.*
A **calendar** *$\mathcal{C}$ is a finite set of time granularities $\{\mathcal{G}_1, \ldots, \mathcal{G}_n\}$ such that there exists a $\mathcal{G}_i \in \mathcal{C}$ and for all $\mathcal{G}_j \in \mathcal{C}$, $i, j \in \{1...n\}$ and $i \neq j$ $\mathcal{G}_j \leq \mathcal{G}_i$.*
$\mathcal{G}_i \in \mathcal{C}$ is called **reference** *of the calendar $\mathcal{C}$.*

Figure 3.3 illustrates a set of time granularities defining a calendar. Each of the illustrated time granularities is a $\leq$-comparable with "millisecond". "millisecond" is the calendar's reference.

Note: A finite set $\mathcal{S}_{\mathcal{C}}$ of calendars is also a calendar according to Definition 3.8, if either the references of the calendars in $\mathcal{S}_{\mathcal{C}}$ are *aligned*, i.e. identical except for the numbering of their indices or there exists a type (time granularity) $\mathcal{G}_0$ which is $\leq$-comparable with the references of the calendars belonging to $\mathcal{S}_{\mathcal{C}}$. In particular, calendar alignment provides a means to reason with types (time granularities) specified in different calendars. In the following, with $\mathcal{C}$ we refer to both a calendar and a finite set of aligned calendars.

3.4 Time Granularity Conversion

Having defined a means to specify time granularities from other time granularities either in terms of aggregations or in terms of inclusions as well as a partial order between such time granularities, a means to *convert* the elements of one time granularity into (sets of) elements of another time granularity is introduced now. In the following, we define time granularity conversion for time granularities $\mathcal{G}$ and $\mathcal{H}$ where $\mathcal{G} \leq \mathcal{H}$.

Note: The conversion for time granularities which are comparable according to the subtype relation shows a strong relationship between the aggregation of and the inclusion of relations. Furthermore, this conversion shows that time granularities can be conveniently ordered according to such an "aggregation-subset" relation (in the sense of constructing subsets from other sets).

Definition 3.9 (Conversion). *Let $\mathcal{G}$ and $\mathcal{H}$ be time granularities of some calendar $\mathcal{C}$. The conversion function $c_{\mathcal{G}\to\mathcal{H}}$, converting elements of $\mathcal{G}$ to (sets of) elements of $\mathcal{H}$ and its inverse function $c_{\mathcal{H}\to\mathcal{G}}$ is defined for $i, j \in \mathbb{Z}$ as follows:*

1. *if $\mathcal{G} \subseteq \mathcal{H}$, then*

$$\begin{array}{rcl} c_{\mathcal{G}\to\mathcal{H}}(\mathcal{G}(i)) & := & \mathcal{H}(j) \\ c_{\mathcal{H}\to\mathcal{G}}(\mathcal{H}(i)) & := & \left\{\begin{smallmatrix} \mathcal{G}(j) & \text{if } \mathcal{G}(j)=\mathcal{H}(i) \\ \emptyset & \text{otherwise} \end{smallmatrix}\right\} \end{array}$$

2. *if $\mathcal{G} \preceq \mathcal{H}$, then*

$$\begin{array}{rcl} c_{\mathcal{G}\to\mathcal{H}}(\mathcal{G}(i)) & := & \{\mathcal{H}(j) \mid \mathcal{H}(j) \subseteq \mathcal{G}(i)\} \\ c_{\mathcal{H}\to\mathcal{G}}(\mathcal{H}(i)) & := & \mathcal{G}(j) \end{array}$$

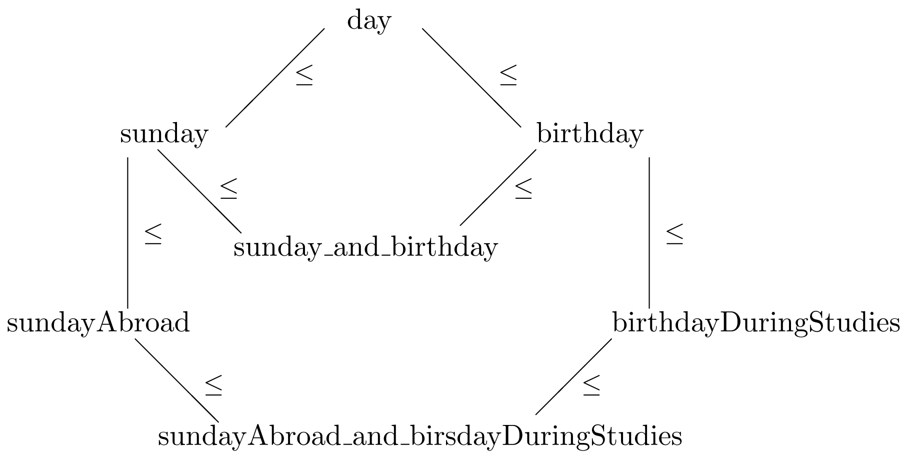
3. *if $\mathcal{G} \leq \mathcal{H}$, then*

$$\begin{array}{rcl} c_{\mathcal{G}\to\mathcal{H}}(\mathcal{G}(i)) & := & \{\mathcal{H}(j) \mid \mathcal{H}(j) \subseteq \mathcal{G}(i)\} \\ c_{\mathcal{H}\to\mathcal{G}}(\mathcal{H}(i)) & := & \left\{\begin{smallmatrix} \mathcal{G}(j) & \text{if } \mathcal{H}(i)\subseteq\mathcal{G}(j) \\ \emptyset & \text{otherwise} \end{smallmatrix}\right\} \end{array}$$

For example, $c_{working\_day\to day}(working\_day(8)) = day(12)$ if $working\_day \subseteq day$ (and thus $working\_day \leq day$) according to the indexing used in the illustration of Figure 3.1.

According to Definition 3.9 we ensure conversions between time granularities in a calendar which are $\leq$-comparable. But since the subtype relation only defines a partial order over the time granularities in a calendar $\mathcal{C}$, so far, conversion is not supported for *any* pair of time granularities in a calendar.

To ensure time granularity conversion between any pair of time granularities, we define a *join* of two time granularities according to the partial order $\leq$. In CaTTS, a join is slightly weaker than the ordinary lattice join, which only allows for the first condition of Proposition 3.1. The second of Proposition 3.1 is an extension to deal with calendars which are not (always) lattices. This particularity (of CaTTS calendars) results from the possibility to construct a new time granularity by conjunction of two other (already defined) time granularities as the following illustration shows:



With the illustration given above, the join of the time granularities "sunday_and_birthday" and "sundayAbroad_and_birsdayDuringStudies" is "day". "Sunday" and "birthday" are both candidates for the join of those two time granularities. "day", in turn, is the join of those two candidates. Thus, the second condition (of Proposition 3.1) forces the join to be the smallest possible *unique*[4] upper bound, i.e. the join of the possible candidates. Then for each pair of calendric types (time granularities) defined in a CaTTS calendar $\mathcal{C}$ such a join exists:

Proposition 3.1 *Let* $(\mathcal{C}, \leq)$ *be a calendar.*
For any pair of calendric types (time granularities) $\mathcal{C}_t$ *and* $\mathcal{C}_s$ *of* C*, there exists a join* $\chi \in \mathcal{C}$ *such that* $\mathcal{C}_s \vee \mathcal{C}_t = \chi$*, i.e.* $\mathcal{C}_s \leq \chi$*,* $\mathcal{C}_t \leq \chi$*, and for all* $\mathcal{C}_i \in C$ *with* $\mathcal{C}_t \leq \mathcal{C}_i$ *and* $\mathcal{C}_s \leq \mathcal{C}_i$*, either*

1. $\chi \leq \mathcal{C}_i$ *or*
2. $\mathcal{C}_i < \chi$*, and there exists another* $\mathcal{C}_k \in C$ *with* $\mathcal{C}_s, \mathcal{C}_t \leq \mathcal{C}_k$*,* $\mathcal{C}_k < \chi$ *and* $\mathcal{C}_i, \mathcal{C}_k$ *being incomparable.*

Proof 3.1 *For a pair of types* $\mathcal{C}$ *and* $\mathcal{D}$ *of calendar* $\mathcal{C}$*, consider the set of upper bounds* $U(\mathcal{C}, \mathcal{D}) = \{\upsilon | \mathcal{C} \leq \upsilon, \mathcal{D} \leq \upsilon\}$*.*
(Existence) If $\mathcal{C}_s = \mathcal{C}_t = \alpha$*, with* α *being the top element of* C*, then* $U(\mathcal{C}_s, \mathcal{C}_t) = \{\alpha\}$*, and our proposition is satisfied through (1). So, if* $\mathcal{C}_s \vee \mathcal{C}_t$ *exists, so does* $\mathcal{C}'_s \vee \mathcal{C}_t$*, with* $\mathcal{C}'_s$ *direct subtype of* $\mathcal{C}_s$*: If* $\mathcal{C}_s \leq \mathcal{C}_t$ *so is* $\mathcal{C}'_s \leq \mathcal{C}_t$ *and in this case* $\mathcal{C}_t$ *is the join, as* $\mathcal{C}_t \in U(\mathcal{C}'_s, \mathcal{C}_t)$ *satisfies (1). In case of* $\mathcal{C}_t < \mathcal{C}_s$*, either* $\mathcal{C}_t \leq \mathcal{C}'_s$ *and thus* $\mathcal{C}'_s \in U(\mathcal{C}'_s, \mathcal{C}_t)$ *satisfies (1), or else* $\mathcal{C}_t$ *and* $\mathcal{C}'_s$ *are incomparable and thus* $\mathcal{C}_s \in U(\mathcal{C}'_s, \mathcal{C}_t)$ *satisfies (1). Finally, if* $\mathcal{C}_t$ *and* $\mathcal{C}_s$ *are incomparable,* $\mathcal{C}'_s$ *cannot be greater than or equal to* $\mathcal{C}_t$*, because then* $\mathcal{C}_t$ *would have to be less than of equal* $\mathcal{C}_s$*; either* $\mathcal{C}'_s$*, too, is incomparable to* $\mathcal{C}_t$ *and thus* $\mathcal{C}'_s \vee \mathcal{C}_t = \mathcal{C}_s \vee \mathcal{C}_t \in U(\mathcal{C}'_s, \mathcal{C}_t)$ *satisfies (2), or else* $\mathcal{C}'_s \leq \mathcal{C}_t$ *and thus* $\mathcal{C}_t \in U(\mathcal{C}'_s, \mathcal{C}_t)$ *satisfies (1).*
(Uniqueness) Be $\chi = \mathcal{C}_s \vee \mathcal{C}_t$*. Let's assume* χ' *would also qualify as a join of* $\mathcal{C}_s$ *and* $\mathcal{C}_t$*. If* χ'

[4]in terms of equality

and χ were incomparable, then neither $\chi' \leq \chi$ nor $\chi < \chi'$ and thus χ' violates (1) and (2). If $\chi' < \chi$, then χ must have satisfied (2), thus exist an upper bound σ_k incomparable to χ'; however, all upper bounds are comparable to χ' if it is a join (1,2). Finally, if $\chi < \chi'$, then χ' must satisfy (2), thus exist an upper bound σ_k incomparable to χ, failing analogously.

The join provides a means to specify conversions between any pair of calendric types (time granularities) as follows: let $\mathcal{G}$ and $\mathcal{H}$ be time granularities. If neither $\mathcal{G} \leq \mathcal{H}$ nor $\mathcal{H} \leq \mathcal{G}$, then specify the join $\mathcal{J}$ of $\mathcal{G}$ and $\mathcal{H}$ according to $\leq$ and perform conversions between $\mathcal{G}$ and $\mathcal{J}$ and between $\mathcal{H}$ and $\mathcal{J}$. In this way, the elements of time granularities $\mathcal{G}$ and $\mathcal{H}$ can be compared in terms of the corresponding elements in the time granularity $\mathcal{J}$. For example, neither "week"$\leq$"month" nor "month"$\leq$"week", but "month"$\vee$"week"$=$"day". Then week and month granules might be both unambiguously represented by intervals of day granules which can be compared.

Note: Time granularity conversion according to such a join is a corner stone of CaTTS' multi-calendar constraint solver (cf. Chapter 5). In fact, time granularity conversion ensures constraint solving with arbitrary time granularities defined in a calendar $\mathcal{C}$ without loss of semantics. Furthermore, multi-calendar constraint solving based on time granularity conversion is more efficient than constraint solving where all elements are initially converted into the reference type of some calendar.[5]

[5]For example, the set of some elements of time granularity "week" is much smaller if represented in terms of "day" than it would be if represented in terms of "second".

Chapter 4

The Language CaTTS

(sun-based stone of the Aztecs
comprising 365 days within 18 months each of 20 days and 5 additional days)

The time model introduced in the previous chapter defines a formal framework to specify time granularities and calendars as types. In this framework, the formal concepts, *calendric type* (time granularity) and *calendar*, and relations between calendric types, in particular *aggregation* and *inclusion* as well as a means to compare and convert values of such calendric types are defined.

This chapter introduces the Calendar and Time Type System (CaTTS). CaTTS is a programming language approach to data modeling and reasoning with time and calendars. CaTTS consists of two languages, a *type definition language*, CaTTS-DL, and a *constraint language*, CaTTS-CL, of a (common) parser for both languages, and of a language processor for each language, in particular, a constraint solver (cf. Chapter 5) and type checking approaches (cf. Chapter 6). The formal specification of the language CaTTS, including

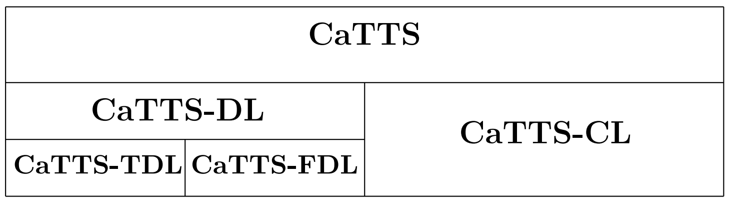


Figure 4.1: Languages of CaTTS.

CaTTS-DL and CaTTS-CL, in a BNF-like notation is given in Appendix A. Figure 4.1 surveys the languages that are part of CaTTS.

CaTTS-DL provides a means to specify in a rather simple manner more or less complex, cultural and/or professional calendars. Calendars expressed in CaTTS-DL are can be composed in the sense that the language offers a means for (possibly parameterized) modules. Thus, one can, for example, extend a standard calendar such as the Gregorian calendar used in Germany with the particular teaching calendar of a specific German university. *Calendric data types* specific to a particular calendar – such as "working day", "Easter Monday", "exam week", or "CS123 lecture" (defining the time the Computer Science lecture number 123 takes place) – can be defined using the language fragment CaTTS-TDL (for Type Definition Language) of CaTTS-DL. *Date formats* for such data types – such as "5.10.2004", "2004/10/05", or "Tue Oct 5 16:39:36 CEST 2004" – can be defined using the language fragment CaTTS-FDL (for Format Definition Language) of CaTTS-DL.

CaTTS-CL provides a means to express a wide range of temporal constraints over activities, i.e. objects with a temporal extend, in particular events and tasks. CaTTS-CL refers to the types defined in calendar(s) specified in the definition language CaTTS-DL. For example, if one specifies in CaTTS-DL a calendar defining both, the Gregorian calendar (with types such as "Easter Monday" or "legal holiday") and the teaching calendar of a given university (with types such as "working day ", "CS123 lecture", and "exam week"), then one can refer in CaTTS-CL to "days that are neither legal holidays, nor days within an examination week". Furthermore, constraints on such days can be expressed such as "after Easter Monday and before August".

This chapter introduces the various syntactic forms of the language CaTTS and how two use them. This chapter consists of three sections. First, CaTTS-DL's forms to specify calendric types, calendars, and date formats for the values of such calendric types are introduced. Subsequently, specifying calendric reasoning problems, in particular (multi-calendar) appointment scheduling problems using time constraints that refer to such data and types in CaTTS-CL is introduced. Finally, the expressiveness of CaTTS (including CaTTS-DL and CaTTS-CL) is illustrated by example.

4.1 CaTTS-DL: Definition Language

CaTTS' definition language, CaTTS-DL, consists of a type definition language, CaTTS-TDL, and a date format definition language, CaTTS-FDL. CaTTS-DL is designed as a modeling language for applications, systems, and services (in particular, in the Web and the Semantic Web) that require *specific* modeling and processing tools in the application domain of time and calendars.

CaTTS-TDL provides a set of type constructors to define calendric types (that satisfy Definition 3.2) like "day", "week", or "working day". Such calendric types are declared by defining *predicates*. Thus, in CaTTS a calendric type specifies a set in terms of a predicate.

In CaTTS, *calendars* (cf. Definition 3.8) are themselves "typed" by *calendar signatures.* Typed calendars have essentially the same advantages as valid XML documents ("typed" by a DTD or an XML Schema declaration). In particular, this makes CaTTS calendar specifications reusable, maintainable, and easy to extend. Furthermore, CaTTS' calendar type definitions allow for user-defined *calendar functions* that parameterize calendars. For example, one can parameterize any CaTTS calendar specification having a common "type" by a calendar function mapping some defined calendar to another time zone. Calendar functions specify parameterized calendars: a calendar function may be applied to each calendar "typed" by the calendar type definition appearing as parameter of the calendar function. Thus, calendar functions provide a means to specify different calendar versions all having the same calendar type definition.

CaTTS-FDL provides with means to specify date formats for time and dates associated with an internal integer index for each value of calendric types defined in a CaTTS-TDL calendar specification.

4.1.1 Reference Time

Each CaTTS implementation has a single predefined (base) type called `reference`. `reference` is a base type, because it has no internal structure as far as the type system of CaTTS is concerned. `reference` is a time unit such as "second" or "hour", chosen e.g. depending on the operating system. If a CaTTS calendar specification refers to the `reference` type, then all further (user-defined) types defined are expressed directly or indirectly in terms of `reference`, using CaTTS' aggregation and/or inclusion subtype constructors. If `reference` is for example the calendric type "second", a convenient choice with the Unix operating system, then one can specify (using CaTTS-DL) coarser types such as "day", "week", and "year" as well as finer types such as "millisecond". The reference type makes conversions among any other types defined in different CaTTS-DL calendar specifications (in terms of aggregation and/or inclusion) possible.

In CaTTS' prototype implementation, `reference` is the time unit "second" following the Unix operating system with midnight at the onset of Thursday, January 1 of year 1970 (Gregorian) as fixed point indexed by 1.

Note: CaTTS does not require calendar specifications to contain the pre-defined `reference` type. It is rather left to the programmer if his/her calendar specification requires a connection with some "real time", or if a mere symbolic specification is sufficient for his/her intended needs. For example, planning the lectures in a teaching term at a university usually does not require a connection to "real time". However, if the pre-defined `reference` type is not used in different CaTTS calendar specifications which shall be compared, the programmer has to define a connection between the user-defined reference types of the different calendars. A user-defined reference type is nothing more than an identifier, i.e. a parameterless type constructor.

4.1.2 CaTTS-TDL: Type Definition Language

The language fragment CaTTS-TDL (for Type Definition Language) of CaTTS-DL gives rise to define *calendric data types* specific to a particular calendar – such as "working day", "Easter Monday", "exam week", or "CS123 lecture" (defining the time when the Computer Science lecture number 123 takes place).

4.1.2.1 Predicate Subtypes

In common-sense set theory, infinite sets are logically encoded by predicates: for any set A, the predicate $p : A \rightarrow \mathbb{B}$ defines the set of those elements of A that satisfy (the predicate) p. Such sets are called *predicate sets.* Examples of predicate sets are non-negative integers (in set notation $\{x : \mathbb{Z} \mid x > 0\}$) and integer lists with $n \in \mathbb{N}$ members (in set notation $\{l : \mathbb{Z}^* \mid length(l) = n\}$).

In type theory, predicate sets are used to define dependent types [Hof97] as well as to define types in specification languages of proof assistents and theorem provers [ROS98]. CaTTS uses predicate sets in different manners:

1. CaTTS uses predicate sets as a means to define time granularities like "month", "working day", "teaching term", or "exam week" as *(calendric) types* either in terms of aggregation (according to Definition 3.5) or in terms of inclusion (according to Definition 3.6) of another calendric type.

2. CaTTS uses type predicates as a means to define *conversions* between (values of) calendric types. Such conversions are expressed in terms of *conversion functions* (cf. Section 6.4), automatically generated from the types' predicates defined in a CaTTS-DL calendar specification.

Refinements: Defining finer Calendric Types. The predicate type constructor `refinement` defines calendric types finer than the pre-defined type `reference` or a user-defined reference type. Refinements must be explicit aggregation supertypes of such a reference type. For example, a CaTTS-DL type defining milliseconds is defined as a refinement of `reference` as follows:[1]

type millisecond = **refinement** 1000 @ **reference**(1);

The calendric type `millisecond` is defined as a thousandth refinement of a second; recall that in CaTTS' reference implementation the type `reference` defines the time unit "second" of the Unix operating system. `millisecond` partitions each second into intervals of 1000 milliseconds, and, in turn, each second may now be represented by an interval of milliseconds. Thus, the type `millisecond` is an *aggregation supertype* of the CaTTS-DL type `reference`. The type `millisecond` is anchored at (denoted @) second 1 (denoted `reference(1)`). In this way, `millisecond(1)` becomes the first millisecond in the interval of milliseconds specifying the first second (i.e. `reference(1)`). Thus, in CaTTS, types such as `reference` or `millisecond` induces clear integer indices of their values.

The indexing of values through type definitions in CaTTS-DL turns out to be extremely useful in practice. Examples for indexing of the calendric types defined throughout this section are illustrated in Figure 4.2, p.104. Of course, in addition to such an index one may define a time and date format for the elements of the type `millisecond` (or any other type defined in CaTTS-TDL) in CaTTS-FDL (cf. Section 4.1.3).

Aggregations: Defining coarser Calendric Types. To define calendric types which are aggregation subtypes (according to Definition 3.5) of other (user-defined) calendric types, CaTTS-DL provides with aggregation-based predicate type constructors.

For example, one can describe the calendric type "week" as the *subset* of those intervals over the type "day" having a duration of 7 days and beginning on Mondays. This can be directly expressed in CaTTS-DL as follows:

type week = **aggregate** 7 day @ day(1);

The calendric type `week` is an aggregation of days such that each of its elements corresponds to an interval over type `day` with a duration of 7 days. The first week (i.e. `week(1)`) is anchored at (denoted @) `day(1),` i.e. the first day of the interval of days *aggregated* to the element `week(1).` Any further index i can be computed relatively to this anchor by applying the type's predicate, i.e. `aggregate 7 day @ day(1)`. The calendric type `week` defined by the predicate `week:day`* $\rightarrow \mathbb{B}$ (`day`* denotes the type of intervals of days, cf. Appendix A) specifies the infinite set of those intervals constructed from day data, short day intervals, satisfying the *predicate* of type `week`. The type `week` is an *aggregation subtype* of the type `day`, written `week` $\preceq$ `day`.

In CaTTS-DL, aggregation subtypes which are partitions of other pre-defined or user-defined types in CaTTS are constructed using CaTTS' data type constructor `aggregate.` CaTTS supports the definition of aggregations with elements of different durations involving often complex conditions (e.g. Gregorian or Hebrew months), as well. For example,

[1]In this example and the following examples, identifiers of user-defined type constructors begin with lower case letters.

Gregorian months, including the leap year conditions for the month February (including the Gregorian leap-year rule), may be defined as a CaTTS-DL type as follows:

```
type month = aggregate
      31 day named january ,
      alternate month(i)
       | (i div 12) mod 4 == 0 &&
          ((i div 12) mod 400 != 100 &&
           (i div 12) mod 400 != 200 &&
           (i div 12) mod 400 != 300)    -> 29 day
       | otherwise                       -> 28 day
      end named february ,
      31 day named march ,
      30 day named april ,
      31 day named may ,
      30 day named june ,
      31 day named july ,
      31 day named august ,
      30 day named september ,
      31 day named october ,
      30 day named november ,
      31 day named december
   @ day(1);
```

The calendric type `month` is an aggregation of days such that each of its elements corresponds to an interval over type `day` with a duration of either 28, 29, 30, or 31 days within a repeating period of 12 months, i.e. the repetition of months in Gregorian years.

As demonstrated in this example, the different months defined can be named using CaTTS' reserved word `named`. The named "subsets" of the type `month` are locally defined types. They actually define *inclusion subtypes* of the type `month`. The leap-year rule for the Gregorian calendar is directly defined using a CaTTS `alternate`-expression to define the type `february`. The `div` operator allows to refer to the year-index the corresponding month-index is contained in. For example, `month(23)` is contained in `year(1)` (i.e. `23 div 12 == 1`) according to the type declaration for type `month` as specified above.

The first month (i.e. `month(1)`) is anchored at (denoted @) `day(1)`, i.e. the first day of the interval of days *aggregated* to the element `month(1)`. Any further index i can be computed relatively to this anchor by applying the type's predicate following the keyword `aggregate`.

Aggregation subtypes defining only a portion of a partition of the base time line like "weekend", "working year", or "academic term" may be constructed in two different manners using CaTTS-TDL:

First, if such aggregation subtypes define periodic calendric types (with finite many exceptions), i.e. types which can be defined by a periodically repeating pattern of intervals of their aggregation supertype, like "weekend", CaTTS' previously introduced `aggregate` type constructor may be used. For example, assuming that a CaTTS type `weekend_day` is defined by the predicate set specifying those days which are either Saturdays or Sundays, the CaTTS' type `weekend` may be defined as follows:

```
type weekend = aggregate 2 weekend_day @ weekend_day(1);
```

The calendric type `weekend` is an aggregation of weekend days such that each of its elements corresponds to an interval over type `weekend_day` with a duration of 2 weekend days. The first weekend (i.e. `weekend(1)`) is anchored at (denoted @) `weekend_day(1),` i.e. the first weekend day of the interval of weekend days *aggregated* to the element `weekend(1)`. The internal indexing of the type `weekend` is specified in the same manner as the indexing of the previously defined type `week` by CaTTS' language processor (cf. Figure 4.2, p.104).

Second, if such aggregation subtypes define non-periodic calendric types like "working week" which cannot be defined by a periodically repeating pattern of (continuous) intervals of their aggregation supertype, CaTTS-DL's predicate type constructor "#<" (read as "restricted aggregation") may be used. This type constructor is a set-aggregation based operation. Its definition refers to the aggregation of predicative sets. For example, assume that a CaTTS type `working_day` is defined by a predicate set specifying those days which are neither weekend days nor holidays, the CaTTS' type `working_week` may be defined as follows:

```
type working_week = week #< working_day;
```

The calendric type `working_week` is a restricted aggregation of working days into weeks such that each of its elements correspond to an interval over type `working_day` with varying durations but aligned to the duration of weeks. Constructing working weeks in this way in CaTTS-DL, the internal indexing of the type `working_week` is aligned to that of the type `week` (cf. Figure 4.2, p.104).

Let us turn our attention back to the anchor used when defining aggregation subtypes in CaTTS-DL using the predicate type constructor `aggregate.` In all previously considered examples, the anchor has been defined *absolutely* (denoted @, read "anchored absolutely to") to that of the corresponding aggregation supertype. Besides absolute anchoring, CaTTS provides the possibility to define the anchor *relatively* (denoted ~@, read "anchored relatively to") to that of the corresponding aggregation supertype. For example, to align the indexing of Gregorian years with the actual year numbering in use today, the anchor of the type `year` can be shifted such that the year previously indexed by 1 (1972, if a connection to the `reference` type is assumed) is indexed by 1970 using CaTTS' relative anchoring construct as follows:

```
type year = aggregate 12 month @ month(1) ~@ 1970;
```

Inclusions: Defining Calendric Types by selecting Subsets. To define calendric types which are inclusion subtypes (according to Definition 3.6) of other (user-defined) calendric types, CaTTS-DL provides with inclusion-based predicate type constructors.

For example, one can describe the calendric type "weekend day" as the *subset* of those elements of type "day" that are either Saturdays or Sundays. This can be directly expressed in CaTTS-DL as follows:

```
type weekend_day = select day(i) where
       relative i in week >= 6 && relative i in week <= 7;
```

or alternatively as follows:

```
type saturday = select day(i) where relative i in week == 6;
type sunday = select day(i) where relative i in week == 7;
type weekend_day = saturday | sunday;
```

depending whether the programmer wants to refer not only to weekend days but also to Saturdays and Sundays.

Considering the first possibility to model the calendric type `weekend_day` in CaTTS-DL, it is defined as a selection of those days which are between the 6th and the 7th day in each week (denoted `relative i in week >= 6 && relative i in week <= 7`).

Considering the second possibility to model the (calendric) type `weekend_day` in CaTTS-DL, it is defined as a union (denoted |) of the types `saturday` and `sunday`. The type `saturday` is a selection of the 6th day in each week (denoted `relative i in week == 6`) and `sunday` a selection of the 7th day in each week. (Assuming that the day with index 1 is a Monday.) The elements belonging to those types must thus satisfy those predicates. The elements of the previously defined type `weekend_day` must satisfy either the predicate of type `saturday` *or* ("|", read as "or") the predicate of type `sunday`. The types `saturday, sunday,` and `weekend_day` are *inclusion subtypes* of the type `day`, written `saturday` $\subseteq$ `day, sunday` $\subseteq$ `day,` and `weekend_day` $\subseteq$ `day`. Furthermore, since the type `weekend_day` is constructed by union form types `saturday` and `sunday` and since this type must satisfy either of those two types' predicates, `saturday` and `sunday` are inclusion subtypes of `weekend_day,` as well.

Figure 4.2, p.104 illustrates the indexing of the previously defined inclusion subtypes `saturday, sunday,` and `weekend_day`. As illustrated, the indexing of inclusion subtypes is successive. The indexing of any inclusion subtype is implicitly related to that of its inclusion supertype, appropriately choosing the element indexed by 0 relative to that one of the supertype. The indexing of the values of some inclusion subtype can be computed from the type's predicate.

CaTTS-DL provides different inclusion (predicate) subtype constructors, presented and discussed in the following. The `select` type constructor (previously used to define the types `saturday`, `sunday`, and `weekend_day`) can be used to define a predicate subtype of some type τ by specifying a predicate over the elements of type τ using (conjunctions) of pre-defined constraints supported in CaTTS-DL. See Appendix A for the supported syntactic forms. Additionally, three general set-theoretic operations, in particular union, intersection, and set difference, are supported as predicate type constructors, written "|" (read as "or"), "&" (read as "and"), and "\" (read as "except"). Assuming that a type `holiday` is defined as a CaTTS type, the following types can be defined using those set-theoretic predicate type constructors:

```
type holiday_or_sunday = sunday | holiday;
type holiday_on_sunday = sunday & holiday;
type weekday = day \ weekend_day;
```

The calendric type `holiday_or_sunday` defines the set of those days which are *either* holidays *or* Sundays or *both*. The calendric type `holiday_on_sunday` defines the set of those days which are both, Sundays *and* holidays. Finally, the calendric type `weekday` defines the set of days *except for* weekend days.

Note: Those predicate type constructors for disjunctions, conjunctions, and exceptions may only be applied to inclusion subtypes that have a common inclusion supertype (e.g. `day` in the previously illustrated examples). This restriction is enforced with CaTTS' type checker that tests well-formedness of CaTTS-DL calendar specifications (cf. Chapter 6).

Groups: Defining Collections of Inclusion Subtypes. CaTTS-DL allows for defining *groups* which are named collections of inclusion subtype definitions introducing less specific "kinds" a type explicitely belongs to. Group identifiers can be referred to in the same way as type identifiers of types defined in a CaTTS-DL calendar specification.

For example, one can describe the (calendric) type of weekend days not only by a CaTTS' predicate type as previously illustrated, but also as a collection of the types `saturday` and `sunday` in terms of a CaTTS-DL group. This can be directly expressed in CaTTS-DL using the `group` constructor as follows:

```
group weekend_day =
        with select day(i) where relative i in week == j
             type saturday where j == 6
             type sunday   where j == 7
        end
```

The group `weekend_day` collects (denoted by `with ...end`) the (locally) defined types `saturday` and `sunday`. The group `weekend_day` is defined as an inclusion subtype of type `day` using CaTTS' `select ...where ...` construct, however with a syntactic form different from the "usual" inclusion subtype constructions in CaTTS-DL: the (user-specified) predicate following the reserved word `where` may contain place holders for constraints (following the reserved word `where`) and/or for types (following the reserved word `for`) instantiated with concrete values for each of the types contained in the respective group. In the previously mentioned example, the specified predicate (i.e. `relative i in week == j`) contains the constraint place holder `j` instantiated for type `saturday` with `6` and for type `sunday` with `7`.

Kinding. CaTTS' typing features considered so far provide a means to define arbitrary calendric types satisfying Definition 3.2 using type constructors (e.g. the type constructors

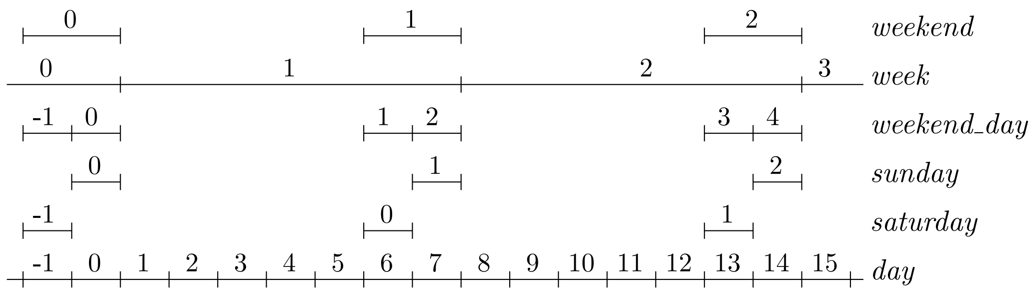


Figure 4.2: Indexing of the types defined in Section 4.1.2.

for predicate types like `reference, aggregate,` and &) defining either aggregation subtypes or inclusion subtypes. The elements of such types are (time) points with calendric type such as "20.05.2005" of type `day`. Such elements are referred to as *events* in CaTTS. The type of an event is denoted τ for every calendric type defined in a CaTTS-DL calendar specification. We say that events have *kind* point.

In many applications, one would not only refer to events but also to more complex sets of time like "my two weeks summer holidays" or "4 years". To refer to such temporal objects in some program or specification using CaTTS-DL, CaTTS provides in addition to kind point, the kinds *interval* and *duration.*

Intervals: To define intervals (i.e. finite continuous sequences of elements of any calendric type), the kinding operator * (read as "interval of") is useful for CaTTS. For every calendric type τ (defined in CaTTS-DL), the type τ^* describes the set of those intervals whose elements are drawn from τ. For example, `day`* is the type of intervals over the type `day`, and "my summer holidays in 2004" is an element of this type. Such elements are referred to as *tasks* in CaTTS.

CaTTS provides different value constructors with kind interval: An interval constructed by its "ending points" has the form $[t_1..t_2]$ with t_1 and t_2 both of type τ, e.g. "[09.12.2004..14.12.2004]". An interval constructed by its "starting point" (resp. "ending point") and a duration has the form *d downto t* (resp. *d upto t*) with t of type τ and d a duration over type τ, e.g. "4 working_day downto 09.12.2004" (resp. "4 working_day upto 14.12.2004"). The supported syntactic forms are summarized in Appendix A.

Durations: To define durations (i.e. the length of convex sequences of elements of any calendric type), the kinding operator ^{n} (read as "duration of") is useful for CaTTS.

For every calendric type τ (defined in CaTTS-DL), the type τ^n describes the set of those durations whose elements are drawn from τ. For example, `day`^{n} is the type of durations of days, and "the duration of my summer holidays in 2004" is an element of this type. Such elements are referred to as *durations* in CaTTS.

CaTTS provides a single value constructor for elements of duration types of the form $n\ \tau$ where n is a natural number with $n > 0$ and τ is the identifier of a calendric type defined in a CaTTS-DL calendar specification. For example, the element `2 working_day` has type `working_day`^{n}. The supported syntactic forms are summarized in Appendix A.

Note: Possible extensions of CaTTS with further kinding operators, for example for generalized time intervals and appropriate value constructors are discussed in Chapter 7.

4.1.2.2 Calendar as Type

In programming languages like Modula 2, C, and SML modules are used as a means for *documentation* of programs, *abstraction* of implementations, *maintenance* of large systems, and *reusability* of software. Modules defined by DTDs or XML Schemas are used for similar reasons. Furthermore, one might conceive any concrete problem implemented in OWL or any other ontology language like an ontology of time as a (reusable) module, however without providing any means of abstraction. CaTTS uses the ideas of modular programming and abstraction by interpreting *"calendars as types"* for the following purposes.

- Define different *views* of a calendar specification by specifying different calendar functions for this calendar.
- *Reuse, combine,* and *extend* calendars specified in CaTTS-DL in any (Semantic) Web language if the language uses calendar data typed after CaTTS-DL calendar specifications.
- *Context aware* interpretation of calendar data typed after calendars specified in CaTTS-DL.

A CaTTS calendar specification (or a finite set of aligned CaTTS-DL calendar specifications) is a calendar in terms of Definition 3.8, p.90. The basic entities of CaTTS-DL calendar specifications are *calendars, calendar signatures,* and *calendar functions.*

Calendars. A calendar is a packaged, finite collection of CaTTS type definitions and calendar specifications, assigning types to type identifiers, groups to group identifiers, and calendars to calendar identifiers, similar to ML structures [MTH90] or XML documents [W3C04a]. The types and calendars specified in a calendar are delimited by the keywords `cal` and `end`. The following specification binds a calendar to the identifier `Cal`. This calendar defines an environment mapping `weekend_day, week,` and `weekend` to their respective group and type definitions.

```
calendar Cal =
 cal
   group weekend_day =
     with select day(i) where relative i in week == j
       type saturday where j == 6
       type sunday   where j == 7
```

```
      end
    type week = aggregate 7 day @ day(1);
    type weekend = week #< weekend_day;
  end
```

The reserved word `cal` appears redundant in the previous example. It is useful for both the programmer and the parser in case of nested calendar specifications.

The identifiers in a calendar are qualified. For example, the qualified identifier `Cal.week` refers to the component `week` in the calendar definition `Cal`.

Calendar Signatures. A calendar signature is a kind of "type" for a calendar defined in CaTTS-DL, similar to ML signatures [MTH90] or XML Schema declarations [W3C01]. Calendar signatures specify identifiers and abstract types in terms of inclusions or aggregations for each of the components of a calendar implementing the signature. The following specification binds a calendar signature to the identifier `SIG`.

```
calendar type SIG =
  sig
(*"c:" denotes the inclusion relation "⊆" in CaTTS-syntax*)
    group weekend_day c: day;
(* "<:" denotes the aggregation relation "⪯" in CaTTS-syntax*)
    type week <: day;
    type weekend <: weekend_day;
  end
```

This calendar type definition describes those calendars having a group `weekend_day`, where each type belonging to must be an inclusion subtype of `day`, and types `week` as aggregation subtype of `day` and `weekend` as aggregation subtype of `weekend_day`. Since the calendar `Cal` introduced above satisfies this calendar type definition, it is said to *match* the calendar type definition `SIG`. Note that a defined calendar usually matches more than one calendar type definition, and a calendar type definition may be implemented by more than one calendar. Calendar type definitions in CaTTS-DL may be used to define *views* of calendars due to *ascription,* i.e. specifying less components than implemented in any matching calendar. The non-specified components are then local to the underlying calendar specification.

Calendar Functions. Calendar functions are user-defined functions on calendars using a syntax similar to function declarations in many programming languages such as SML. A calendar function `HebrewWeekend` defining Hebrew weekend days can be declared in CaTTS as follows:

```
cal_fun HebrewWeekend(C:SIG):SIG =
  cal
    group weekend_day =
      with select C.day(i) where relative i in C.week == j
      type friday   where j == 5
      type saturday where j == 6
    end
end
```

The calendar function `HebrewWeekend` takes as argument any calendar `C` matching the calendar type definition `SIG`, and yields as result a calendar also matching `SIG`. When applied to a suitable calendar, the calendar function `HebrewWeekend` yields as result the calendar whose group `weekend_day` is that of (Hebrew) weekend days, i.e. Fridays and Saturdays. Furthermore, *any* type definition in `C` depending on that of `weekend_day` is changed according to the calendar function `HebrewWeekend` when applied to C. For example applying `HebrewWeekend` to the previously illustrated calendar definition `Cal` yields a "new" group `weekend_day` and the type `weekend` is changed, as well. Since the calendar function `HebrewWeekend` may be applied to *any* calendar matching the signature `SIG` (e.g. it can be applied to the Gregorian calendar if its specification matches the "type" `SIG`), the function is polymorph, thus, it defines a parameterized calendar.

CaTTS provides with this mechanism to define functions used in type definitions to ensure the advantages of modular programming. The advantages cover modularization, maintenance, and reusability of (parts of) CaTTS calendar specifications. Modules increase readability of CaTTS-DL calendar specifications.

Macros and Imports of External Functions. CaTTS allows for type definitions containing functions defined elsewhere either externally (e.g. in a programming language like Prolog, Haskell, or Java) or internally, i.e. in a CaTTS calendar specification.

Imports: Externally defined functions, e.g. a Prolog program, can be used within CaTTS type definitions by importing the respective file(s) into some CaTTS calendar specification using the reserved word `import`. For example, defining the calendric type "Pentecost", i.e. the set of days which are always 49 days after Easter Sunday, an externally defined function computing Easter Sundays may be used as follows in a CaTTS-DL calendar specification where `easterSunday` denotes the externally defined function for computing the Easter Sunday in some year:

```
import easterSunday;
...
type pentecost = select day(i) where
         shift easterSunday(j) forward 49 day == day(i);
```

CaTTS supports the import of externally defined functions to ensure that any calendric type that may appear within a cultural or professional calendar can be expressed in CaTTS.

Macros: Internal function definitions are identified by the reserved word `macro` followed by a macro identifier and its definition. The definition follow the equals sign using syntactic forms defined with CaTTS (cf. Appendix A). For example, defining the type `month` in a CaTTS calendar specification (in terms of the Gregorian calendar), an internally defined function for computing leap years may be used as follows:

```
macro isLeapYear(i) =
    i mod 4 == 0 && i mod 400 != 100 &&
    i mod 400 != 200 && i mod 400 != 300

type month = aggregate
    31 day named january,
    alternate month(i)
      | isLeapYear(i div 12) -> 29 day
      | otherwise            -> 28 day
    end named february,
    31 day named march,
    30 day named april,
    31 day named may,
    30 day named june,
    31 day named july,
    31 day named august,
    30 day named september,
    31 day named october,
    30 day named november,
    31 day named december @ day(1);
```

Definitions of more complex macros in CaTTS-DL such as Easter Sunday are given in [BHRS05].

4.1.3 CaTTS-FDL

With most applications, one would appreciate not to specify dates and times using indices of the elements of CaTTS types like `day(23)` or `second(-123)`, but instead *date formats* like “5.10.2004”, “2004/10/05”, or “Tue Oct 5 16:39:36 CEST 2004”. CaTTS-FDL provides a means to define date formats. As user-defined predicate types, user-defined date formats are defined by a set of predicates. CaTTS’ constraint solver (cf. Chapter 5) is used to compute to and from integer indices and user-defined date formats for any calendric type defined in a CaTTS-DL calendar specification.

CaTTS-FDL date formats are defined by (none-recursive) regular grammars using syntactic forms supported with CaTTS. A date format specification maps place holders to the actual (numeric or in words) representation of a date. Each date is clearly associated with the respective index of the element of the corresponding CaTTS-TDL type defined. The programmer does not have to explicitly refer to grammars, but instead straightforwardly define a date format in CaTTS-FDL. Let’s assume a CaTTS-DL calendar specification with types defining Gregorian days, months, and years. Furthermore, we assume that the CaTTS calendar specification uses CaTTS’ pre-defined `reference` type. Thus, the year “1972” is internally indexed by 1.[2] For example, a standard date format for the Gregorian days defined in the given CaTTS calendar specification according to the ISO 8601 Standard, e.g. “2004-11-12” (12th November 2004) can be defined in CaTTS-FDL as follows:

```
format STD_date:day = y '-' m '-' d where
  STD_date within year(y - 1972 + 1),
  M is 1 month,
  STD_date within M,
  m == relative index M in year ,
  d == relative index STD_date in month;
```

This CaTTS-FDL format specification binds the variable `STD_date` to an element of type `day`. The date format itself is specified by a (user-defined) pattern (in this example `y '-' m '-' d`) where `y`, `m`, and `d` represent numbers (or indices) computed according to the formulated constraints. For example, the corresponding day index of the date “1972-01-27” is computed in CaTTS as follows by evaluating the (user-defined) predicates: the first condition (`STD_date within year(y - 1972 + 1)` where y is 1972 in this example) computes for the year-index 1 the interval of days contained, i.e. `day(1)` to `day(366)`. (Note that 1972 is a leap year.) The second condition (`m == relative index M in year`) where m is 1 reduces the interval of possible days to those not only contained in the considered year but also in the considered month of this year, i.e. the day interval `day(1)` to `day(31)` remains.[3] The last condition (`d == relative index STD_date in month`) where d is 27, further reduces the interval of days so that it satisfies this condition, i.e. the day `day(27)` remains which actually is the corresponding internal index of the date “1972-01-27”. In the same manner, the date format for the day `day(27)` can be computed.

Formats are grouped into (format) catalogs specifying to which calendar type definition a set of formats can be applied. These catalogs may be nested, applying to common scoping rules. For example, one could nest a catalog for the extended ISO standard date format (having hyphens between year, month, and day identifiers) into a catalog for the ISO standard date format (having no hyphens between the identifiers) as follows:

```
catalog ISO: STD =
  cat
(* nested catalog for extended ISO formats *)
  catalog Extended =
    cat
      format STD_date:day = y '-' m '-' d where
        STD_date within year(y - 1972 + 1),
        M is 1 month,
        STD_date within M,
```

[2]If this assumption is not made, the user has to specify a respective relation to the types' indices within his/her CaTTS-FDL date format definitions.

[3]`M is 1` is a constraint that introduces a variable M of type `month` of values with duration 1.

```
        m == relative index M in year,
        d == relative index STD_date in month;
    end
  end
```

As with calendars, identifiers defined within a catalog are qualified, e.g. `ISO.Extended.STD_date` is the full name of the above format. Date formats specified in CaTTS-FDL may be imported into a program in the language CaTTS-CL, XQuery, or any other language using calendric data typed after CaTTS-DL calendar specifications by CaTTS' import mechanism for formats `use_format`.

4.2 CaTTS-CL: Constraint Language

CaTTS-CL, CaTTS' constraint language, is typed after CaTTS-DL type definitions. CaTTS-CL is a language to declaratively express a wide range of temporal and calendric problems over domains of calendric types. In particular, CaTTS-CL allows for modeling *Constraint Satisfaction Problems (CSPs)* over events and tasks. Such problems are solved by CaTTS-CL's (multi-calendar) constraint reasoner (cf. Chapter 5).

4.2.1 Specifying Constraint Problems

Given a CaTTS-DL specification of the Gregorian calendar (with types "day", "working day", and "month") and CaTTS-FDL format specifications for types "day" and "month", the following (simple) appointment scheduling problem:

A person wants to plan a meeting of three working days after April 22nd 2005 and before May 2005. A colleague's visit of 5 days must overlap with the planned meeting.

can be formulated in CaTTS-CL as follows:[4]

```
Meeting is 3 working_day &&
Meeting after "22.04.2005" && Meeting before "05.2005" &&
Visit is 5 day && Visit overlaps Meeting
```

The variable `Meeting` represents the domain of three-working-day long intervals (denoted `Meeting is 3 working_day`). The constraint `Meeting after "22.04.2005"` formulates the condition that `Meeting` starts after the day `"22.04.2005"`. The constraint `Meeting before "05.2005"` formulates the condition that `Meeting` ends before the month `"05.2005"`. The variable `Visit` represents intervals of 5 days (denoted `Visit is 5 day`). The constraint `Visit overlaps Meeting` formulates the condition that `Visit` and `Meeting` such that `Visit` starts before `Meeting` starts and ends before `Meeting` ends and such that there is at least one (working) day that is shared by both activities.

[4]In this and the following examples, constraint variables start with capital letters.

CaTTS-CL provides with language constructs to model (multi-calendar) appointment scheduling problems such as the one previously mentioned over activities (cf. Definition 3.3) with durations (cf. Definition 3.4) referring to calendric types defined in some CaTTS-DL calendar specification. Such activities (e.g. `Meeting` and `Visit`) are represented by constraint variables. The CaTTS-CL constraint `X is D` relates to the variable `X` a domain `D`. The domain `D` represents intervals (possibly with a known duration) over a calendric type. For example, the constraint `Visit is 5 day` represents the domain of all five day long intervals. In general, the domain `D` is specified by a (possibly unknown duration) of a calendric type following the constraint `is`, i.e. by the constraint `X is` τ (read as "`X` represents the domain of all intervals of values of the type τ") or by the constraint `X is n` τ (read as "`X` represents the domain of all intervals of duration `n` of values of the type τ"), where `X` is a variable, `n` a natural number, and τ a calendric type defined in CaTTS-DL. Examples for activities that are defined using such a CaTTS-CL *calendar domain constraint* are `Meeting` and `Visit` as defined in the previously given example.

In addition to the calendar domain constraints to model activities, CaTTS-CL supports several *time constraints*. Such time constraints like `before` and `overlaps` (as used in the previously given example) can be used to model "conditions" that must hold between activities like `Meeting before "05.2005"` and `Visit overlaps Meeting`. Such time constraints (usually) restrict the domains of its variables according to the formulated "conditions". The time constraints provided with CaTTS-CL are, in particular, Allen's interval algebra [All83] and additional metric relations like `shift...forward...`, e.g. to shift a day forward by 3 days. CaTTS-CL's syntax is given in Appendix A.

4.2.2 Answers and Solutions to Constraint Problems

Answers and *solutions* to problems modeled in CaTTS-CL like the one discussed in the previous section are computed by CaTTS' constraint solver (cf. Chapter 5).

An answer to a problem specified in CaTTS-CL is itself a CaTTS-CL constraint, that can no longer be simplified, i.e. no further rule of CaTTS' constraint solver can be applied. Thus, an answer corresponds to the result computed by CaTTS' constraint solver. The answer to the problem modeled in the section above is given by the following in CaTTS-CL:

```
Meeting is 3 working_day && Visit is 5 day &&
Meeting after "22.04.2005" && Meeting before "05.2005" &&
Visit overlaps Meeting &&
(begin_of Meeting) within ["25.04.2005".."27.04.2005"] &&
(begin_of Visit) within ["21.04.2005".."24.04.2005"]
```

CaTTS' constraint solver adds two new constraints to the problem which restrict the possible starting days of the variables `Meeting` and `Visit` (denoted `(begin_of Meeting) within ["25.04.2005".."27.04.2005"]` and `(begin_of Visit) within ["21.04.2005" .."24.04.2005"]`). The constraint `(begin_of Meeting) within ["25.04.2005".."27. 04.2005"]` has the following meaning: the first working day of the three working day long meeting must be either the 25th, or the 26th, or the 27th of April 2004. The constraint `(begin_of Visit) within ["21.04.2005".."24.04.2005"]` is interpreted respectively.

The user may ask the CaTTS system to compute (at least one) *solution* to the answer given above.[5] One possible solution to the answer given above in CaTTS-CL syntax is:

```
Meeting = ["25.04.2005".."27.04.2005"]
Visit  = ["22.04.2005".."26.04.2005"]
```

With the above given solution, `Meeting` is bound (using the equality sign "=") to the interval starting at (working day) 25th April 2005 and ending (three working days later) at (working day) 27th April 2005 (denoted `["25.04.2005".."27.04.2005"]`). Similarly, `Visit` is bound to the interval starting at (day) 22nd April 2005 and ending (five days later) at (day) 26th April 2005.

Note: Answers and solutions are closely related: an answer is a compact, constraint-based, representation of several solutions (e.g. the above given answer contains 3 possible solutions). In some cases, an answer might contain unsatisfiable parts.[6] As a consequence, solutions are necessary. Solutions are computed by searching (using back tracking) the answer. CaTTS-CL provides the user with the possibility to ask the system to compute one (or all) possible solutions from an answer.

4.2.3 Programs

A CaTTS-CL *program* is a finite conjunction (expressed by the keyword `&&`) of CaTTS-CL constraints Calendars defined in CaTTS-TDL are referred to by the `use_calendar` construct, data formats defined in CaTTS-FDL are referred to by the `use_format` construct, and external libraries are referred to by the `import` construct. The constraints specified in a CaTTS-CL program are delimited by the reserved words `prog` and `end`. CaTTS' complete syntax including the syntactic forms for CaTTS-CL are given in Appendix A.

4.3 Example: Modeling Calendars and Constraints in CaTTS

This section illustrates CaTTS' expressiveness to model cultural and professional calendars using CaTTS-DL and a multi-calendar temporal reasoning problem using CaTTS-CL referring to some of the calendric types specified in the example calendars.

In what follows, different cultural and professional calendars are modeled in CaTTS-DL. To clarify CaTTS' possibility to connect different calendars and thus to provide with a means for conversions between arbitrary calendars, a CaTTS-DL calendar signature is

[5]A solution to a Constraint Satisfaction Problem is an assignment of values taken from the domains of the variables, one to each variable, such that each of the specified constraints is satisfied.

[6]The reason for possible unsatisfiable parts results from constraint propagation that results in a locally consistent Constraint Satisfaction Problem (CSP) (cf. Chapter 5). To ensure (global) consistency, solutions may be computed by searching the reduced domains to some user's request.

```
calendar type STANDARD =
    sig
        type second;
        type minute       <: second;
        type hour         <: minute;
        type day          <: hour;
        type week         <: day;
        type month        <: day;
        type year         <: month;
        group day_of_week  c: day;
    end
```

Figure 4.3: A CaTTS-DL calendar signature for standard calendars.

given first. Recall that CaTTS treats a (finite) set of such aligned calendars as a single calendar specification. Any of the CaTTS-DL calendars modeled in the following matches this calendar signature. The CaTTS-DL calendar signature given in Figure 4.3, assigned to the identifier `STANDARD` describes standard calendar types that are matched by most calendars.

4.3.1 Calendar Signature

In the calendar signature `STANDARD` given in Figure 4.3, `second` is a non-further specified type identifier. `second` is intended to define a reference type. `minute` is specified as an aggregation subtype of `second` (denoted `type minute<:second`). As it is the case for the type specification of `minute`, the further common time units hour, day, week, month, and year are specified as aggregation subtypes. The group specification `day_of_week` (i.e. a finite set of related type specifications) is specified as an inclusion subtype of `day` (denoted `group day_of_week c:day`).

Any calendar that matches the CaTTS-DL calendar signature `STANDARD` (cf. Figure 4.3) must at least declare (i.e. implement) the specified types considering the subtype specifications. Any matching CaTTS-DL calendar may, of course, enrich the CaTTS-DL calendar signature `STANDARD`.

4.3.2 Gregorian Calendar

In Figure 4.4 the standard Gregorian calendar is given in CaTTS-DL. The calendar is bound to the identifier `Gregorian` which has "type" `STANDARD` (denoted `Gregorian:STANDARD`), i.e. which matches the calendar signature for standard calendars given in Figure 4.3.

CaTTS allows for importing external libraries using the reserved word `import.` The calendar `Gregorian` imports a library `LeapSeconds,` containing, among other things, a

```
calendar Gregorian:STANDARD =
 cal
   import LeapSeconds;
   type second = reference;
   type minute = aggregate
     alternate minute (i)
       | i == 1051200 (*1.1.1972*) -> 70 second
       | hasLeapSec?(i)            -> 61 second
       | otherwise                 -> 60 second
     end @ second(1);
   type hour = aggregate 60 minute @ minute(1);
   type day = aggregate 24 hour @ hour(1);
   type week = aggregate 7 day @ day(-2);
   type month = aggregate
     31 day named january,
     alternate month(i)
       | (i div 12) mod 4 == 0 &&
         ((i div 12) mod 400 != 100 &&
          (i div 12) mod 400 != 200 &&
          (i div 12) mod 400 != 300)   -> 29 day
       | otherwise                     -> 28 day
     end named february,
     31 day named march,
     30 day named april,
     31 day named may,
     30 day named june,
     31 day named july,
     31 day named august,
     30 day named september,
     31 day named october,
     30 day named november,
     31 day named december @ day(1);
   type year = aggregate 12 month @ month(1) ~@ 1970;
   group day_of_week =
     with select day(i) where relative i in week == j
        type monday    where j == 5
        type tuesday   where j == 6
        type wednesday where j == 0
        type thursday  where j == 1
        type friday    where j == 2
        type saturday  where j == 3
        type sunday    where j == 4
     end
 end
```

Figure 4.4: The standard Gregorian calendar in CaTTS-DL.

boolean function `hasLeapSec?` over minutes. Leap second insertion into UTC-time (recall that `reference` is the time granularity of UTC-seconds) has started in 1972 (Gregorian); 10 leap seconds have been inserted into the first minute of the year 1972. In the present CaTTS-DL modeling, the index of this minute is directly referred to. The type identifier `second` is assigned to the predefined base type `reference.` The rules for the Gregorian leap month February are expressed by a suitable combination of operations predefined in CaTTS. The group `day_of_week` defines in a set the types for the seven Gregorian weekdays. The weekdays are defined by CaTTS' "relative in" predicate. The predicate of the group `day_of_week` defined after the keyword `where` selects specific elements `i` of type `day` such that they are the j^{th} element relative in a week (denoted `relative i in week == j`). `j` is a placeholder that is substituted for each of the cases for the weekdays. For simplicity reasons, the internal indexing of this calendar is chosen relative to the Unix epoch ("1 January 1970" (Gregorian)). Any further type definition is straightforward following the rules of the Gregorian calendar [DR01].

4.3.3 Hebrew Calendar

In Figure 4.5 the standard Hebrew calendar is given in CaTTS-DL. The calendar is bound to the identifier `Hebrew` which has "type" `STANDARD` (denoted `Hebrew:STANDARD`), i.e. which matches the calendar signature for standard calendars given in Figure 4.3.

The Hebrew day "23 Tevet 5730", a Wednesday (yom revii) is the day corresponding to the Unix epoch ("1 January 1970" (Gregorian)). To implement an alignment of Hebrew regaim[7] and halaqim[8], i.e. Hebrew partitions of hours with CaTTS' `reference` type (and thus with the Gregorian calendar defined in CaTTS-DL which is given in Figure 4.4) a (shifted) refinement of CaTTS' pre-defined reference type `reference` is defined (denoted by `_ref = refinement 114 @ reference(-43199)`). The type identifiers `second` and `minute` are only used to match the Hebrew calendar with the standard signature `STANDARD`[9]. These two types define regaim and halaqim: rega is 5/114 usual second (denoted `type _ref = refinement 114 @ reference(-43199)` and then `type rega = aggregate 5 _ref @ _ref(1)`) and a heleq aggregates 1080 regaim (denoted `aggregate 1080 heleq @ heleq(1)`). Further types are defined straightforward following the rules of the Hebrew calendar according to those suggested in [DR01]. Note that Hebrew weeks start on Sundays (yom rishon), and that the first month in any Hebrew year is Tishri. Since Hebrew leap year computations depend on the Metonic cycle aligning 19 sun-based years to 235 lunar-based months, the index 1 for Hebrew months is respectively moved from "Tevet 5730" to "Nisan 5720" by resetting this month's index (133) relatively to the index 1 (denoted `~@133`).

Now let us turn our attention to the functions `newYearDelay?`, `isLeapAdarRishon?`,

[7]Plural of rega.

[8]Plural of heleq.

[9]Whether such an alignment of the Hebrew calendar to the `STANDARD` calendar signature is appropriated or not, does not have to be discussed here. The present example aims at showing that it is possible and easy to express with CaTTS.

```
calendar Hebrew:STANDARD =
 cal
   type _ref = refinement 114 @ reference(-43199);
   type rega = aggregate 5 _ref @ _ref(1);
   type heleq = aggregate 76 rega @ rega(1);
   type second = rega;
   type minute = heleq;
   type hour = aggregate 1080 heleq @ heleq(1);
   type day = aggregate 24 hour @ hour(1);
   type week = aggregate 7 day @ day(-2);
   type month = aggregate
     30 day named nisan,
     29 day named iyyar,
     30 day named sivan,
     29 day named tammuz,
     30 day named av,
     29 day named elul,
     30 day named tishri,
     alternate month(i)
       | newYearDelay?(i) == 2 -> 30 day named long_marheshvan
       | otherwise              -> 29 day named short_marheshvan
     end named marheshvan,
     alternate month(i)
       | newYearDelay?(i) > 0 -> 30 day named long_kislev
       | otherwise             -> 29 day named short_kislev
     end named kislev,
     29 day named tevet,
     30 day named shevat,
     alternate month(i)
       | isLeapAdarRishon?(i) -> 30 day named adar_rishon
       | otherwise none
     end,
     29 day named adar_sheni @ day(-21) ~@ 133;
   type adar = adar_rishon | adar_sheni;
   type year = aggregate
     alternate year(i)
       | isHebrewLeapYear?(i) -> 13 month
       | otherwise             -> 12 month
     end @ month(7) @ month(7);
   group day_of_week =
     with select day(i) where relative i in week == j
        type yom_rishon   where j == 5
        type yom_sheni    where j == 6
        type yom_shelishi where j == 0
        type yom_revii    where j == 1
        type yom_hamishi  where j == 2
        type yom_shishi   where j == 3
        type yom_shabbat  where j == 4
     end
 end
```

Figure 4.5: The standard Hebrew calendar in CaTTS-DL.

and `isHebrewLeapYear.` Such functions can be implemented in CaTTS-DL using the `macro` construct. Any macro that is defined in a CaTTS-DL calendar can be used in the same manner as any CaTTS-DL type defined by accessing indices. In the following CaTTS-DL macro definitions for the three previously addressed functions used to model the Hebrew calendar given in Figure 4.5 are specified:

```
macro newYearDelay?(y) = if ny2 - ny1 == 356 then 2
                         else if ny1 - ny0 == 382 then 1
                         else 0 where
                                 ny0 = y2ed?(y - 1),
                                 ny1 = y2ed?(y),
                                 ny2 = y2ed?(y + 1);
macro y2em?(y) = (235*y - 234) div 19;
macro m2y?(m) = if y2em?(y) == em then y
                else y + 1 where
                             em = m + 70852,
                             y  = (19*em + 234) div 235;
macro y2ed?(y) = if 3*(d + 1) mod 7 < 3 then d + 1
                 else d where
                          em = y2em?(y),
                          ep = 12084 + 13753*em,
                          d  = 29*em + (ep div 25920);
macro isLeapAdarRishon?(m) =
  isLeapAdarRishonInCycle?((m mod 235) + 1);
macro isLeapAdarRishonInCycle?(i) =
  i ==  36 || i ==  73 || i ==  98 || i == 135
  || i == 172 || i == 209 || i == 234;

macro isHebrewLeapYear?(y) = (7*y + 1) mod 19 < 7;
```

The new year delay (denoted `newYearDelay?(y)`) is computed for Hebrew years `y`. This macro uses three further macros: `y2em?(y)` computes the number of elapsed months before the year indexed by `y`. `m2y?(m)` computes the year that contains an elapsed month `month(m)`. Finally, `y2ed?(y)` computes the number of elapsed halaqim before the year indexed by `y` where `em` denotes the number of months elapsed and `eq` denotes the number of part halaqim elapsed. The macro `isLeapAdarRishon?(m)` computes the last month in a Hebrew year which depends on the Metonic cycle. The macro `isHebrewLeapYear?(y)` merely applies the Hebrew leap year rule on Hebrew years.

4.3.4 An Academic Calendar

In the following, an enrichment of standard calendars matching the CaTTS-DL calendar signature `STANDARD` (cf. Figure 4.3) modeling (fictive) academic calendar types is modeled in CaTTS-DL. The calendar function given in Figure 4.6 binds the identifier `Academic` to CaTTS-DL calendars which match the calendar signature `STANDARD` (denoted `cal_fun Academic(C:STANDARD):STANDARD`).

```
cal_fun Academic(C:STANDARD) : STANDARD =
 cal
   type term = aggregate
                  6 C.month named winter_term,
                  6 C.month named summer_term
               @ C.month(10);
   type term_break = select C.month(i) where relative i in term == 6;
   type c123_lecture = select C.hour(i) where
                         (relative i in C.monday) >= 8 &&
                         (relative i in C.monday) <= 10;
   type c0815_lecture = select C.hour(i) where
                         (relative i in C.wednesday) >= 14 &&
                         (relative i in C.monday) <= 17;
   group exam_week = with select C.week(i) where max(relative i in P)
      type winter_exam for P = winter_term
      type summer_exam for P = summer_term
   end
   type weekend_day = C.saturday | C.sunday;
   group holiday = with select C.day(i) where (relative i in M) == j
      type new_year   where j == 1  for M = C.january
      type all_saints where j == 2  for M = C.november
      type christmas  where j == 25 for M = C.december
   end
   type day_off = weekend_day | holiday;
   type working_day = C.day \ day_off;
 end
```

Figure 4.6: An academic calendar in CaTTS-DL.

```
(* CET: Central European Time, GMT + 1 hour *)
cal_fun CET(C:STANDARD) : STANDARD =
     cal
       type day = aggregate 24 C.hour @ C.hour(2);
     end
(* EET: Eastern European Time, GMT + 2 hours *)
cal_fun EET(C:STANDARD) : STANDARD =
     cal
       type day = aggregate 24 C.hour @ C.hour(3);
     end
```

Figure 4.7: Calendars in different time zones in CaTTS-DL.

The calendar function `Academic` specifies CaTTS-DL calendars defining (in addition to the types defined in any calendar C matching `STANDARD`) academic calendar notions. The type `term`, an aggregation subtype of type `C.month` (i.e. the type `month` which is defined in the calendar `C`) defines winter terms and summer terms, each of a duration of 6 months where winter terms always begin in October (denoted `@ C.month(10)`). The type `term_break` denotes those months which are the 6^{th} in each term, i.e. March and September, denoted `select C.month(i) where relative in in term == 6`, specifying an inclusion subtype of type `C.month`. The group `exam_week` represents those examination weeks which are the last weeks in each term (denoted `max(relative i in P)` where `P` either refers to winter terms or to summer terms). The computer science lecture with identifier `cs123_lecture` is an inclusion subtype of type hour, selecting the hours between 8 a.m. and 10 a.m on each Monday. Similarly, the computer science lecture with identifier `cs0815_lecture` is an inclusion subtype of hour, selecting the hours between 2 p.m. and 5 p.m. on each Wednesday. Weekend days are all Saturdays and Sundays. The group `holidays` defines some holidays common in western countries. The type `day_off` specifies those days which are either holidays or weekend days (denoted `(weekend_day|holiday)`, and finally, working days are those days which are not free days (denoted `C.day\day_off`).

4.3.5 Time Zones

Since CaTTS-DL's pre-defined type `reference` corresponds to UTC-time and since UTC-time is adjusted to the time zone Greenwich Mean Time (GMT), the previously modeled calendars `Gregorian` (cf. Figure 4.4) and `Hebrew` (cf. Figure 4.5) correspond to this time zone.

Any further calendar C also matching the CaTTS-DL calendar signature `STANDARD` (cf. Figure 4.3), but which should refer to another time zone can be expressed by a (user-defined) CaTTS-DL calendar function, redefining the definition of type `day` by choosing suitable anchors for the considered time zones (cf. Figure 4.7). If the calendar function

`EET` is applied for example to the calendar `Gregorian` or to the calendar `Hebrew,` then any (aggregation and/or inclusion) subtype of `day` is respectively changed.

4.3.6 Date Formats

Apart from calendars themselves, formats used to render dates of the types of such calendars also depend on culture, in particular in some common-sense use. For example the order of units in a date format is frequently determined by the natural language they are used in, for example

(US)	“November fifth, two thousand three”	→	“11/5/2003”
(F)	“Le cinq novembre deux mille trois”	→	“5/11/2003”
(D)	“Fünfter November Zweitausenddrei”	→	“5.11.2003”
(J)	“Nisensannen jūichigatsu itsuka”	→	“2003.11.5”

Roughly speaking, formats could be (and are in CaTTS-FDL) constructed from numbers and delimiters, both of which are of arbitrary representation and order. CaTTS-FDL supports definition of any such format in an elegant and intuitive manner.

A definition of a so-called “catalog” of formats in CaTTS-FDL begins with the keyword `catalog`. This is followed by an identifier and the actual collection of formats (within `cat` and `end`). The definition given in Figure 4.8 binds a catalog of four formats to the identifier `SampleCatalog`.

Every format is introduced by the keyword `format` immediately followed by an identifier and a type signature, then by a format description (consisting of a series of identifiers and/or string constants) and finally by a list of constraints relating the format to the intended value. Inside these constraints, the identifier of the format serves as a constraint variable representing the value of the format. For example, the corresponding CaTTS internal day index of the US date “01/24/2005” is computed in CaTTS by inferring the constraint specified in the format `US` in Figure 4.8 for each of the values of the identifiers `m`, `d`, and `y`.

4.3.7 Multi-Calendar Appointment Scheduling Problem

Assume that a student’s fellow wants to visit his colleague during the term break of the running term for 4 days such that the visit contains at least one weekend day (i.e. a Saturday or a Sunday) and that he leaves at least one week before the examination week starts. This scheduling problem (that refers to the Gregorian and the academic calendars as previously specified in CaTTS-DL) can be straightforwardly expressed in CaTTS-CL as follows:

The program identifier `Visit` is bound to the CaTTS-CL program within the keywords `prog` and `end`. The program refers to two different CaTTS-DL calendar specifications,

```
catalog SampleCatalog =
 cat
  format US:day = m "/" d "/" y where
     US within year(y),
     M is month && US within M,
     m == relative index M in year,
     d == relative index US in month;
  format French:day = d "/" m "/" y where
     French within year(y),
     M is month && French within M,
     m == relative index M in year,
     d == relative index French in month;
  format German:day = d "." m "." y where
     German within year(y),
     M is month && German within M,
     m == relative index M in year,
     d == relative index German in month;
  format JapanDot:day = y "." m "." d where
     JapanDot within year(y),
     M is month && JapanDot within M,
     m == relative index M in year,
     d == relative index JapanDot in month;
  format JapanRomaji:day = y "nen" m "gatsu" d DayReading where
     JapanRomaji within year(y),
     M is month && JapanRomaji within M,
     m == relative index M in year,
     d == relative index JapanRomaji in month,
     DayReading reads if d <= 10 || d mod 10 == 4 then "ka"
                      else "nichi";

(* a non-standard format for the Academic Calendar in Figure 4.6*)
  format AcademicBreak:term_break = "Term_break" y where
     AcademicBreak within year(y);
 end
```

Figure 4.8: A sample catalog of various formats in CaTTS-FDL.

```
program Visit
 prog
   use_calendar Gregorian;
   use_calendar unqualified Academic(Gregorian);
   use_format unqualified AcademicBreak;

   Visit is 4 Gregorian.day && Visit within "Term_break 2005" &&
   X is 1 weekend_day && Visit contains X &&
   Y is 1 exam_week && (shift Visit forward 7 Gregorian.day) before Y
 end
```

Figure 4.9: A multi-calendar appointment scheduling problem in CaTTS-CL.

`Gregorian` (cf. Figure 4.4) and `Academic(Gregorian)`[10] (cf. Figure 4.6). The calendars are referred to using the `use_calendar` construct. The types in the Academic calendar may be referred to using short identifiers. The constraint `Visit is 4 Gregorian.day` associates the variable `Visit` with a domain of intervals of 4 days. The condition that the visit should be during the term break of the running term is expressed by the constraint `Visit within "Term_break 2005"` where `"Term_break 2005"` is a CaTTS-FDL format `AcademicBreak` (as specified in the catalog given in Figure 4.8), the program refers to (denoted `use_format unqualified AcademicBreak`). The constraint `X is 1 weekend_day && Visit contains X` formulates the condition that the visit must contain (at least) one weekend day. Finally, the condition that the student-fellow leaves his friend at least one week before the examination week starts is denoted by the constraint `Y is 1 exam_week && (shift Visit forward 7 Gregorian.day) before Y`[11].

[10]The calendar function `Academic(C:STANDARD)` is instantiated with the Gregorian calendar specification in this program.

[11]Note that CaTTS' current implementation does not allow for subtyping between durations to avoid any form of imprecision. Thus, the duration of a week needs to be specified in terms of days in this constraint.

Chapter 5

Constraint Reasoning with Calendric Data

(Salvador Dalí, 1904 – 1989)

So far, the calendar type language CaTTS, its language constructs to declaratively define both time granularities[1] and calendars as types has been presented (cf. Chapter 4). We have introduced CaTTS' means to define multi-calendar temporal reasoning problems using time constraints on calendric data referring to such calendric types (cf. Chapter 4). Furthermore, CaTTS' underlying interval-based time model with time granularities has been formalized (cf. Chapter 3).

This chapter is dedicated to an approach to constraint-based reasoning on calendric data in CaTTS to solve multi-calendar temporal reasoning problems, in particular, *multi-calendar appointment scheduling problems* that can be expressed in the constraint language

[1]Recall that CaTTS provide with type constructors to define time granularities as calendric types by specifying predicates.

CaTTS-CL (cf. Section 4.2). CaTTS' reasoner is based on a (specific form of) "theory reasoning" (also known as "theory resolution") [Sti85, BRS05, BM05], a well-known example of which is paramodulation [RW69]. Like paramodulation ensures efficient processing of equality in resolution theorem proving, CaTTS provides the user with convenient constructs for calendric types and efficient processing of data and constraints over those types. This reasoner thus complements general purpose "axiomatic reasoning" approaches such as ontology reasoning, frequently used in Semantic Web applications. CaTTS' reasoner refers to and relies on user-defined calendric types of calendars specified in CaTTS-DL. This makes search space restrictions possible that would not be possible if calendars and temporal notions would be specified in a generic formalism such as first-order logic and processed with generic reasoning methods such as first-order logic theorem provers.

In CaTTS, multi-calendar temporal reasoning addresses the problem of finding free time slots for related activities[2] such as a meeting in the presence of various (temporal) conditions. Such problems are called appointment scheduling problems. In the case of multi-calendar appointment scheduling problems, such activities may refer to arbitrary (cultural and/or professional) calendars. Multi-calendar appointment scheduling problems can be conveniently modeled in the constraint system finite domains (FD) [FA97, MS98]. In such constraint systems, variables range over finite domains. Those constraint systems form a well-studied and useful class of constraint problems which are often called *Constraint Satisfaction Problems (CSPs)* in Artificial Intelligence. They are important because they can be used to model combinatorial problems such as scheduling, timetabling, or planning which have widespread commercial applications. Specific to the approach to solve multi-calendar temporal reasoning problems, and, in particular, appointment scheduling problems that can be expressed in a finite domain framework, reported about in this chapter, is that those problems involve different calendric types. Therefore, we suggest an extension of the type system FD to *typed finite domains*, denoted typedFD. To support conversions between domains of different calendric types, a *conversion constraint* is proposed. Conversions are supported for any pair of calendric types defined in a CaTTS-DL calendar specification. Conversions rely on and refer to the type predicates of calendric types specified in some CaTTS-DL calendar specification.

At first, this chapter recalls the basic notions and concepts of *Constraint (Logic) Programming*, in particular, CSPs, necessary to understand CaTTS' constraint solver as presented in this chapter. Subsequently, it introduces the CSPs to be solved by CaTTS' constraint solver, i.e. the problem of multi-calendar appointment scheduling by example. The necessary extensions to the constraint system FD, in particular, typed finite domains and the conversion constraint are introduced. Subsequently, the various constraints, called *calendric constraints* to model multi-calendar appointment scheduling problems of CaTTS' reasoner in this constraint system are introduced. The *constraint propagation algorithm* is

[2]The notion activity is frequently used in temporal reasoning, referring to objects with a temporal extend [Apt03, FGV05]. In CaTTS, such activities are restricted such that they can be represented by (finite) intervals over some calendric type.

given by a finite set of proof rules. A reference implementation of the algorithm in CHR (Constraint Handling Rules) embedded into Sicstus Prolog is given in Appendix B. This solver is proved to be complete. This chapter concludes with a section on the algorithm's complexity.

5.1 Constraint Programming in a Nutshell

This section can be skipped by readers familiar with Constraint Programming.

This section briefly recalls the main aspects of *Constraint (Logic) Programming*, introducing its main concepts, in particular, the class of constraint problems which are so-called *Constraint Satisfaction Problems (CSPs)* [FA97, MS98, Apt03]. It illustrates an example of a well-known Constraint Satisfaction Problem. Furthermore, a formal framework for proof rules and derivations convenient to specify constraint propagation algorithms for CSPs is introduced.

Constraint Programming is an alternative approach to programming. Constraint Programming is particularly used to solve those kinds of problems for which efficient algorithms are lacking such as computationally intractable problems. For example combinatorial problems where each variable can be at most applied to finite many values (of the variable's domain) such as scheduling, timetabling, or planning.

The main difference to conventional programming is that in Constraint Programming the problem to be solved is modeled by means of relations, expressing the relationships between (partially unknown) objects, i.e. certain conditions that must hold between such objects. Those objects are usually modeled by variables. Modeling a problem by means of such relationships allows for more flexibility and for use of such programs for different purposes. In principle, a constraint programming process is determined by generation of requirements, so-called *constraints*, and a *solution* to these constraints by means of combining general reasoning methods with domain specific reasoning methods. The former are usually techniques to reduce the search space and specific search methods such as backtracking. The latter are special purpose algorithms for specific relations between objects of a particular domain, so-called *constraint solvers* (implementing a specific *constraint propagation algorithm*), such as an implementation of the unification algorithm.

In the last decades, Constraint Programming has been successfully applied in several domains such as operations research (e.g. scheduling problems), business applications (e.g. transporting and logistics), and molecular biology (e.g. DNA sequencing).

5.1.1 Constraint Satisfaction Problems

A *Constraint Satisfaction Problem (CSP)* consists of a finite set of relations, so-called *constraints,* over some *domains.* An example of a domain is a (finite) subset of the integers. In practice, variables are used to represent those domains such that each variable is associated with a domain. A constraint over a finite sequence of variables is then a subset of the Cartesian product of the variables' domains. Relevant definitions are given in the following, taken from [FA97, Apt03].

Definition 5.1 (Constraint Satisfaction Problem (CSP)). *A* **Constraint Satisfaction Problem (CSP)** *is a finite sequence of variables* $\mathcal{X} := X_1, \ldots, X_n$*,* $n > 0$ *with corresponding domains* $\mathcal{D} := D_1, \ldots, D_n$ *together with a finite set* $\mathcal{C}$ *of constraints, each on*

a subsequence of $\mathcal{X}$.
A **constraint** *$C \in \mathcal{C}$ on $\mathcal{X}$ is a subset of $D_1 \times \ldots \times D_n$. If C equals $D_1 \times \ldots \times D_n$, C is* **solved**.
A CSP is **solved**, *if all its constraints are solved and no domain is empty, and* **failed** *either if the CSP contains the false constraint, denoted $\perp$, or if at least one of its domains is empty.*

CSPs are usually written as pairs $\langle \mathcal{C}; \mathcal{D}_\mathcal{E} \rangle$ where $\mathcal{D}_\mathcal{E} := X_1 \in D_1, \ldots, X_n \in D_n$. $X_i \in D_i$ is called *domain expression*.

Intuitively, a solution to a CSP is a sequence of legal values for all of its variables such that all its constraints are satisfied.

Definition 5.2 (Solution). *Let $\langle \mathcal{C}; \mathcal{D}_\mathcal{E} \rangle$ a CSP, and let $\mathcal{D}_\mathcal{E} := X_1 \in D_1, \ldots, X_n \in D_n$. An n-tuple $(d_1, \ldots, d_n) \in D_1 \times \ldots \times D_n$ is a* **solution** *to $\langle \mathcal{C}; \mathcal{D}_\mathcal{E} \rangle$ if for every constraint $C \in \mathcal{C}$ on the variables $X_{i_1}, \ldots, X_{i_m}$, $(d_{i_1}, \ldots, d_{i_m}) \in \mathcal{C}$.*
If a CSP has a solution, then it is **consistent**, *otherwise* **inconsistent**.

To solve a CSP, it is often transformed in a specific way until all solutions have been found, or until it is clear that no solution exists, i.e. that the CSP is inconsistent. Such transformations need to be such that their *equivalence* is preserved according to some appropriate sense.

Definition 5.3 (Equivalence of CSPs). *Let $\mathcal{P}_1$ and $\mathcal{P}_2$ two CSPs and let $\mathcal{X}$ a sequence of common variables of $\mathcal{P}_1$ and $\mathcal{P}_2$.*
$\mathcal{P}_1$ and $\mathcal{P}_2$ are **equivalent** *if*

- *for every solution d to $\mathcal{P}_1$ a solution to $\mathcal{P}_2$ exists that coincides with d on the variables in $\mathcal{X}$, and*
- *for every solution d to $\mathcal{P}_2$ a solution to $\mathcal{P}_1$ exists that coincides with d on the variables in $\mathcal{X}$.*

Note: Solved constraints can be deleted from any CSP without effect on equivalence.

In practice, one is usually interested in determining (one of) the following answers when having modeled a problem as a CSP:

- Determine whether the CSP has a solution, i.e. whether the CSP is *consistent*.
- Finding a solution (resp. all solutions) to a CSP.
- Fining an optimal solution (resp. all optimal solutions) to a CSP according to some quality measure.

In some cases, one is additionally interested in *entailment* of the CSP.

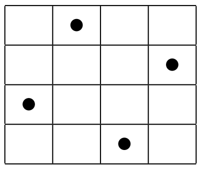


Figure 5.1: One solution to the 4 queens problem.

5.1.2 Example

Let us consider a well-known CSP, the so-called *n queens problem:* the problem is to place n queens on an $n \times n$ chess board, where $n \geq 4$, so that they do not "attack" each other. Two queens attack each other, if they are either in the same row, in the same column, or in the same diagonal. Figure 5.1 illustrates an example with a solution for $n = 4$. To represent this problem as a CSP, we use n variables $X_1, \ldots X_n$, each associated with the domain $1..n$. The idea is that the variable X_i represents the position of the queen on the i^{th} row. For example, the solution for $n = 4$ that is presented in Figure 5.1 corresponds to the sequence of values $[3, 1, 4, 2]$. The appropriate constraints can be formulated in terms of the following inequalities (over integers) for $i \in 1..n-1$ and $j \in i+1..n$:

- $X_i \neq X_j$ (i.e. no two queens in the same column)
- $X_i - X_j \neq i - j$ (i.e. no two queens in the same diagonal from the top left to the bottom right corner of the chess board)
- $X_i - X_j \neq j - i$ (i.e. no two queens in the same diagonal from the bottom left to the top right corner of the chess board)

All these constraints are inequalities. Thus, they can only be propagated if at least one of the variables is associated with a value. To enumerate possible values for the variables, a specific predicate is used. This specific predicate sequentially associates each variable with each of its possible values from its domain.

5.1.3 Proof Rules and Derivations

In what follows, a formal framework for proof rules and derivations that is used to define CaTTS' constraint propagation algorithm is introduced. Those rules allow for a straightforward implementation of the algorithm.

Proof rules are of the form

$$\frac{\phi}{\psi}$$

where $\phi := \langle \mathcal{C}; \mathcal{D}_\mathcal{E} \rangle$ and $\psi := \langle \mathcal{C}'; \mathcal{D}'_\mathcal{E} \rangle$ are CSPs, assumed that ϕ is not failed and its set of constraints is not empty. Where $\mathcal{C}$ and $\mathcal{C}'$ denote sets of constraints and $\mathcal{D}_\mathcal{E}$ and $\mathcal{D}'_\mathcal{E}$ sets of domain expressions.

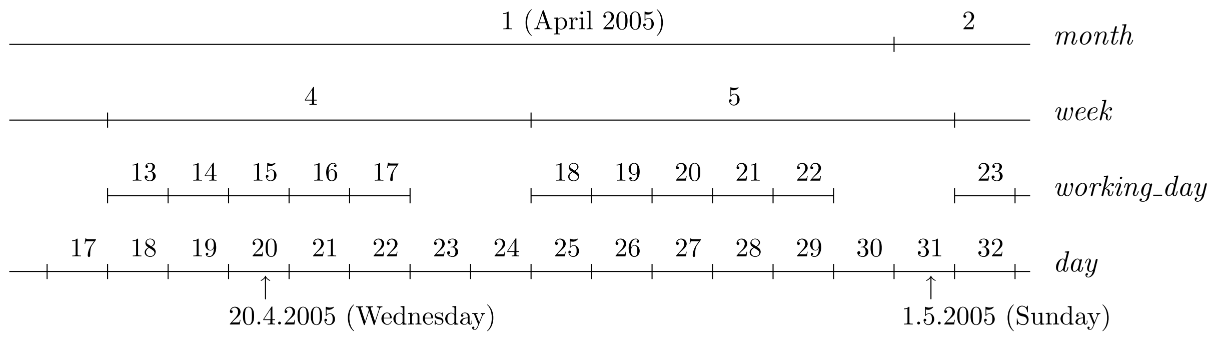


Figure 5.2: Indexing of the calendric types addressed in Example 5.1.

Proof rules are distinguished between *domain reduction rules* and *transformation rules.* In domain reduction rules, new domains are corresponding subsets of the old domains, and the new constraints are corresponding restrictions of the old constraints to the new domains. Transformation rules introduce or remove constraints without reducing domains.

Applying a proof rule to a CSP means that those parts in a CSP are replaced that coincides with the premise by the conclusion. Furthermore, applying a proof rule restricts the "old" constraints to the new domains in case the rule has reduced the domains. This is done such that possible variable clashes are prevented.

Note: From the way the proof rules of CaTTS' constraint propagation algorithm are introduced in the subsequent sections, it will be clear that all of them are *equivalence preserving* (w.r.t. a sequence of variables $\mathcal{X}$), i.e. ϕ and ψ are equivalent (w.r.t. $\mathcal{X}$) according to Definition 5.3.

5.2 Multi-Calendar Appointment Scheduling Problems

Recall that the type language CaTTS provides with language constructs to conveniently model cultural calendars like the Gregorian and Hebrew calendars or some professional calendar (e.g. an academic calendar of a university) including various calendric types like "month", "week", or "teaching term" using CaTTS' type definition language CaTTS-DL (cf. Section 4.1). Additionally, CaTTS provides with means to model *multi-calendar temporal reasoning problems*, in particular, appointment scheduling problems over calendric data referring to such types using CaTTS' constraint language CaTTS-CL (cf. Section 4.2).

An example of a multi-calendar appointment scheduling problem that can be modeled and solved in CaTTS is given in the following:

Example 5.1 (An Appointment Scheduling Problem). *A person plans a meeting lasting 3 working days after 20th April 2005 and before May 2005. A colleague's visit of 1 week must overlap with the planned meeting.*

To properly analyze and solve such a problem, we are led to an abstract analyze of *activities* that take time, such as "meeting" and "visit". In CaTTS, such activities can be identified with (time) intervals. Activities may refer to different calendric types. For example, “meeting” refers to working days (i.e. Monday to Friday)[3] while “visit” refers to weeks. Figure 5.2, p.129 illustrates the different calendric types addressed in Example 5.1. Activities are temporally related – either to calendric information, for example “the meeting must be before 20th April 2005” or relatively to each other, for example “the visit must overlap the meeting”. Such temporal relations can be conveniently expressed in terms of specific constraints which are called *time constraints* in CaTTS. In order to apply such time constraints if the related variables have different calendric types, then the variables’ domains have to be converted into domains that represent the equivalent sets of times, however in another calendric type. For example, to apply the time constraint “visit (in terms of weeks) overlaps with meeting (in terms of working days)” according to Example 5.1, then the variables’ domains have to be converted into domains that represent the equivalent sets of times, however in another calendric type. CaTTS supports such conversions by means of a specific, novel constraint, called *conversion constraint*. The conversion constraint relies on and refers to calendric types defined in some CaTTS-DL calendar specification.

Appointment scheduling problems such as that illustrated in Example 5.1 can be conveniently modeled in a *constraint network*. Constraint networks are illustrated by directed graph where the variables (with associated *(finite) domains*) are represented by nodes and the constraints by directed arcs between those variables. In case of CaTTS’ multi-calendar appointment scheduling problems, the nodes represent activities and the arcs represent time constraints that must hold between those activities. In modeling activities, we differ between *events* and *tasks* for efficiency reasons. An event X is defined by a time S_X and a calendric type τ. Domain expressions for events are represented by $X :: n..m, \tau$ where $n \leq X \leq m$. For short, we write X^τ, read as “X has kind point over calendric type τ”.[4] A task X is defined by a starting time S_X, a duration d_X[5], and a calendric type τ. Domain expressions for tasks are represented by $X :: n..m + d^-..d^+, \tau$ where $n \leq S_X \leq m$ and $n + d^- - 1 \leq E_X \leq m + d^+ - 1$. For short, we write X^{τ^*}, read as “X has kind interval over calendric type τ”. τ denotes a calendric type defined in some CaTTS-DL calendar specification.

The appointment scheduling problem described in Example 5.1 can be represented by the constraint network illustrated in Figure 5.3 according to the internal indexing illustrated in Figure 5.2, p.129: the domain expression “Meeting :: $1..\infty + 3..3$, working_day” is read as follows. The variable “Meeting” which represents a task is associated with the finite domain $1..\infty + 3..3$, meaning that the possible starting time of the meeting is any working day (denoted $1..\infty$, i.e. 1 less or equal than “start of Meeting” less or equal than ∞), represented by its internal integer index and that “Meeting” has a duration of (minimal and maximal) 3 (working days) (denoted 3..3). Possible values Meeting can be instantiated with are the interval $[1, 3]$ (i.e. $[1, 1 + 3 - 1]$), the interval $[2, 4]$, etc. The

[3]One might also define the type “working day” by excluding weekend days and holidays in CaTTS-DL and refer to those in CaTTS-CL.

[4]$\leq$ denotes the finite domain constraint “less than or equal” over integers.

[5]From those two values the ending time E_X can be easily computed: $E_X := S_X + d_X - 1$.

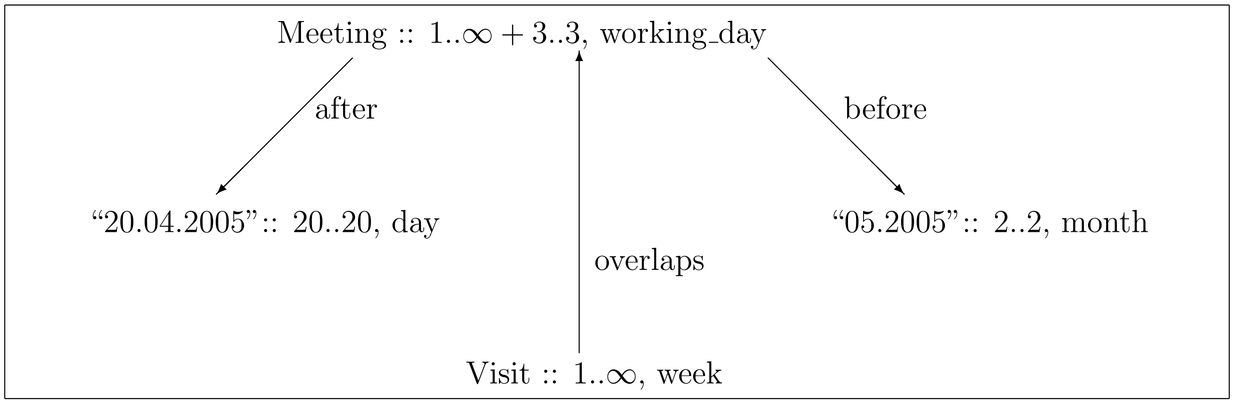

Meeting :: 1..∞ + 3..3, working_day
after
before
"20.04.2005":: 20..20, day
"05.2005":: 2..2, month
overlaps
Visit :: 1..∞, week

Figure 5.3: Illustration of the appointment scheduling problem of Example 5.1 as constraint network.

time constraint "Meeting after 20.04.2005" is modeled by a directed arrow. The domain expression ""20.04.2005":: 20..20, day" is read as follows. The variable "20.04.2005" which represents an event is associated with the finite domain 20..20, i.e. 20 less or equal than "20.04.2005" less or equal than 20, meaning that the possible day "20.04.2005" can be instantiated with is the day indexed with 20 (according to Figure 5.2, p.129). The remaining constraints and domain expressions which are illustrated in Figure 5.3 are modeled in the same manner.

The idea to solve problems that can be modeled in such constraint networks is to remove those values from the domains of the variables that do not participate in a solution to the problem.[6] That is, each time a time constraint like "after" is applied, the domains of the variables that participate in the domains are reduced. But if we try to apply the time constraint "Meeting after 20.04.2005" according to Figure 5.3, the following problem appears: "Meeting" is associated with a domain whose values are drawn from the calendric type "working_day" while "20.04.2005" refers to days; essentially, "Meeting" and "20.04.2005" have different calendric types; thus, their associated domains cannot be compared. For example, the index 20 from the domain of days refers to another extend of time (i.e. 20^{th} April 2005) than the index 20 from the domain of working days (i.e. 27^{th} April 2005) as illustrated in Figure 5.2, p.129. To overcome this problem, we have to *convert* the domains of the variables participating in a time constraint such as "after" to some appropriate "common calendric type", apply the time constraint on the variables in this common type and "relate" the domains' representation in their original types to that in the common type. This common type is inferred by CaTTS-CL's subtype checker (cf. Section 6.7). In particular, the *join* according to Proposition 3.1 is computed. In case of the constraint "Meeting after 20.04.2005" that means that the set of (intervals of) working days that is represented by the domain "1..∞ + 3..3, working_day" must be represented

[6]Recall that a solution to a CSP is an assignment of each of the variables to values from their domains such that all the constraints in the CSP are satisfied.

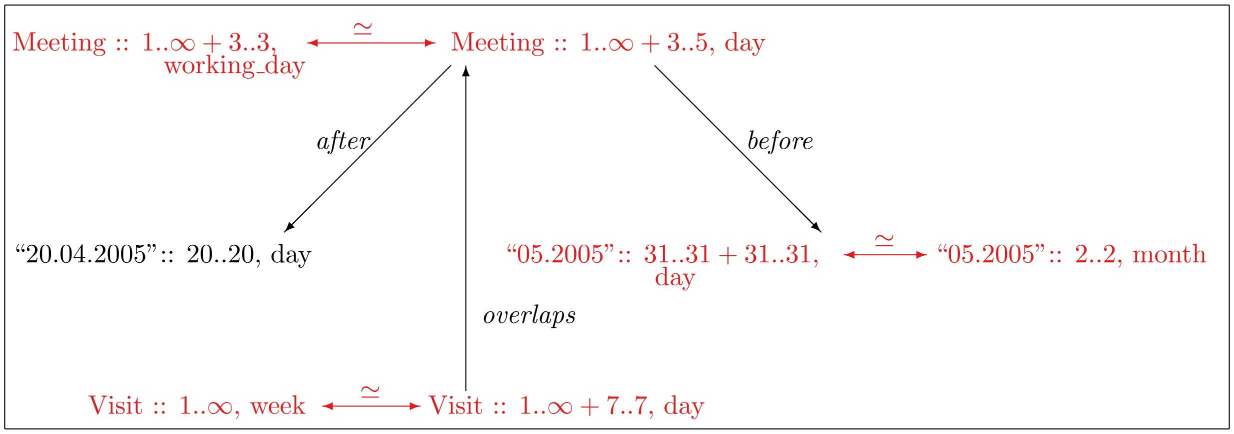

Meeting :: 1..∞ + 3..3, working_day
≃
Meeting :: 1..∞ + 3..5, day
after
before
overlaps
"20.04.2005":: 20..20, day
"05.2005":: 31..31 + 31..31, day
≃
"05.2005":: 2..2, month
Visit :: 1..∞, week
≃
Visit :: 1..∞ + 7..7, day

Figure 5.4: Illustration of the appointment scheduling problem of Example 5.1 with conversion constraints as constraint network.

by an equivalent set of time intervals, however referring to days. This kind of "equivalence relation" between variables that refer to different calendric types is supported by CaTTS' conversion constraint, denoted with $\simeq$. The appointment scheduling problem of Example 5.1 with conversions is illustrated in Figure 5.4. Now we can reduce the domains in the domain expressions in this constraint network by applying the different time constraints which always involve applications of the related conversion constraints: applying the time constraint "after" on "Meeting" and "20.04.2005", reduces the domain of "Meeting" such that its smallest possible starting time is the day "21.04.2005", i.e. Meeting :: $21..\infty + 3..5$, day[7], and, therewith, Meeting :: $16..\infty + 3..3$, working_day according to the indexing used in Figure 5.2. That is, we have removed all those days possibly starting the interval "Meeting" that do not satisfy the constraint "Meeting after 20.04.2005". We say, that the constraint "after" propagates to the Meeting that it cannot start before "21.04.2005" which, in turn propagates this information into an equivalent set expressed in terms of working days. In the same way, we proceed by applying the remaining time constraints until the CSP is not further reducible, i.e. until no further constraint propagation would reduce any of the considered domains. The not further reducible CSP is illustrated in Figure 5.5.

Once CaTTS' constraint solver has reduced a CSP such as the one defined in Example 5.1 to an equivalent CSP that is not further reducible, this result is provided as an *answer* in CaTTS-CL. The answer to the problem illustrated in Example 5.1 that would be computed by CaTTS' constraint solver (as illustrated in Figure 5.5) is given in the following:

[7]Note that the variable duration of this constraint (denoted 3..5) results from the conversion: three consecutive working days might be Friday-Monday-Tuesday. In terms of days this would be Friday-Saturday-Sunday-Monday-Tuesday, i.e. 5 days.

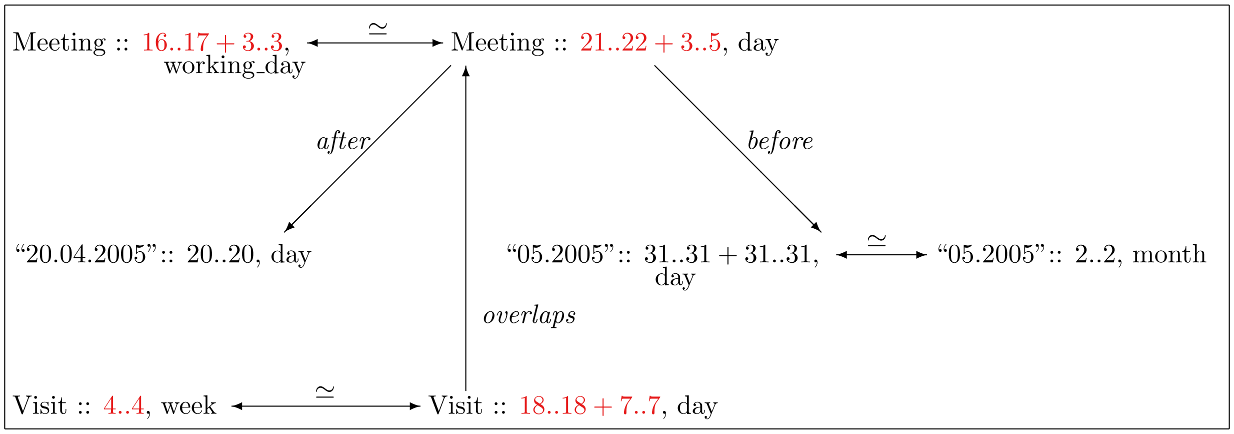

Meeting :: 16..17 + 3..3, working_day
≃
Meeting :: 21..22 + 3..5, day
after
before
"20.04.2005":: 20..20, day
"05.2005":: 31..31 + 31..31, day
≃
"05.2005":: 2..2, month
overlaps
Visit :: 4..4, week
≃
Visit :: 18..18 + 7..7, day

Figure 5.5: Illustration of the answer to the appointment scheduling problem of Example 5.1 as constraint network.

Example 5.2 (An Appointment Scheduling Problem: Answer). *The meeting lasting 3 working days must either start at working day 21st April 2005 or at working day 22nd April 2005. The 1 week visit must be the 4th week in April 2005, i.e. the week from 18th April 2005 to 24th April 2005.*

The solutions to this answer that can be requested by the user in CaTTS-CL are given in Example 5.3. A solution is computed by searching the reduced CSP using backtracking. The solutions to the considered problem are given in the following:

Example 5.3 (An Appointment Scheduling Problem: Solutions).

1. *The meeting starts at working day 21st April 2005 and ends (three working days later) at working day 25th April 2005 and the visit is the 4th week of April 2005 (i.e. the week from 18th April 2005 to 24th April 2005).*

2. *The meeting starts at working day 22nd April 2005 and ends (three working days later) at working day 26th April 2005 and the visit is the 4th week of April 2005 (i.e. the week from 18th April 2005 to 24th April 2005).*

The previously illustrated and discussed appointment scheduling problem can be modeled in CaTTS-CL as follows[8]:

```
(* the appointment scheduling problem according to Example 5.1, p.129 *)
Meeting is 3 working_day &&
Meeting after "20.04.2005" && Meeting before "05.2005" &&
Visit is 1 week && Visit overlaps Meeting
```

[8]Assuming that the date formats used are specified in CaTTS-FDL (cf. Section 4.1.3).

```
(* the answer to this problem according to Example 5.2, p.133 *)
Meeting is 3 working_day &&
(begin_of Meeting) within ["21.04.2005".."22.04.2005"] &&
Visit is 1 week && (begin_of Visit) within ["W04.2005".."W04.2005"]

(* the solutions to this problem according to Example 5.3, p.133 *)
(Meeting = ["21.04.2005".."25.04.2005"] && Visit = "W04.2005") ||
(Meeting = ["22.04.2005".."26.04.2005"] && Visit = "W04.2005")
```

5.3 The Underlying Constraint System

In the following, we consider the formal *constraint theory* and its embedding *constraint system* underlying CaTTS' constraint solver to model and to solve multi-calendar appointment scheduling problems such as the problem illustrated in Example 5.1. For the definition of the constraint system, we use an abstract syntax. A constraint system formally defines the set of *allowed constraints* along with their specifications. The constraint theory restricts the possible interpretations of the constraint symbols. In the subsequent Section 5.4, CaTTS' calendar constraints which can be expressed in terms of the allowed constraints of the underlying constraint system are introduced.

Multi-calendar appointment scheduling problems such as that illustrated in Example 5.1 can be conveniently modeled using the constraint system *FD (finite domains)* [FA97, MS98]. Since CaTTS deals with different calendric types, the original constraint system FD needs to be extended in a certain sense. In particular, the constraint system underlying CaTTS' constraint solver is based on a *typed* constraint system over finite domains including a novel kind of constraint, called *conversion constraint* to related domain expressions of different calendric types to each other. A definition of the constraint system *typedFD* underlying CaTTS' constraint solver is given in the following:

Definition 5.4 (Constraint System typedFD). *The constraint system* **typedFD** *is a triple* $(\Sigma, CT, \mathcal{C})$ *where*

- *the* **signature** Σ *consists of*
  - *function symbols: integers, lists,* $+$, *and*
  - *constraint symbols:* $=, \leq, <, >, \geq, \neq, ::, \simeq,$
- *the* **constraint theory** CT *is Presburger Arithmetic, i.e. a linear fragment of the integer arithmetic including natural numbers,* $+$, *and* $=$, *and*
- *the set* $\mathcal{C}$ *of allowed* **constraints**

$$\begin{aligned} C := \quad & true \mid false \mid X :: n..m, \tau \mid X :: [k_1, \ldots, k_l], \tau \mid X^\tau \odot Y^\tau \mid \\ & X^\tau + Y^\tau = Z^\tau \mid X^\sigma \simeq Y^\tau \mid C \wedge C \end{aligned}$$

where

- $n, m, k_1, \ldots, k_l$ are integers of type (also called sort in logics) τ, $1 \leq l$,
- $\odot \in =, \leq, <, >, \geq, \neq$,
- X^τ, Y^τ, Z^τ variables or constants over type τ, and
- τ, σ types.

Note: Presburger Arithmetic is complete and decidable. Extending the underlying constraint theory of the constraint system typedFD to Peano Arithmetic (including $*$ and induction principle) would lead to an incomplete constraint theory.

Disregarding the conversion constraint $X^\sigma \simeq Y^\tau$, the constraint system underlying CaTTS' constraint solver is a typed version of the constraint system FD [FA97, MS98]. In the constraint system typedFD, each type represents a calendric type defined in some CaTTS-DL calendar specification. The additionally introduced conversion constraint, denoted $X^\sigma \simeq Y^\tau$, is a novel constraint introduced with CaTTS' constraint solver. This conversion constraint provides with means to related domains of different calendric types to each other such that both domains refer to the same amount of time, however expressed in different calendric types. That is, the conversion constraint represents a kind of equivalence relation between domains represented in different types. Conversions become necessary whenever calendric data of different calendric types need to be compared in an appointment scheduling problem modeled in CaTTS-CL.

Let us turn attention to the domain expressions: $X :: D^\tau$ means that $X \in D^\tau$, i.e. the variable X represents the domain D^τ of values over τ. Two different representations for the domains are possible:

- $X :: [k_1, \ldots, k_l], \tau$ denotes an enumeration domain, i.e. $X = k_1 \vee \ldots \vee X = k_l$, and
- $X :: n..m, \tau$ denotes an interval domain, i.e. $n \leq X \wedge X \leq m$.

The representation of CaTTS-CL *calendar domains* in the application of the constraint system typedFD (cf. Definition 5.4) to model multi-calendar appointment scheduling problems as well as the implementation of the constraints provided with CaTTS-CL over such domains, i.e. *activity constraints*, *time constraints*, and *conversion constraints* expressed in the constraint system typedFD are introduced and discussed in the subsequent Section 5.4.

5.4 Calendric Constraints

CaTTS' constraint solver applies the constraint system typedFD (cf. Definition 5.4) to model and solve multi-calendar appointment scheduling problems which can be modeled in CaTTS-CL such as that illustrated in Example 5.1, p.129. The constraints supported by this solver are referred to as *calendric constraints* which are defined over *calendar domains*. CaTTS' constraint solver supports three different kinds of calendric constraints:

$\forall\alpha,\beta.(\alpha :: \tau \vee \alpha :: \tau^*),$
$\quad(\beta :: \sigma \vee \beta :: \sigma^*)$

$C ::=$	$true \mid false \mid X :: n..m, \tau \mid X :: n..m + d_X, \tau \mid d_X :: d^-..d^+, \tau \mid$ $i_X :: [i_1, \ldots, i_l], \tau \mid X^\alpha\ \mathcal{R}\ Y^\alpha \mid d_{X^\alpha} \odot d_{Y^\alpha} \mid \mathcal{M}\ X^\alpha d_{Y^\alpha} = Z^\alpha \mid$ $\mathcal{I}\ i_{X^\sigma}\tau \odot i \mid E\ X^\alpha = Y^\tau \mid I\ X^\tau = i_{X^\tau} \mid D\ X^\alpha = d_{X^\alpha} \mid$ $X^\alpha \simeq Y^\beta \mid C, C$
where	
τ, σ	calendric type defined in a CaTTS-DL calendar specification
X, Y, Z	variable
X^τ	variable X has kind point of type τ
X^{τ^*}	variable X has kind interval of type τ
d_X	variable associated with duration domain of activity X
i_X	variable associated with index domain of activity X
$E \in$	{*begin_of, end_of*}
$I \in$	{*index*}
$D \in$	{*duration*}
$\mathcal{R} \in$	{*equals, before, after, starts, started_by, finishes, finished_by, meets, met_by, during, contains, overlaps, overlapped_by, within, on_or_before, on_or_after*}
$\odot \in$	$\{=, <, \leq, \geq, >, \neq\}$
$\mathcal{M} \in$	{*shift_forward, shift_backward, extend_by, shorten_by*}
$\mathcal{I} \in$	{*relative_in, relative_to*}
$n, m, d^-, d^+, i_1, i_l, i$	integers

Table 5.1: The abstract syntax of the language CL$_{catts}$.

activity constraints to model calendar domain expressions (that consist of a variable and a calendar domain), *time constraints* to model temporal relations between calendar domains over the same calendric type, and the *conversion constraint* to model conversions between calendar domains of different calendric types. The conversion constraint defines an equivalence relation between the calendar domains of activity constraints of different calendric types. Those conversion constraints relay on and refer to calendric types defined in some CaTTS-DL calendar specification. In particular, the conversion constraint applies conversion functions (cf. Section 6.4) generated from the type predicates. The abstract syntax of the language of CaTTS' constraint solver, called CL$_{catts}$ that is used throughout this text is given in Table 5.1.

Constraint:	**Definition**	**Description:**
$X :: n..m, \tau$	$n \leq X \leq m$	interval domain expression for events
$X :: n..m + d^-..d^+, \tau$	$n \leq S_X \leq m \wedge$ $n + d^- - 1 \leq E_X \leq m + d^+ - 1$	interval domain expression for tasks
$S_X :: n..m, \tau$		interval domain expression for starting time S_X of X
$d_X :: d^-..d^+, \tau$		interval domain expression for duration d_X of X
$E_X :: n_E..m_E, \tau$	$n_E := n + d^- - 1,$ $m_E := m + d^+ - 1$	interval domain expression for ending time E_X of X
$i_X :: [i_1, \ldots, i_l], \tau$	$i_X = i_1 \vee \ldots \vee i_X = i_l$	enumeration domain expression for index i_X of X

Table 5.2: Activity constraints.

5.4.1 Activity Constraints

Activity constraints represent specific calendar domain expressions, denoted $\forall\alpha.\ \alpha :: \tau \vee \alpha :: \tau^*.\ X \in D^\alpha$ where X is a variable and D^α is the calendar domain that is associated to X. $\forall\alpha.\ \alpha :: \tau \vee \alpha :: \tau^*$ is a kinding expression read as "for all CaTTS types α which are either of kind point of some type τ, written $\alpha :: \tau$, or of kind interval of some type τ, written $\alpha :: \tau^*$". That means, X either refers to a time point or to a time interval in some calendric type τ defined in a CaTTS-DL calendar specification. If X refers to time points, we say that X is an *event*. Events are used to represent point-like data like "22nd April 2005" or the starting day of a week. If X refers to time intervals, we say that X is a *task*. Tasks are used to represent intervals (with specific durations) like "from 22nd April 2005 until 24th April 2005" or "the last two weeks in April 2005".[9] In particular, we only take into account the fact that activities (i.e. events and tasks) take a finite, continuous period of time, expressed in some calendric type defined in a CaTTS-DL calendar specification.[10] Recall that CaTTS' underlying time model is interval-based and discrete (cf. Chapter 3). Thus, events and tasks both refer to (convex) intervals in time. A differentiation is made in the internal presentation of the activities' domains for efficiency reasons. The activity constraints are summarized in Table 5.2.

In the case that activities have a duration and take a finite, continuous period of time according to a calendric type, they can be identified with closed, non-empty intervals of (integer) indices of calendric types defined in some CaTTS-DL calendar specification. Such intervals are written $[a, b]$ where $a \leq b$ where a is the starting point and b the ending point

[9]The notions "event" and "task" are taken from research on "planning" and "scheduling", well-known kinds of CSPs over finite domains [FA97].

[10]Constraint reasoning over possibly non-continuous activities with calendric types, for example (infinite) periodic activities like someone's consultation hours is a further (more complex) reasoning problem that might appear with multi-calendar appointment scheduling problems. Although it is an interesting, yet, unsolved constraint problem over calendar domains, it is not considered within this research project.

of the interval. If such an interval refers to an event, then $a = b$.[11]

5.4.1.1 Events

Events represent point-like calendric data such as "22.4.2005" or the starting day of a week. In particular, events represent single values of a calendric type like "22.4.2005" of type "day" or "05.2005" of type "month". Recall that in a CaTTS-CL program (cf. Section 4.2) an event can be declared as follows:

```
X is 1 day
```

This CaTTS-CL activity constraint is read as follows: the variable `X` represents an event of type `day`. That is, `X` can be assigned to *any* value of type `day` with duration 1 (day). In terms of Constraint Programming, we say that the variable X represents the domain of *days*, written $X \in D^{day}$, where $D^{day} := \{x \mid x : day\}$. Such domain constraints are implemented by (ordinary) finite domain constraints (over integers) in the constraint system typedFD which are represented in terms of interval domains. Specific to the constraint solver of CaTTS is, that such domain expressions additionally have a calendric type attached. In terms of finite domain constraints in the language CL$_{catts}$, the activity constraint $X \in D^{day}$ is represented as follows by an interval domain constraint in the constraint system typedFD:

$$X :: n..m, day, \text{ i.e. } n \leq X \wedge X \leq m.$$

where $n, m \in \mathbb{Z}$ represent indices in type *day*.

CaTTS-CL constraints to model events are transformed into such activity constraints in the language CL$_{catts}$ which can be handled by CaTTS' constraint solver. Transformation of CaTTS-CL into CL$_{catts}$ is introduced in Chapter 6.

5.4.1.2 Tasks

Tasks represent calendric data with finite durations such as a visit that lasts for five days. In particular, tasks represent finite, continuous sequences of values of a calendric type like "from April 22nd 2005 until April 24th 2005" of type "day" or "the last two weeks in April 2005" of type "week". Such sequences can be always represented by their starting points and their duration both in a calendric type, for example "from April 22nd 2005 until April 24th 2005" can be represented by its starting point "April 22nd 2005" and its duration "3 day". Recall that in CaTTS-CL (cf. Section 4.2) a task can be declared as follows:

```
X is 3 day
```

[11] Since CaTTS' underlying time model is purely interval-based, activities can be identified with closed intervals. For example, the interval "20.04.2005" meets the interval "21.04.2005" and there exists no (durationless) time point that might be true at "20.04.2005" and "21.04.2005".

This CaTTS-CL activity constraint is read as follows: the variable `X` represents a task of type `day`* with duration 3 (days). Recall that τ^* denotes the kind of intervals of type τ. That is, `X` can be assigned to *any* value of type `day`* that has a length of 3 (days). In terms of Constraint Programming, we say that the variable X represents the domain of *intervals of days of duration 3 (days)*, written $X \in D^{day^*}$, where $D^{day^*} := \{x+d \mid x : day \wedge d : day^n\}$; in this example, $d = 3$. Recall that τ^n denotes the kind of durations of type τ. Such activity constraints are implemented by (specific) finite domain constraints (over integers) in the constraint system typedFD with the following representation: starting times are represented by finite domain constraints (over integers) in the constraint system typedFD in terms of interval domains. Durations (which are added to the starting times) are also represented by finite domain constraints.[12]. Specific to the constraint solver of CaTTS is, that such calendar domain constraints additionally have a calendric type attached. In terms of finite domain constraints in the language CL$_{catts}$, the activity constraint $X \in D^{day^*}$ is represented as follows:

$$X :: n..m + d^-..d^+, day, \text{ i.e. } n \leq S_X \;\wedge\; n + d^- - 1 \leq E_X \;\wedge\; S_X \leq m \;\wedge\; E_X \leq m + d^+ - 1.$$

where $n, m \in \mathbb{Z}$ represent indices in type *day* and $d^-, d^+ \in \mathbb{Z}$ represent durations in type *day*. Recall that S_X denotes a starting time of the (time) interval represented by X and E_X an ending time of this interval with $E_X := S_X + d_X - 1$ where $S_X = n..m$ and $d_X = d^-..d^+$.

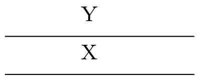
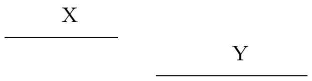
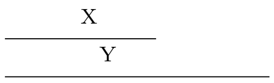
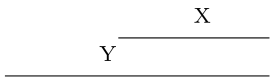
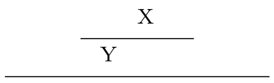
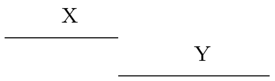
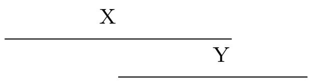
CaTTS-CL constraints to model tasks are transformed into such activity constraints in the language CL$_{catts}$ which can be handled by CaTTS' constraint solver. Transformation of CaTTS-CL into CL$_{catts}$ is introduced in Chapter 6.

5.4.2 Time Constraints

CaTTS provides with a considerably large set of *time constraints* over the calendar domains of activity constraints. Time constraints are used to specify the (temporal) conditions that must hold between different activities in a multi-calendar appointment scheduling problem such as that illustrated in Example 5.1. For example "the meeting must be *before* May 2005 and *after* 20th April 2005" and "the visit must *overlap* with the meeting 2 days". Table 5.3 summarizes the time constraints supported in CL$_{catts}$. Those time constraints are the time constraints supported by CaTTS-CL (cf. Table A.4), but on calendar domains with the same calendric type. The translation of CaTTS-CL time constraints into CL$_{catts}$ time constraints and conversion constraints is introduced in Chapter 6. CaTTS' conversion constraint is introduced in the subsequent Section.

The definitions of CL$_{catts}$ time constraints in terms of finite domain constraints in the constraint system typedFD are illustrated in Figure 5.6, Figure 5.7, and Figure 5.8: each time constraints is defined by a set of built-in arithmetic constraints $\{=, \leq, <, \geq, > \neq\}$

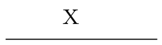
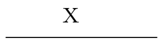
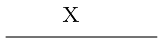
[12]Durations need to be represent by interval domains to meet conversion constraints. For example, converting months to days yields in a range of durations form 28 to 31 (days) if the Gregorian calendar is used.

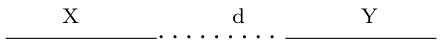
time constraints, $X, Y \in \tau$: $X :: S_X + d_X, E_X := S_X + d_X - 1$ $Y :: S_Y + d_Y, E_Y := S_Y + d_Y - 1$		**definition on end points:**
X equals Y		$S_X = S_Y \wedge E_X = E_Y$
X before Y Y after X := X before Y		$S_X \leq E_X < S_Y \leq E_Y$
X starts Y Y started_by X := X starts Y		$S_X = S_Y \leq E_X < E_Y$
X finishes Y Y finished_by X := X finishes Y		$S_Y < S_X \leq E_X = E_Y$
X during Y Y contains X := X during Y		$S_Y < S_X \leq E_X < E_Y$
X meets Y Y met_by X := X meets Y		$S_X \leq E_X = S_Y \leq E_Y$
X overlaps Y Y overlapped_by X := X overlaps Y		$S_X < S_Y < E_X < E_Y$

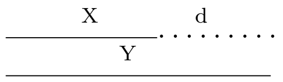
X within Y := X equals Y $\vee$ X starts Y $\vee$ X finishes Y $\vee$ X during Y

X on_or_before Y := X equals Y $\vee$ X before Y   X on_or_after Y := Y on_or_before X

Figure 5.6: Illustration of CL$_{catts}$ time constraints that define interval relations on activities.

time constraints, $X, Y \in \tau$: $X :: S_X + d_X, E_X := S_X + d_X - 1$ $Y :: S_Y + d_Y, E_Y := S_Y + d_Y - 1$		**definition on end points:**
begin_of(X,Y)		Y= S_X
end_of(X,Y)		Y = E_X
index(X,Y)		Y = i_X

shift_forward(X,d,Y)
shift_backward(Y,d,X) := shift_forward(X,d,Y)
		S_X + d = S_Y
		E_X − d = E_Y

extend(X,d,Y)
shorten(Y,d,X) := extend(X,d,Y)
		S_X = S_Y
		E_X + d = E_Y

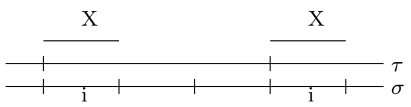
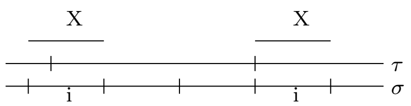
relative_in(i_X,τ,i), X $\in \sigma$		σ_in_τ(i_X)
relative_to(i_X,τ,i), X $\in \sigma$		σ_to_τ(i_X)

Figure 5.7: Illustration of CL_{catts} time constraints that define metric relations on activities.

Constraint:	Description:
$\forall \alpha.(\alpha :: \tau \vee \alpha :: \tau^*)$	
$X^\alpha \, \mathcal{R} \, Y^\alpha$	interval relations between X^α and Y^α
$d_{X^\alpha} \odot d_{Y^\alpha}$	comparisons of durations of X^α and Y^α
$\mathcal{M} \; X^\alpha d_{Y^\alpha} = Z^\alpha$	shift in time or alternation of the duration of X^α by duration of Y^α to Z^α
$\mathcal{I} \; i_{X^\sigma} \tau \odot i$	relating indices of X^σ relative to values from τ such that i_{X^σ} in $\tau \odot i$
$E \; X^\alpha = Y^\tau$	ending point of X^α
$I \; X^\tau = i_{X^\tau}$	index of X^τ
$D \; X^\alpha = d_{X^\alpha}$	duration of X^α

where

$\mathcal{R} \in$	{*equals, before, after, starts, started_by, finishes, finished_by,meets, met_by, during, contains, overlaps, overlapped_by, within, on_or_before, on_or_after*}
$\odot \in$	$\{=, <, \leq, \geq, >, \neq\}$
$\mathcal{M} \in$	{*shift_forward, shift_backward, exten_by, shorten_by*}
$\mathcal{I} \in$	{*relative_in, relative_to*}
$E \in$	{*begin_of, end_of*}
$I \in$	{*index*}
$D \in$	{*duration*}

Table 5.3: Time constraints.

on activity starting and ending times and/or durations. This is possible since the starting and ending times and the durations of activity constraints are represented by finite domain constraints (over the same calendric type) in the constraint system $typedFD$.

5.4.3 The Conversion Constraint

As already mentioned above, CaTTS' constraint solver propagates time constraints in the language CL$_{catts}$ like "after" only for activities over the same calendric type. To propagate activities of different calendric types in a CSP defined in CaTTS-CL, and, thus, in the language CL$_{catts}$ of CaTTS' constraint solver, the domains of those activities need to be related to each other such that they become comparable. For this purpose, CaTTS provides with a novel constraint, the *conversion constraint*, written $\forall \alpha, \beta. \; (\alpha :: \tau \vee \alpha :: \tau^*)$, $(\beta :: \sigma \vee \beta :: \sigma^*) \; X^\alpha \simeq Y^\beta$, and read as "for all types α, either of kind point or interval of calendric type τ and β, either of kind point or interval of calendric type σ, X (of type α) is related in the equivalence relation $\simeq$ to Y (of type β)". Thus, the conversion constraint $X^\alpha \simeq Y^\beta$ defines a certain kind of *equivalence relation* between the activities X^α and Y^β: the activities X^α and Y^β represent the same continuous, amount of time, however, expressed in terms of different calendric types.

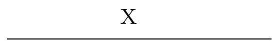
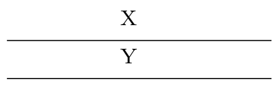
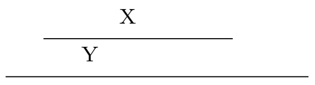
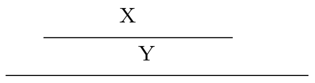
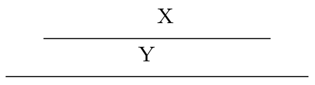
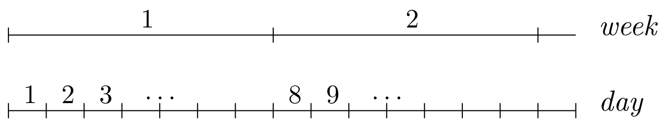
time constraints, $X, Y \in \tau$:		**definition on durations:**
duration(X,Y)		$Y = d_X$
$X = Y$		$d_X = d_Y$
$X < Y$ $Y > X := X < Y$		$d_X < d_Y$
$X \leq Y$ $Y \geq X := X \leq Y$		$d_X \leq d_Y$
$X \neq Y$		$d_X \neq d_Y$

Figure 5.8: Illustration of CL$_{catts}$ time constraints that define metric relations on durations.

For example,

$$X :: 1..8 + 7..7, day \simeq Y :: 1..2 + 1..1, week$$

according to the following illustration.



Then the time interval $[8, 15]$ of days represented by X corresponds to the time interval $[2, 2]$ of weeks represented by Y, for example.

Note that since CaTTS' constraint solver merely implements bounds consistency and not arc consistency, the conversion constraint just enforces that the bounds of the starting times (resp. the bounds of the ending times) of the activity constraint start (resp. finish) intervals in each case of the related type. That is, the days 1 and 8 start weeks (namely weeks 1 and 2), but for example day 3 in the domain of X does not start a week. Nevertheless, bounds consistency is sufficient for CaTTS' multi-calendar constraint solver since the equivalent representation (in the example in terms of weeks) which excludes any incompatible solutions remains in the constraint store during constraint propagation.

The conversion constraint relies on and refers to calendric types defined in some CaTTS-DL calendar specification. In particular, the conversion constraint applies the conversion functions generated from the type predicates (cf. Section 6.4).

5.5 The Constraint Propagation Algorithm

A *constraint propagation algorithm* is a specific algorithm that transforms a given CSP into another one such that the reduced CSP is equivalent (according to Definition 5.3, p. 127) to the original CSP. Recall that the constraints of such equivalent CSPs have the same set of solutions, but those of the transformed CSP are easier to solve. A constraint propagation algorithm can be specified by a finite set of proof rules.

In what follows, a constraint propagation algorithm to solve multi-calendar appointment scheduling problems that can be specified in CaTTS-CL is introduced. This algorithm implements the previously introduced time constraints and the conversion constraint of the language CL_{catts}. The algorithm is specified by a finite set of proof rules. Following conventions are used for the notations in the proof rules: whenever constraint propagation requires that the corresponding variables have the same type, type information for variables and corresponding domains is omitted. Whenever variables may have different types or whenever the type influences constraint propagation, type information is explicitly given for variables and corresponding domains. Side conditions that must hold for some proof rule are given in the premise of the rule following "|". A reference implementation of this algorithm in CHR (Constraint Handling rules) is given in the Appendix B.

CaTTS' constraint propagation algorithm to solve multi-calendar appointment scheduling problems tests *consistency*, *simplifies* constraints (i.e. the algorithm tries to transform a given constraint C into an equivalent, but simpler constraint C'), and, if possible, *solves* constraints. Since the consistency test for combinatorial problems such as multi-calendar appointment scheduling is NP-complete, no efficient algorithm exists. Therefore, CaTTS' constraint solver implements a constraint propagation algorithm that is based on a *local-consistency method*, in particular *bounds consistency* [FA97, MS98, Apt03]. To ensure global consistency the algorithm needs to be interleaved with a search method.[13]

5.5.1 Achieving Local Consistency

Algorithms that achieve a kind of reduction of a CSP into an equivalent, but easier to solve CSP usually aim at reaching some form of *local consistency.* Informally, local consistency means that some subparts of the considered CSP are consistent (i.e. have a solution). That means, a small, fixed-size sub-problem of the initial CSP is considered repeatedly until a fixpoint is reached. Those sub-problems are simplified, and new implied (redundant) constraints are propagated from them such that those added constraints (hopefully)

[13]Since CaTTS' constraint solver is implemented in CHR integrated into Prolog, Prolog search, i.e. backtracking is used.

cause simplification. In general, achieving local consistency consists either in reducing the domains of the considered variables or in reducing the considered constraints.

A basic method to simplify constraints over finite domains are so-called *consistency techniques* [FA97, MS98, Apt03], ensuring local consistency of a CSP. For the implementation of CaTTS' constraint propagation algorithm on interval domains, we apply the consistency technique of *bounds consistency* [FA97, MS98]. Bounds consistency is a weaker, but analogous, form of *arc consistency* [FA97, MS98, Apt03] which is adapted such that only interval boundaries are considered instead of the enumerated lists. Arc consistency originates from Montanari [Mon74]. Bounds consistency (also called interval consistency) has been introduced by various authors, e.g. [vHSD92]. Arc and bounds consistency are algorithms that ensure local consistency. Classical consistency algorithms have been first explored for constraint networks in Artificial Intelligence research in the late 1960ies.

Bounds consistency is defined as follows [FA97, MS98, Apt03].

Definition 5.5 (Bounds Consistency). *A constraint $C \subseteq D_1 \times \ldots \times D_n$ on the variables $X_1, \ldots, X_n$ with respective domains $D_1, \ldots, D_n$ is* **bounds consistent***, if for all $i \in \{1, \ldots, n\}$ and for all bounds $v_i \in D_i$ ($v_i \in \{n_i, m_i\}$) the constraint $\exists(X_1 \in D_1 \wedge \ldots \wedge X_i = v_i \wedge \ldots \wedge X_n \in D_n \wedge C(X_1, \ldots, X_n))$ is satisfiable.*

A CSP is **bounds consistent** *if all its constraints are bounds consistent.*

Intuitively, a constraint C is bounds consistent if for every involved interval domain over the same type there exists a solution $(d_1, \ldots, d_i, \ldots, d_n)$ to C such that $n_i \leq d_i \leq m_i$. An algorithm implementing bounds consistency makes constraints bounds consistent by tightening their interval domains. A conjunction of constraints, in particular, a CSP is made bounds consistent by making the participating constraints bounds consistent. The algorithm implementing bounds consistency for CaTTS' constraint solver is based on the logical formulation of Definition 5.5:

If $X_1 \in D_1 \wedge \ldots \wedge X_i = v_i \wedge \ldots \wedge X_n \in D_n \wedge C(X_1, \ldots, X_n) \rightarrow X_i = v'_i$, then $v_i \cap v'_i$ is a new interval bound of X_i, $v_i \in \{n_i, m_i\}$.

That means, whenever a constraint C has been propagated during constraint solving, the intersection of the interval bound v'_i and the old interval bound v_i of the variable X_i in C is computed. The result of applying this *bounds consistency rule* then yields in the new (reduced domain) of X_i, i.e. $v_i \cap v'_i$ which replaces the interval bound v_i.

Note: v'_i is defined by the proof rule of the constraint C.

The notion of bounds consistency (and its implementation in terms of the bounds consistency rule) formulates the terminates criterion for CaTTS' constraint propagation algorithm, i.e. the algorithm terminates (and returns an equivalent but smaller CSP) if the CSP is bounds consistent and fails otherwise.

Lemma 5.1 *A CSP is bounds consistent iff it is closed under the application of the "bounds consistency rule".*

Proof 5.1 *Since the algorithm achieving bounds consistency is based on the logical formulation of Definition 5.5, it is sufficient to note that a constraint C on the variables $X_1,\ldots,X_n$ with respective interval bounds $v_1,\ldots,\ v_n$ is bounds consistent iff for every $i \in \{1,\ldots,n\}$ and for every $v_i \in \{n_i, m_i\}$, $v_i \cap v'_i = v_i$, i.e. iff $v_i = v'_i$ where v'_i is defined as in the proof rule of C.*

5.5.2 Proof Rules for Time Constraints

In Section 5.4.2, we have introduced the set of time constraints (over the same calendric types) supported in CaTTS, i.e. CL$_{catts}$ time constraints. The (informal) definitions of the time constraints on starting and ending times and/or durations are illustrated in Figure 5.6 p.140, Figure 5.7, p.141, and Figure 5.8, p.143. In the following, at least one proof rule is specified for each of those time constraints, formalizing the previously given illustrations. Those proof rules allow for manipulating time constraints on activities during constraint propagation. In particular, those proof rules either reduce the domains of the associated activities or transform the constraints in the constraint store. The proof rules for CL$_{catts}$ time constraints are given subsequent to this section in Tables, 5.6, 5.4, 5.5, 5.8, and 5.7. The "bounds consistency algorithm" (cf. Section 5.5.1) is directly incorporated into each of the subsequently introduced proof rules for time constraints.

Since the proof rules of CaTTS' time constraints are all defined analogously according to the illustrations given in Figure 5.6, Figure 5.7, and Figure 5.8, the rules for one of these constraints are discussed in detail. Any further rule should be then explanatory by itself.

So, let us turn attention to the time constraint *before*. The constraint X *before* Y (where X and Y have the same type) says that any (time) interval that can be represented by X must end before any (time) interval that can be represented by Y starts, i.e. $S_X \leq E_X < S_Y \leq E_Y$ where $X :: S_X + d_X, \tau$ and $Y :: S_Y + d_Y, \tau$ and $E_X := S_X + d_X - 1$ and $E_Y := S_Y + d_Y - 1$. Since the conditions $S_X \leq E_X$ and $S_Y \leq E_Y$, i.e. that the starting time of a (time) interval must less or equal than its ending time is formulated with the activity domain constraint itself, the condition $E_X < S_Y$ formulates the criterion under which the constraint X *before* Y is bounds consistent. For example, assume the following CSP:

$$\langle X\ \mathit{before}\, Y; X :: 20..25 + 1..1, \tau,\ Y :: 18..23 + 2..4, \tau\rangle$$

I.e. the constraint $X\, before\, Y$ is in the constraint store with associated activity constraints $X :: 20..25 + 1..1, \tau$ and $Y :: 18..23 + 2..4, \tau$. X represents the time intervals $[20, 20], [21, 21], \ldots, [25, 25]$ and Y represents the time intervals $[18, 19], [18, 20], [18, 21], \ldots,$ $[23, 24], [23, 25], [23, 26]$. Applying the domain reduction rule for the time constraint *before* on this CSP:

$$\frac{\langle X\ \mathit{before}\, Y; X :: n_X..m_X + d_X^-..d_X^+, Y :: n_Y..m_Y + d_Y^-..d_Y^+\rangle}{\begin{array}{c}\langle X\ \mathit{before}\, Y; \\ X :: n_X..min(m_X, (min(m_{E_X}, m_Y - 1) - d_X^- + 1)) + d_X^-..min(d_X^+, (min(m_{E_X}, m_Y - 1) - n_X + 1)) \\ Y :: max(n_Y, n_{E_X} + 1)..m_Y + d_Y^-..d_Y^+\rangle\end{array}}$$

$$\frac{\langle X \ \mathit{equals}\, Y; X :: D_X, Y :: D_Y\rangle}{\langle X \ \mathit{equals}\, Y;\ X :: D_X \cap D_Y, Y :: D_X \cap D_Y\rangle}$$

$$\frac{\langle X \ \mathit{before}\, Y; X :: n_X..m_X + d_X^-..d_X^+, Y :: n_Y..m_Y + d_Y^-..d_Y^+\rangle}{\begin{array}{c}\langle X \ \mathit{before}\, Y; \\ X :: n_X..\mathit{min}(m_X, (\mathit{min}(m_{E_X}, m_Y - 1) - d_X^- + 1)) + d_X^-..\mathit{min}(d_X^+, (\mathit{min}(m_{E_X}, m_Y - 1) - n_X + 1)) \\ Y :: \mathit{max}(n_Y, n_{E_X} + 1)..m_Y + d_Y^-..d_Y^+\rangle\end{array}}$$

$$\frac{\langle X \ \mathit{starts}\, Y; X :: S_X + d_X, Y :: S_Y + d_Y\rangle}{\langle X \ \mathit{starts}\, Y;\ X :: S_X \cap S_Y + d_X^-..d_X^+,\ Y :: S_X \cap S_Y + d_Y^-..d_Y^+\rangle}$$

$$\frac{\langle X \ \mathit{starts}\, Y; X :: n_X..m_X + d_X^-..d_X^+, Y :: n_Y..m_Y + d_Y^-..d_Y^+\rangle}{\begin{array}{c}\langle X \ \mathit{starts}\, Y; \\ X :: n_X..\mathit{min}(m_X, (\mathit{min}(m_{E_X}, m_{E_Y} - 1) - d_X^- + 1)) + d_X^-..\mathit{min}(d_X^+, (\mathit{min}(m_{E_X}, m_{E_Y} - 1) - n_X + 1)), \\ Y :: \mathit{max}(n_Y, (\mathit{max}(n_{E_Y}, n_{E_X} + 1) - d_Y^+ + 1))..m_Y + d_Y^-..d_Y^+\rangle\end{array}}$$

$$\frac{\langle X \ \mathit{finishes}\, Y; X :: S_X + d_X, Y :: S_Y + d_Y\rangle}{\begin{array}{c}\langle X \ \mathit{finishes}\, Y;\ X :: (((S_X + d_X - 1) \cap (S_Y + d_Y - 1)) - d_X + 1) + d_X, \\ Y :: (((S_X + d_X - 1) \cap (S_Y + d_Y - 1)) - d_Y + 1) + d_Y\rangle\end{array}}$$

$$\frac{\langle X \ \mathit{finishes}\, Y; X :: n_X..m_X + d_X^-..d_X^+, Y :: n_Y..m_Y + d_Y^-..d_Y^+\rangle}{\langle X \ \mathit{finishes}\, Y;\ X :: n_X..\mathit{min}(m_X, m_Y - 1) + d_X^-..d_X^+,\ Y :: \mathit{max}(n_Y, n_X + 1)..m_Y + d_Y^-..d_Y^+\rangle}$$

$$\frac{\langle X \ \mathit{during}\, Y; X :: n_X..m_X + d_X^-..d_X^+, Y :: n_Y..m_Y + d_Y^-..d_Y^+\rangle}{\langle X \ \mathit{during}\, Y;\ X :: \mathit{max}(n_X, n_Y + 1)..m_X + d_X^-..d_X^+,\ Y :: n_Y..\mathit{min}(m_Y, m_X - 1) + d_Y^-..d_Y^+\rangle}$$

$$\frac{\langle X \ \mathit{during}\, Y; X :: n_X..m_X + d_X^-..d_X^+, Y :: n_Y..m_Y + d_Y^-..d_Y^+\rangle}{\begin{array}{c}\langle X \ \mathit{during}\, Y; \\ X :: n_X..\mathit{min}(m_X, (\mathit{min}(m_{E_X}, m_{E_Y} - 1) - d_X^- + 1)) + d_X^-..\mathit{min}(d_X^+, (\mathit{min}(m_{E_X}, m_{E_Y} - 1) - n_X + 1)), \\ Y :: \mathit{max}(n_Y, (\mathit{max}(n_{E_Y}, n_{E_X} + 1) - d_Y^+ + 1))..m_Y + \mathit{max}(d_Y^-, (\mathit{max}(n_{E_Y}, n_{E_X} + 1) - m_Y + 1))..d_Y^+\rangle\end{array}}$$

$$\frac{\langle X \ \mathit{meets}\, Y; X :: S_X + d_X, Y :: S_Y + d_Y\rangle}{\langle X \ \mathit{meets}\, Y;\ X :: ((E_X \cap S_Y) - d_X + 1) + d_X,\ Y :: (E_X \cap S_Y) + d_Y\rangle}$$

$$\frac{\langle X \ \mathit{overlaps}\, Y; X :: n_X..m_X + d_X^-..d_X^+, Y :: n_Y..m_Y + d_Y^-..d_Y^+\rangle}{\langle X \ \mathit{overlaps}\, Y;\ X :: \mathit{max}(n_X, n_Y + 1)..m_X + d_X^-..d_X^+,\ Y :: n_Y..\mathit{min}(m_Y, m_X - 1) + d_Y^-..d_Y^+\rangle}$$

$$\frac{\langle X \ \mathit{overlaps}\, Y; X :: n_X..m_X + d_X^-..d_X^+, Y :: n_Y..m_Y + d_Y^-..d_Y^+\rangle}{\begin{array}{c}\langle X \ \mathit{overlaps}\, Y; \\ X :: \mathit{max}(n_X, (\mathit{max}(n_{E_X}, n_Y + 1) - d_X^+ + 1))..m_X + \mathit{max}(d_X^-, (\mathit{max}(n_{E_X}, n_Y + 1) - m_X + 1))..d_X^+ \\ Y :: n_Y..\mathit{min}(m_Y, m_{E_X} - 1) + d_Y^-..d_Y^+\rangle\end{array}}$$

$$\frac{\langle X \ \mathit{overlaps}\, Y; X :: n_X..m_X + d_X^-..d_X^+, Y :: n_Y..m_Y + d_Y^-..d_Y^+\rangle}{\begin{array}{c}\langle X \ \mathit{overlaps}\, Y; \\ X :: n_X..\mathit{min}(m_X, (\mathit{min}(m_{E_X}, m_{E_Y} - 1) - d_X^- + 1)) + d_X^-..\mathit{min}(d_X^+, (\mathit{min}(m_{E_X}, m_{E_Y} - 1) - n_X + 1)) \\ Y :: \mathit{max}(n_Y, (\mathit{max}(n_{E_Y}, n_{E_X} + 1) - d_Y^+ + 1))..m_Y + \mathit{max}(d_Y^-, (\mathit{max}(n_{E_Y}, n_{E_X} + 1) - m_Y + 1))..d_Y^+\rangle\end{array}}$$

Table 5.4: The domain reduction rules for time constraints on activities according to the illustrations in Figure 5.6, p.140.

$$\frac{\langle \mathit{shift\_forward(X,d,Y)}; X :: n_X..m_X + d_X^-..d_X^+, d :: d^-..d^+ \rangle}{\langle \mathit{shift\_forward(X,d,Y)};\ Y :: n_X + d^-..m_X + d^+ + d_X^-..d_X^+ \rangle}$$

$$\frac{\langle \mathit{shift\_forward(X,d,Y)}; d :: d^-..d^+, Y :: n_Y..m_Y + d_Y^-..d_Y^+ \rangle}{\langle \mathit{shift\_forward(X,d,Y)};\ X :: n_Y - d^-..m_Y - d^+ + d_X^-..d_X^+ \rangle}$$

$$\frac{\langle \mathit{relative\_to}(i_{X^\sigma}, \tau, i); i_X :: D_{X^\sigma}}{\langle \mathit{relative\_to}(i_{X^\sigma}, \tau, i);\ i_{X^\sigma} :: \sigma\_to\_\tau(i, D_{X^\sigma}) \rangle}$$

$$\frac{\langle \mathit{relative\_in}(i_{X^\sigma}, \tau, i); i_X :: D_{X^\sigma} \rangle}{\langle \mathit{relative\_in}(i_{X^\sigma}, \tau, i);\ i_{X^\sigma} :: \sigma\_in\_\tau(i, D_{X^\sigma}) \rangle}$$

$$\frac{\langle \mathit{extend\_by(X,d,Y)}; X :: n_X..m_X + d_X^-..d_X^+, d :: d^-..d^+ \rangle}{\langle \mathit{extend\_by(X,d,Y)};\ Y :: n_X..m_X + d_X^- + d^-..d_X^+ + d^+ \rangle}$$

$$\frac{\langle \mathit{extend\_by(X,d,Y)}; d :: d^-..d^+, Y :: n_Y..m_Y + d_Y^-..d_Y^+ \rangle}{\langle \mathit{extend\_by(X,d,Y)};\ X :: n_Y..m_Y + d_Y^- - d^-..d_Y^+ - d^+ \rangle}$$

$$\frac{\langle \mathit{end(X,Y)}; X :: S_X + d_X \rangle}{\langle \mathit{end(X,Y)}; Y :: E_X \rangle}$$

$$\frac{\langle \mathit{begin(X,Y)}; X :: S_X + d_X \rangle}{\langle \mathit{begin(X,Y)}; Y :: S_X \rangle}$$

$$\frac{\langle \mathit{duration(X,Y)}; X :: S_X + d_X \rangle}{\langle \mathit{duration(X,Y)}; Y :: D_X \rangle}$$

$$\frac{\langle \mathit{index(X,Y)}; X :: S_X \rangle}{\langle \mathit{index(X,Y)}; Y :: i_{S_X} \rangle}$$

Table 5.5: The domain reduction rules for time constraints on activities according to the illustrations in Figure 5.7 p.141.

$$\frac{\langle X \ \textit{equals}\ X; X :: D_X\rangle}{\langle ; X :: D_X\rangle}$$

$$\frac{\langle X\ \mathcal{I}\ X; X :: D_X\rangle}{\langle ; X :: \emptyset\rangle}$$

$$\frac{\langle X \ \textit{during}\, Y; X :: S_X + d_X, Y :: S_Y + d_Y\rangle \mid d_X \geq d_Y}{\langle ; \bot\rangle}$$

$$\frac{\langle X \ \textit{overlaps}\, Y; X :: S_X + d_X, Y :: S_Y + d_Y\rangle \mid d_X = 1}{\langle ; \bot\rangle}$$

$$\frac{\langle X \ \textit{overlaps}\, Y; X :: S_X + d_X, Y :: S_Y + d_Y\rangle \mid d_Y = 1}{\langle ; \bot\rangle}$$

$$\frac{\langle X \ \textit{overlaps}\, Y; X :: S_X + d_X, Y :: S_Y + d_Y\rangle \mid d_X = d_Y}{\langle ; \bot\rangle}$$

$$\frac{\langle X \ \textit{equals}\, Y; X :: S_X + d_X, Y :: S_Y + d_Y\rangle \mid d_X \neq d_Y}{\langle ; \bot\rangle}$$

where $\mathcal{I} \in \{\textit{before, starts, finishes, during, overlaps, meets}\}$

Table 5.6: The transformation rules for time constraints on activities.

$$\frac{\langle X = Y; X :: S_X + d_X, Y :: S_Y + d_Y\rangle}{\langle X = Y;\ X :: S_X + d_X \cap d_Y,\ Y :: S_Y + d_X \cap d_Y\rangle}$$

$$\frac{\langle X \leq Y; X :: S_X + d_X^-..d_X^+, Y :: S_Y + d_Y^-..d_Y^+\rangle}{\langle X \leq Y;\ X :: S_X + d_X^-..min(d_X^+, d_Y^+),\ Y :: S_Y + max(d_Y^-, d_X^-)..d_Y^+\rangle}$$

$$\frac{\langle X < Y; X :: S_X + d_X^-..d_X^+, Y :: S_Y + d_Y^-..d_Y^+\rangle}{\langle X < Y;\ X :: S_X + d_X^-..min(d_X^+, d_Y^+ - 1),\ Y :: S_Y + max(d_Y^-, d_X^- + 1)..d_Y^+\rangle}$$

$$\frac{\langle X \neq Y; X :: S_X + d_X^-..d_X^+, Y :: S_Y + d_Y^-..d_Y^+\rangle \mid d_X^- = d_Y^-, d_Y^- = d_Y^+}{\langle X \neq Y;\ X :: S_X + d_X^- + 1..d_X^+,\ Y :: S_Y + d_Y^-..d_Y^+\rangle}$$

Table 5.7: The domain reduction rules for time constraints on durations according to the illustrations in Figure 5.8, p.143.

$$\frac{\langle X \geq Y; X :: D_X, Y :: D_Y +\rangle}{\langle Y \leq X;\ X :: D_X,\ Y :: D_Y\rangle} \qquad \frac{\langle X > Y; X :: D_X, Y :: D_Y\rangle}{\langle Y < X;\ X :: D_X,\ Y :: D_Y\rangle}$$

$$\frac{\langle X \neq Y; X :: D_X, Y :: D_Y\rangle \mid d_X \cap d_Y = \emptyset}{\langle ;\ X :: D_X,\ Y :: D_Y\rangle}$$

Table 5.8: The transformation rules for time constraints on durations.

The application of *before* on those time intervals represented by X and Y removes those values of the domains of X and Y which do not satisfy the condition that the ending time of the interval represented by X is smaller than the starting time of the interval represented by Y. Let's start with the (time) intervals represented by X: none of the intervals $[23, 23]$, $[24, 24]$, and $[25, 25]$ satisfies the condition of the constraint *before*, i.e. their ending times are larger than the starting times of those intervals with the largest possible starting time of Y. That is, the intervals $[20, 20]$, $[21, 21]$, and $[22, 22]$ remain in the domain of X which is now represented by $X :: 20..22 + 1..1, \tau$. Remain the (time) intervals represented by Y: none of the intervals $[18, 19]$, $[18, 20]$, $[18, 21]$, $\ldots$, $[20, 22]$ satisfy the condition of the constraint *before*, i.e. their starting times are smaller than the ending times of intervals represented by X. That is, the intervals $[21, 22]$, $\ldots [23, 24]$, $[23, 25]$, and $[23, 26]$ remain in the domain of Y which is now represented by $Y :: 21..23 + 2..4, \tau$.

The proof rule makes this a little bit more elegant. It merely considers the maximal starting time of X and the minimal ending time of Y according to the condition $S_X < E_Y$ that must hold for the constraint `X before Y`. Thus, formally, the domain reduction rule for the time constraint *before* is read as follows: if the constraint X *before* Y is in the constraint store of a CSP with associated activity constraints $X :: n_X..m_X + d_X^-..d_X^+$ and $Y :: n_Y..m_Y + d_Y^-..d_Y^+$, then new domain expressions for the activities X and Y (over the same calendric type) are propagated as follows according to the condition $E_X < S_Y$, i.e. the ending time of X must be before, i.e. less than the starting time of Y. The condition $E_X < S_Y$ implies that the maximal ending time of X (i.e. m_{E_X}) must be less than the minimal starting time of Y (i.e. m_Y). Therefore, the minimum of m_{E_X} and $m_Y - 1$ is computed (denoted $min(m_{E_X}, m_Y - 1)$). Since the activity X is represented by its starting time and duration, a new maximal starting time for X must be computed from the value $min(m_{E_X}, m_Y - 1)$. This is dome by subtracting the minimal duration of X plus 1 from this value (denoted $min(m_{E_X}, m_Y - 1) - d_X^- + 1$) and taking the minimal value from X' old maximal starting time m_X and this value (denoted $min(m_X, (min(m_{E_X}, m_Y - 1) - d_X^- + 1)))$. The same is done to compute the new maximal duration of X. The condition $E_X < S_Y$ further implies that the minimal starting time of Y (i.e. n_Y) must be larger than the minimal ending time of X (i.e. n_{E_X}). Therefore, the maximum of these two values is added as a new minimal starting time for Y (denoted $max(n_Y, n_{E_X} + 1)$).

The proof rule for the time constraint X *before* Y thus ensures that the bounds that represent the interval domain of the ending times of X are always smaller than the bounds that represent the interval domain of the starting times of Y. The conditions that take the minimum of the old and new upper bounds and the maximum of the old and new lower bounds define the bounds consistency rule: the bounds consistency rule is applied and the domains of X and Y are replaced by the new, reduced domains.

The application of this rule (that incorporates the application of the bounds consistency rule) is illustrated by the following example which applies the rule for *before* in two steps, first to X and then to Y (where $X :: 20..25 + 1..1, \tau$ and $Y :: 18..23 + 2..4, \tau$ and, thus, $E_X = 20..25$ and $E_Y = 19..26$):

$$X\ \textit{before}\ Y;\ X :: 20..25 + 1..1, \tau,\ Y :: 18..23 + 2..4, \tau$$

$$\begin{aligned}
\mapsto_{before\_x}\quad & X\ \textit{before}\ Y; \\
& X :: 20..min(25, (min(25, 23-1) - 1 + 1) + 1..min(1, (min(25, 23-1) - 20 + 1)), \tau, \\
& Y :: 18..23 + 2..4, \tau \\
& X\ \textit{before}\ Y;\ X :: 20..min(25, (min(25, 22) - 1 + 1) + 1..min(1, min(25, 22) - 20 + 1), \tau, \\
& Y :: 18..23 + 2..4, \tau \\
& X\ \textit{before}\ Y;\ X :: 20..min(25, 22) + 1..min(1, 3), \tau,\ Y :: 18..23 + 2..4, \tau \\
& X\ \textit{before}\ Y;\ X :: 20..22 + 1..1, \tau,\ Y :: 18..23 + 2..4, \tau
\end{aligned}$$

$$\begin{aligned}
\mapsto_{before\_y}\quad & X\ \textit{before}\ Y;\ X :: 20..22 + 1..1, \tau,\ Y :: max(18, 20+1)..23 + 2..4, \tau \\
& X\ \textit{before}\ Y;\ X :: 20..22 + 1..1, \tau,\ Y :: max(18, 21)..23 + 2..4, \tau \\
& X\ \textit{before}\ Y;\ X :: 20..22 + 1..1, \tau,\ Y :: 21..23 + 2..4, \tau
\end{aligned}$$

In addition to this domain reduction rule, the following transformation rule is defined for the time constraint *before*.

$$\frac{\langle X\ \textit{before}\ X; X :: D_X\rangle}{\langle ; X :: \emptyset\rangle}$$

This rule is read as follows: if the constraint X *before* X is in the constraint store of a CSP and the variable X represents a domain D_X, denoted $X :: D_X$[14], then the constraint X *before* X is deleted from the constraint store and X represents an empty domain, denoted $X :: \emptyset$. Thus, an application of this rule would, in particular, terminate constraint propagation with the result, that the considered CSP is inconsistent, i.e. the algorithm fails. For example, $X :: 2..10 + 3..3, \textit{day}$, i.e. a tasks of 3 days starting earliest at day indexed by 2 and latest at day indexed by 10, and the time constraint X *before* X is in the constraint store, then the interval domain constraint $X :: \emptyset$ is propagated since there does not exist an interval domain for which the constraint X *before* X is bounds consistent according to Definition 5.5. In particular, the activity X activity cannot be before itself.

5.5.3 The Proof Rule for the Conversion Constraint

CaTTS' constraint propagation algorithm provides with a specific proof rule for the *conversion constraint* $\forall\alpha,\beta.(\alpha :: \tau \vee \alpha :: \tau^*), (\beta :: \sigma \vee \beta :: \sigma^*)$, $X^\alpha \simeq Y^\beta$ with $\alpha \leq \beta$ (or $\beta \leq \alpha$) according to the subtype relation (cf. Definition 3.7) over time granularities (calendric types defined in a CaTTS-DL calendar specification). The conversion constraint relies on and refers to the calendric type definitions in a CaTTS-DL calendar specification. In particular, for each pair of calendric types α and β where $\alpha \leq \beta$ in a CaTTS-DL calendar specification, a *conversion function* is generated from the (user-defined) type predicates by CaTTS-DL's language processor. Those functions are applied during constraint solving whenever a conversion constraint is propagated on activities of types α and β. Conversion function generation from CaTTS-DL calendric type predicates is formally introduced

[14]Recall that a CSP is represented by a pair $\langle\mathcal{C}; \mathcal{D}_\mathcal{E}\rangle$, where $\mathcal{C}$ is a set of constraint, the constraint store, and $\mathcal{D}_\mathcal{E}$ a set of associated domain expressions.

$\forall \alpha, \beta.(\alpha :: \tau \vee \alpha :: \tau^*), (\beta :: \sigma \vee \beta :: \sigma^*)$

$$\frac{\langle X^\alpha \simeq Y^\beta; X^\alpha :: n_X..m_X + d_X^-..d_X^+, Y^\beta :: n_Y..m_Y + d_Y^-..d_Y^+\rangle \mid \alpha \leq \beta}{\begin{array}{c}\langle X^\alpha \simeq Y^\beta; \\ X^\alpha :: \mathit{max}(n_X, c^+_{\beta\to\alpha}(n_Y))..\mathit{min}(m_X, c_{\beta\to\alpha}(m_Y)) + \\ \mathit{max}(d_X^-, (c_{\beta\to\alpha}(n_{E_Y}) - c_{\beta\to\alpha}(m_Y) + 1))..\mathit{min}(d_X^+, (c^-_{\beta\to\alpha}(m_{E_Y}) - c^+_{\beta\to\alpha}(n_Y) + 1)), \\ Y^\beta :: \mathit{max}(n_Y, c^-_{\alpha\to\beta}(n_X))..\mathit{min}(m_Y, c^-_{\alpha\to\beta}(m_X)) + \\ \mathit{max}(d_Y^-, (c^+_{\alpha\to\beta}(n_{E_X}) - c^-_{\alpha\to\beta}(m_X) + 1))..\mathit{min}(d_Y^+, (c^+_{\alpha\to\beta}(m_{E_X}) - c^-_{\alpha\to\beta}(n_X) + 1))\rangle\end{array}}$$

Table 5.9: The domain reduction rule for the conversion constraint.

and discussed in Section 6.4. For $\alpha \leq \beta$, the conversion function is written $c_{\alpha\to\beta}$, $c^-_{\alpha\to\beta}$ for predecessors, and $c^+_{\alpha\to\beta}$ for successors, and $c_{\beta\to\alpha}$ denotes its inverse ($c^-_{\beta\to\alpha}$ inverse for predecessors, and $c^+_{\beta\to\alpha}$ inverse for successors).

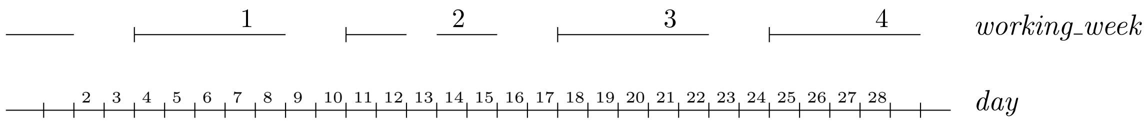
Let us now consider the proof rule for the conversion constraint $X^\alpha \simeq Y^\beta$. This rule applies the conversion functions form α to β, where $\alpha \leq \beta$, for starting times and ending times and its inverse functions. The rule for CaTTS' conversion constraint is given in Table 5.9.

This rule is read as follows: if the constraint $X^\alpha \simeq Y^\beta$ is in the constraint store with corresponding domain expressions $X^\alpha :: n_{X^\alpha}..m_{X^\alpha} + d^-_{X^\alpha}..d^+_{X^\alpha}$ and $Y^\beta :: n_{Y^\beta}..m_{Y^\beta} + d^-_{Y^\beta}..d^+_{Y^\beta}$ where $\alpha \leq \beta$, then new domain expressions are added for X and Y. The new domains are such that the possible time intervals of X in type α correspond to the possible time intervals of Y in type β. This is achieved by adapting the bounds of the starting times of X and Y such that the bounds of X (which refer to type α) are started by the bounds of Y (which refer to type β) and by adapting the bounds of the durations of X and Y such that the bounds of X (which refer to durations of type α) correspond to the bounds of Y (which refer to durations of type β). The proof rule for the conversion constraint thus ensures that the bounds of the interval domains that represent the starting times of X in type α are always equivalent to the bounds of the interval domains that represent the starting times of Y in type β. Furthermore, this proof rule ensures that the bounds of the intervals that represent the durations of X in type α always correspond to the bounds of the intervals domains that represent the durations of Y. For example, if X represents weeks of duration 1 with starting times 1 to 2 and Y represents corresponding intervals of days, then the duration of Y must be 7 (i.e. the duration of 1 week in terms of days is 7) and the minimal bound of the interval domain that represents its starting times must start the week 1 and the maximal bound of the interval domain that represents its starting times must start the week 2. If the day with index 1 starts the week 1, then the minimal bound of the interval domain that represents its starting times of Y is 1 and the maximal bound is 8.

The application of the conversion constraint is illustrated by the following example.

$$\langle X \simeq Y;\ X :: 2..4 + 1..1, \mathit{working\_week},\ Y :: 9..19 + 7..7, \mathit{day}\rangle$$

where $E_X = 2..4, \mathit{working\_week}$ and $E_Y = 15..25, \mathit{day}$ and the following indexing of the values of types `working_week` and `day` is assumed.



The rule for the conversion constraint is now applied stepwise, first on X and subsequently on Y.

$$X \simeq Y;\ X :: 2..4 + 1..1, \mathit{working\_week},\ Y :: 9..19 + 7..7, \mathit{day}$$

$$\begin{aligned}
\mapsto_{\simeq\_x}\quad & X :: max(2, c^{+}_{day\to working\_week}(9))..min(4, c_{day\to working\_week}(19)) + \\
& max(1, (c_{day\to working\_week}(15) - c_{day\to working\_week}(19) + 1)).. \\
& min(1, (c^{-}_{day\to working\_week}(25) - c^{+}_{day\to working\_week}(9) + 1)), \mathit{working\_week} \\
& X :: max(2,2)..min(4,3) + max(1,(2-3+1))..min(1,(3-2+1)), \mathit{working\_week} \\
& X :: 2..3 + 1..1, \mathit{working\_week}
\end{aligned}$$

$$\begin{aligned}
\mapsto_{\simeq\_y}\quad & X \simeq Y;\ X :: 2..3 + 1..1, \mathit{working\_week},\ Y :: 9..19 + 7..7, \mathit{day} \\
& Y :: max(9, c^{-}_{working\_week\to day}(2))..min(19, c^{-}_{working\_week\to day}(3)) + \\
& max(7, (c^{+}_{working\_week\to day}(2) - c^{-}_{working\_week\to day}(3) + 1)).. \\
& min(7, (c^{+}_{working\_week\to day}(3) - c^{-}_{working\_week\to day}(2) + 1)), \mathit{day} \\
& Y :: max(9,11)..min(19,18) + max(7, 17-18+1)..min(7, 24-11+1), \mathit{day} \\
& Y :: 11..18 + 7..7, \mathit{day}
\end{aligned}$$

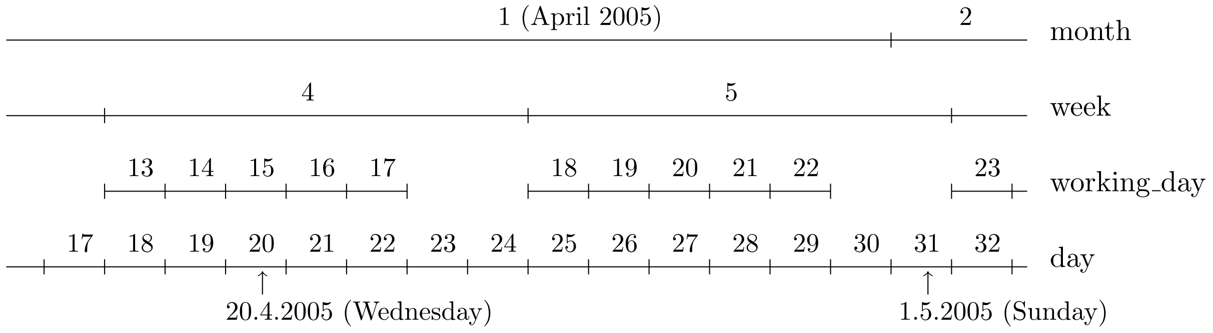
$$X \simeq Y;\ X :: 2..3 + 1..1, \mathit{working\_week},\ Y :: 11..18 + 7..7, \mathit{day}$$

5.5.4 Example: Application of Proof Rules

Let us turn attention back to the Example 5.1 discussed in Section 5.2:

A person plans a meeting lasting 3 working days after 20th April 2005 and before May 2005. A colleague's visit of 1 week must overlap with the planned meeting.

where the following internal indexing of the values of the involved calendric types is assumed.



Recall that this problem can be expressed in CaTTS-CL as follows:

```
Meeting is 3 working_day &&
Meeting after "20.04.2005" && Meeting before "05.2005" &&
Visit is 1 week && Visit overlaps Meeting
```

Transformed into CL$_{catts}$, the problem is expressed as follows:

$$\begin{array}{l}\langle M' \mathit{after}\ A, M \simeq M', M' \mathit{before}\ B', B \simeq B', V' \mathit{overlaps}\ M', V \simeq V';\\ M :: 1..\infty + 3..3, \mathit{w\_day},\ M' :: 1..\infty + 3..5, \mathit{day},\ A :: 20..20, \mathit{day},\ B :: 2..2, \mathit{month},\\ B' :: 31..31 + 31..31, \mathit{day},\ V :: 1..\infty, \mathit{week},\ V'1..\infty + 7..7, \mathit{day}\rangle\end{array}$$

where M denotes `Meeting`, A denotes `"20.04.2005"`, B denotes `"05.2005"`, and V denotes `Visit`.

The application of the previously introduced proof rules is now demonstrated on this example:

$$\begin{array}{ll}
 & \langle M' \mathit{after}\ A, M \simeq M', M' \mathit{before}\ B', B \simeq B', V' \mathit{overlaps}\ M', V \simeq V';\\
 & M :: 1..\infty + 3..3, \mathit{w\_day},\ M' :: 1..\infty + 3..5, \mathit{day},\ A :: 20..20, \mathit{day},\ B :: 2..2, \mathit{month},\\
 & B' :: 31..31 + 31..31, \mathit{day},\ V :: 1..\infty, \mathit{week},\ V'1..\infty + 7..7, \mathit{day}\ \rangle\\
\mapsto_{\mathit{after}} & \langle M' \mathit{after}\ A, M \simeq M', M' \mathit{before}\ B', B \simeq B', V' \mathit{overlaps}\ M', V \simeq V';\\
 & M :: 1..\infty + 3..3, \mathit{w\_day},\ M' :: 21..\infty + 3..5, \mathit{day},\ A :: 20..20, \mathit{day},\ B :: 2..2, \mathit{month},\\
 & B' :: 31..31 + 31..31, \mathit{day},\ V :: 1..\infty, \mathit{week},\ V'1..\infty + 7..7, \mathit{day}\ \rangle\\
\mapsto_{M \simeq M'} & \langle M' \mathit{after}\ A, M \simeq M', M' \mathit{before}\ B', B \simeq B', V' \mathit{overlaps}\ M', V \simeq V';\\
 & M :: 16..\infty + 3..3, \mathit{w\_day},\ M' :: 21..\infty + 3..5, \mathit{day},\ A :: 20..20, \mathit{day},\ B :: 2..2, \mathit{month},\\
 & B' :: 31..31 + 31..31, \mathit{day},\ V :: 1..\infty, \mathit{week},\ V'1..\infty + 7..7, \mathit{day}\ \rangle\\
\mapsto_{\mathit{before}} & \langle M' \mathit{after}\ A, M \simeq M', M' \mathit{before}\ B', B \simeq B', V' \mathit{overlaps}\ M', V \simeq V';\\
 & M :: 16..\infty + 3..3, \mathit{w\_day},\ M' :: 21..27 + 3..5, \mathit{day},\ A :: 20..20, \mathit{day},\ B :: 2..2, \mathit{month},\\
 & B' :: 31..31 + 31..31, \mathit{day},\ V :: 1..\infty, \mathit{week},\ V'1..\infty + 7..7, \mathit{day}\ \rangle\\
\mapsto_{M \simeq M'} & \langle M' \mathit{after}\ A, M \simeq M', M' \mathit{before}\ B', B \simeq B', V' \mathit{overlaps}\ M', V \simeq V';\\
 & M :: 16..20 + 3..3, \mathit{w\_day},\ M' :: 21..27 + 3..5, \mathit{day},\ A :: 20..20, \mathit{day},\ B :: 2..2, \mathit{month},\\
 & B' :: 31..31 + 31..31, \mathit{day},\ V :: 1..\infty, \mathit{week},\ V'1..\infty + 7..7, \mathit{day}\ \rangle\\
\mapsto_{\mathit{overlaps}} & \langle M' \mathit{after}\ A, M \simeq M', M' \mathit{before}\ B', B \simeq B', V' \mathit{overlaps}\ M', V \simeq V';\\
 & M :: 16..20 + 3..3, \mathit{w\_day},\ M' :: 21..22 + 3..5, \mathit{day},\ A :: 20..20, \mathit{day},\ B :: 2..2, \mathit{month},\\
 & B' :: 31..31 + 31..31, \mathit{day},\ V :: 1..\infty, \mathit{week},\ V'18..18 + 7..7, \mathit{day}\ \rangle\\
\mapsto_{V \simeq V'} & \langle M' \mathit{after}\ A, M \simeq M', M' \mathit{before}\ B', B \simeq B', V' \mathit{overlaps}\ M', V \simeq V';\\
 & M :: 16..20 + 3..3, \mathit{w\_day},\ M' :: 21..22 + 3..5, \mathit{day},\ A :: 20..20, \mathit{day},\ B :: 2..2, \mathit{month},\\
 & B' :: 31..31 + 31..31, \mathit{day},\ V :: 4..4, \mathit{week},\ V'18..18 + 7..7, \mathit{day}\ \rangle\\
\mapsto_{M \simeq M'} & \langle M' \mathit{after}\ A, M \simeq M', M' \mathit{before}\ B', B \simeq B', V' \mathit{overlaps}\ M', V \simeq V';\\
 & M :: 16..17 + 3..3, \mathit{w\_day},\ M' :: 21..22 + 3..5, \mathit{day},\ A :: 20..20, \mathit{day},\ B :: 2..2, \mathit{month},\\
 & B' :: 31..31 + 31..31, \mathit{day},\ V :: 4..4, \mathit{week},\ V'18..18 + 7..7, \mathit{day}\ \rangle
\end{array}$$

Since no further rule is applicable at this step, CaTTS' constraint propagation algorithm terminates with this (bounds consistent) reduced problem. The CaTTS' user may request one or all solutions to this problem which are searched using backtracking.

5.6 Complexity of the Multi-Calendar Constraint Solver

CSPs as introduced in Section 5.1 date back to work in Artificial Intelligence, in particular, picture processing and computer vision [Mon74, Wal75, Ste81].

It follows from the NP-completeness of SAT [GJ79], a well-known NP-complete problem, that solving arbitrary CSPs is (at least) NP-complete. SAT is essentially the problem of finding if a Boolean formula (expressed in propositional logics) is satisfiable or not. Thus, it can be considered as a particular type of CSP. Multi-calendar constraint solving in CaTTS can be reduced to the SAT problem by redefining the proof rules introduced in Section 5.5 and summarized in Table 5.6, Table 5.4, Table 5.5, Table 5.8, Table 5.7, and Table 5.9 in a SAT solver.

In what follows, the complexity of CaTTS' multi-calendar constraints propagation algorithm is analyzed.

Property 5.1 *CaTTS' constraint propagation algorithm for multi-calendar appointment scheduling problems expressed by time constraints on events and tasks, modeled by typed interval domains with conversion constraints has a worst time complexity of* $O((c + c * n) * (v + c * n) * s)$ *where* $(c + c * n)$ *is the number of constraints,* $(v + c * n)$ *is the number of variables, and* s *is the size of the interval domains representing activities.*

Proof 5.2 *The time constraints are implemented using the arithmetic built-in Prolog constraints* $\{=, =<, <, >, >=, \backslash =\}$ *which take* constant time *to compute. Associated domain constraints of each variable can be found in* constant time *since CHR uses indexing.*[15]

CaTTS' activities, i.e. events and tasks are represented by interval domains with additional type information. The type information associated with each variable can be found in constant time*. The* (maximal) size *of the interval domains to represent activities is determined as follows.*

- $s := m - n + 1$ *maximal size of an event* $X :: n..m$
- $s := (m - n + 1) * (d^+ - d^- + 1)$ *maximal size of a task* $X :: n..m + d^-..d^+$

A CSP $\mathcal{P}$ *modeled in CaTTS-CL contains the following* (maximal) number *of different* variable *and* time constraints*.*

- v *number of different variables*
- c *number of different n-ary time constraints*

[15]Recall that the prototype implementation of CaTTS' constraint solver is implemented in CHR embedded into Sicstus Prolog.

Each CaTTS-CL time constraint $X^{\alpha}\, C\, Y^{\beta}$ is transformed into the following set of CL_{catts} constraints:

$$X^{\iota}\, C\, Y^{\iota} \wedge X^{\alpha} \simeq X^{\iota} \wedge Y^{\beta} \simeq Y^{\iota}$$

where $\iota := \alpha \vee \beta$ is the join of types γ and β according to Proposition 3.1 such that $\alpha \leq \iota$ and $\beta \leq \iota$ (recall that $\leq$ denotes CaTTS' subtype relation). This transformation is done before constraint propagation. Thus, it has no influence on the complexity of the constraint propagation algorithm. But it increases the (maximal) number *of* constraints *and* variables*:*

- *In the worst case, each n-ary time constraint must be replaced by an equivalent constraint with activities over the same type and n conversion constraints. Therefore, the worst case number of conversion constraints is $O(c * n)$. The overall worst case number of constraints then is $O(c + c * n)$.*
- *In the worst case, each n-ary constraint must be replaced by an equivalent constraint with activities over the same type and n conversion constraints, i.e. $O(c * n)$ conversion constraints. Each conversion constraint introduces an additional variable. Therefore, the worst case number of variables is $O(v + c * n)$.*

The number of variables $O(v + c*n)$ and the number of constraints $O(c+c*n)$ both do not increase during constraint propagation: each time a proof rule of CaTTS' constraint propagation algorithm is applied which either transforms the constraints in the constraint store or adds new interval domains to already existing variables. The propagated new interval domain of a variable and its old interval domain are replaced by a single interval domain by applying the bounds consistency rule.

Each application of a proof rule generates a fixed number of new interval domain constraints. For each new interval domain constraint the bounds consistency rule is applied. In the declarative proof rules given in Table 5.6, Table 5.4, Table 5.5, Table 5.8, Table 5.7, and Table 5.9, bounds consistency is directly incorporated. Furthermore, each application of a proof rule generates a fixed number of applications of the conversion constraint where each conversion constraint, in turn, generates a fixed number of new interval domain constraints which require application of bounds consistency, as well.

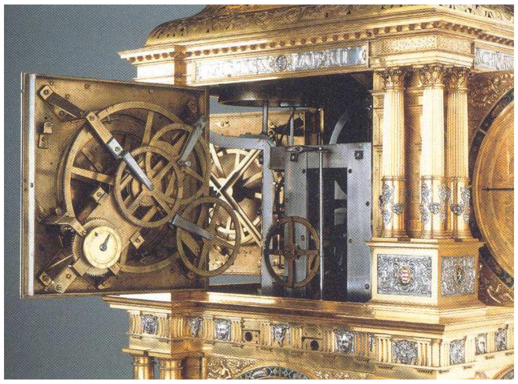
Each application of a conversion constraints requires an application of a conversion function. Since those conversion functions are generated before constraint propagation (from the type predicates of calendric types defined in a CaTTS-DL calendar specification), a conversion function is applied in constant time*. The worst case number of rule tries (where each rule try has* constant time*) therefore is $O(c + c * n)$.*

With this result, the worst case number of rule applications is $O((v + c * n) * s)$, i.e. each rule can be applied for a variable at most in the maximal size of the corresponding interval domain. To conclude, the overall worst time complexity is $O((c + cn)(v + cn)s)$. In particular, the overall worst time complexity is **linear** *in the number of constraints as well as in the number of variables.*

Note: CaTTS' current prototype implementation provides conversion function generation from the type predicates only for some predicate type constructors and not for the complete language CaTTS-DL. So far, conversion function generation from type predicates including irregularities or exceptions is done by approximating lower and upper bounds for the conversions. When extending this implementation to full CaTTS-DL, it might become necessary to search the whole domain (up to its origin) of some type to find a better approximation than the domain itself. Consequently, constraint solving in CaTTS would become quadratic, since conversion functions would be then applied in linear time complexity in the size of the type's domain.

Chapter 6

An Approach to Predicate Subtyping with Calendric Types



(astronomical clock, Eberhard Baldewein, Hans Bucher, Hermann Diepel, by order on the elector of Saxonia, 1563–1568)

Having defined the calendar type language CaTTS (cf. Chapter 4) that consists of the two sub-languages CaTTS-DL, a type definition language for calendars and CaTTS-CL, a constraint language to express calendric constraints as well as a multi-calendar constraint solver for CaTTS-CL (cf. Chapter 5), a formalization of calendric types in CaTTS is still missing.

This chapter is dedicated to the type system underlying CaTTS. A type system is a formal specification of a *type language* (usually a sub-language of a more complex language) together with a finite set of inference rules, the *typing relation*, defining conditions under which expressions (in the object language to be typed) are well-formed and well-behaved.

CaTTS' type system serves the following objectives:

1. Checking *well-formedness* of calendars specified in CaTTS-DL, i.e. "meta-type checking" of the type language itself.

2. Generating *conversion functions* from the (well-formed) type predicates.

3. Checking *correctness* and (to some extend) *consistency* of constraint programs written in the language CaTTS-CL and typed after CaTTS-DL calendar specifications using subtyping approaches.

4. *Transforming* CaTTS-CL programs into the language CL_{catts} which is used by CaTTS' constraint solver. That means in particular that CaTTS-CL programs are evaluated by transforming their typing and subtyping derivations into another language, i.e. the language CL_{catts}. This is obtained in terms of a coercion semantics for subtyping.

Furthermore, CaTTS' type language is based on predicate subtypes rather than on structural type constructors commonly used in programming languages.

This chapter, provides with a type system and discussions on typing CaTTS-DL calendar specifications and CaTTS-CL constraint programs with a good choice concerning a minimum of type annotations, however easy to read and write programs and an efficient (i.e. in polynomial time complexity) static type checker with additional dynamic checks. At first, some basic notions and concepts of typing, in particular of subtyping necessary to understand CaTTS' type system are recalled. Advantages of CaTTS' programming language approach to calendric data modeling and reasoning using types and type checking approaches are gathered and the concept of predicate subtypes as it is used in CaTTS is introduced. Subsequently, conversion function generation from CaTTS-DL type predicates is specified and illustrated by example. A type system to check well-formedness of CaTTS-DL calendar specifications is formalized, complemented with a discussion on equivalence of calendric type definitions. Typing and subtyping relations for CaTTS-CL programs typed after CaTTS-DL calendar specifications as well as a coercion semantics for subtyping in CaTTS-CL are defined. Coherence for an implementation of this coercion semantics is recovered. Finally, typing CaTTS-DL calendars themselves is addressed. Whenever advisable, larger examples conclude the several sections.

Appendix C is associated with this chapter. This appendix summarizes the Haskell-based [Tho99] implementation of CaTTS' type checkers testing well-formedness of CaTTS-DL calendar specifications, type checking CaTTS-CL programs, and transforming CaTTS-CL programs into programs in the language CL_{catts} of CaTTS' constraint solver (as introduced in the previous Chapter 5). Soundness and completeness of the implementation is sketched.

6.1 (Sub-)Typing in a Nutshell

This section can be skipped by readers familiar with type checking and subtyping.

This section briefly recalls the main aspects of *(sub)typing* in the areas of (theoretical) computer science including logics, proof assistents, and automated theorem provers. A well-known type system with subtyping, the *simply type lambda calculus with subtypes* is recalled. Different *subtyping semantics* are addressed as well as advanced forms of subtyping, in particular, *predicate subtyping* and *dependent types*.

In modern software engineering, types and type checking are applied for different purposes: types complement data with machine readable and processable semantics. Type checking is a very popular and well established "lightweight formal method" to ensure program and system behavior and to enforce high-level modularity properties. Types and type checking enhance efficiency and consistency of (modern) programming and modeling languages.

Answers to the questions *"what are types?"* and *"what is type checking?"* can hardly be given since types and type checking cover various usages of programming and modeling languages. Usually, a type system (i.e. the formal specification of types and a type checking algorithm) is defined by a sub-language of the complete language. This type expression sub-language should be sufficiently rich to support types for all object language expressions with which one wishes to compute, but sufficiently tractable to permit decidable and efficient (i.e. in polynomial time complexity) type checking[1]. In general, the type expression sub-language is specified by a context-free grammar. The type checker (i.e. the algorithm that implements the rules for typing object language expressions specified by the type system) is typically build into a compiler such that it must be able to do its job automatically *before* running a program, i.e. *static* (at compile time), with no manual intervention or interaction with the programmer, i.e. the type checker must embody computationally tractable analyses. But the programmer may "guide" the type checker in the form of explicit *type annotations* in programs. Usually, these annotations are kept fairly light, to make programs easier to write and read. A well-designed statically typed language will never require huge amounts of type information to be explicitly maintained by the programmer. It is a matter of language design how much explicit type annotation is too much. For example, the designers of Standard ML keep type annotation to a minimum, using *type inference* methods to recover the necessary type information. In general, type checking algorithms perform two different tasks:

1. *check* whether the type annotations made in a program are fulfilled, and

2. *infer* the type of an object language expression in a program (this is also called *type reconstruction*).

[1]The notion type checking is used whenever static type checking is referred to; otherwise the notion dynamic checking is used.

A type checker should not just be static and automatic in principle, but it should come with *efficient* algorithms for checking types. Efficient here means in polynomial time complexity. However, what exactly counts as efficient, and when *dynamic* checking (i.e. at run time) is necessary is a matter of debate depending on concrete problems of a specific programming or modeling language.

Besides type checking programs, one is also interested in the semantics of the type expression sub-language, in particular, what types denote, and what relations among type expressions exist. The most basic relation among type expressions is *type equivalence*. But also similar relations, weaker than type equivalence, exist such as *(type) inclusion* or, more general, *subtyping*. Subtyping gives rise to deal with subtypes and inheritance. In languages with subtyping, an element can be viewed as belonging to many different types. Those types need not to be disjoint, i.e. they may be inclusions of types. Here a language expression can be viewed as belonging to any different types which are inclusions of each other. That means that in a language with subtyping, many types may be given to a single object language expression. Similarity relations among type expressions that permit a type expression to denote more than one type, or to be compatible with many types, are referred to as *polymorphism*[2].

6.1.1 The Simply Typed Lambda Calculus with Subtyping

The simply typed lambda calculus is the (theoretical) foundation for most existing statically typed languages such as ML, Haskell, and OCaml. In the 1930s, Church [Chu36, Chu40] invented a formal system in which all computation is reduced to function definition and application, the *lambda calculus.* In the 1960s, Landin [Lan64, Lan65, Lan66] took the lambda calculus as a core to formulate a complex programming language. McCarthy's work on Lisp [McC62] has led to widespread use of the lambda calculus in programming language design and implementation as well as to various enrichments and combinations of this calculus with other calculi such as the object calculus [AC96].

Syntax. The syntax of the lambda calculus has three sorts of expressions[3]. The syntax is given in a BNF-like grammar:

```
e ::=            expressions:
        x            variables
        λx.e     abstractions
        e e      applications
```

A variables $\mathtt{x}$ itself is an expression. The abstraction of a variable $\mathtt{x}$ from an expression $\mathtt{e}_1$, written $\lambda\mathtt{x}.\mathtt{e}_1$, is an expression. Finally, the application of an expression $\mathtt{e}_1$ to another expression $\mathtt{e}_2$, written $\mathtt{e}_1\ \mathtt{e}_2$, is an expression.

[2]Subtyping is usually considered as a specific form of polymorphism, called inclusion (or subtype) polymorphism [Mit96, Pie02]

[3]In this text, the notion *expression* refers to object language expression.

Types. To construct a type system for the language of the lambda calculus, typing rules for variables, abstractions, and applications are needed that

1. maintain *safety* (also called *soundness*), i.e. that well-typed expressions[4] do not reach a stuck state, i.e. do not go wrong, and
2. are not too conservative, i.e. they should assign types to most programs.

Since the lambda calculus is essentially a calculus for representing functions, a type classifying expressions whose evaluation results in functions is needed. Such a type is usually called *arrow type* and it is denoted by $\rightarrow$. Adding a typing rule for $\rightarrow$ to lambda abstractions, one might classify the types of the variables to be abstracted. Furthermore, to give a useful type to the result of an application, one might need to know what is the type of the result the function returns. To keep track of this information, the following type expression (type, for short) is needed:

$$\begin{array}{lll} \tau ::= & & \textit{types:} \\ & \tau \rightarrow \tau & \textit{function types} \end{array}$$

$\tau_1 \rightarrow \tau_2$ denotes an infinite family of types, each classifying functions that expect arguments of type τ_1 and return results of type τ_2. Note that the *type constructor* $\rightarrow$ is right-associative. For example, assuming a type `Bool` for booleans in the given framework, `Bool` $\rightarrow$ `Bool` is the type of functions mapping boolean arguments to boolean results.

The Typing Relation. A *typing relation* assigns a type to any expression in the considered language. In the simply typed lambda calculus, variables, abstractions, and applications must be typed. The typing relation for the simply typed lambda calculus is given in Table 6.1.

A variable (cf. the typing rule (T-Var) in Table 6.1) has whatever type one might currently assume it to have. The premise $\texttt{x}{:}\tau \in \Gamma$ is read: “the type assumed for `x` in the context Γ is τ”.

A *typing context* (*context*, for short) is given by the following:

$$\begin{array}{lll} \Gamma ::= & & \textit{contexts:} \\ & \emptyset & \textit{empty contexts} \\ & \Gamma, \texttt{x}{:}\tau & \textit{variable binding} \end{array}$$

A context is a sequence of variables and their types. The “comma” operator extends the context Γ by adding a new binding (i.e. a variable and its type) on the right. The empty context is written $\emptyset$. Note that always “fresh” variables are chosen to avoid naming conflicts between a new binding and any bindings that may be in the context Γ.

[4] A well-typed expression is either a value or it can take a step according to the language’s evaluation rules.

$$\frac{\mathtt{x}\!:\!\tau \in \Gamma}{\Gamma \vdash \mathtt{x} \;:\; \tau} \quad \text{(T-Var)}$$

$$\frac{\Gamma,\mathtt{x}\!:\!\tau_1 \vdash \mathtt{e}_2\!:\!\tau_2}{\Gamma \vdash \lambda\mathtt{x}\!:\!\tau_1.\mathtt{e}_2\!:\!\tau_1 \rightarrow \tau_2} \quad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash \mathtt{e}_1\!:\!\tau_{11} \rightarrow \tau_{12} \qquad \Gamma \vdash \mathtt{e}_2\!:\!\tau_{11}}{\Gamma \vdash \mathtt{e}_1\;\; \mathtt{e}_2\!:\!\tau_{12}} \quad \text{(T-App)}$$

Table 6.1: Typing relation of the simply typed lambda calculus.

Assigning a type to an abstraction like $\lambda\mathtt{x}\!:\!\tau$ the following informations need to be calculated. What happens when the abstraction is applied to some argument? And how to know what type of arguments to expect? In the typing rule (T-Abs) for lambda abstractions given in Figure 6.1, the intended type of a function's arguments is annotated, written $\lambda\mathtt{x}\!:\!\tau_1.\mathtt{e}_2$, where the annotation on the bounded variable tells the type checker to assume that the argument `x` is of type τ_1. Knowing the type of the argument to the lambda abstraction, the type of the function's result obviously is the type of the body, denoted $\mathtt{e}_2$, where occurrences of `x` in $\mathtt{e}_2$ are assumed to have type τ_1. The typing rule (T-Abs) given in Figure 6.1 mirrors this reading.

The typing rule (T-App) for function applications (cf. Figure 6.1): if the expression $\mathtt{e}_1$ evaluates to a function mapping elements of type τ_{11} to elements of type τ_{12}, and if $\mathtt{e}_2$ evaluates to a result of type τ_{11}, then the result of applying $\mathtt{e}_1$ to $\mathtt{e}_2$ is of type τ_{12}. This rule applies under the assumption that the values represented by the free variables in expression $\mathtt{e}_1$ have the types assumed for them in the context Γ.

Subtyping. One of the simplest type systems with subtyping is an extension of the previously recalled simply typed lambda calculus to subtyping. The syntax of the language remains unchanged. The type `Top` is added to the type expression sub-language:

τ ::=		*types:*
	...	
	Top	*maximum type*

The existing typing relation (cf. Figure 6.1) remains also unchanged. Only a single typing rule is added, that of *subsumption*:

$$\frac{\Gamma \vdash \mathtt{e}\!:\!\sigma \qquad \sigma \leq \tau}{\Gamma \vdash \mathtt{e}\!:\!\tau} \quad \text{(T-Sub)}$$

$$\Gamma \vdash \sigma <: \sigma \qquad (\text{S-Refl})$$

$$\frac{\Gamma \vdash \sigma <: \rho \qquad \Gamma \vdash \rho <: \tau}{\Gamma \vdash \sigma <: \tau} \qquad (\text{S-Trans})$$

$$\Gamma \vdash \sigma <: Top \qquad (\text{S-Top})$$

$$\frac{\Gamma \vdash \tau_1 <: \sigma_1 \qquad \Gamma \vdash \sigma_2 <: \tau_2}{\Gamma \vdash \sigma_1 \to \sigma_2 <: \tau_1 \to \tau_2} \qquad (\text{S-Arrow})$$

Table 6.2: Subtyping relation for the simply typed lambda calculus.

That means, if σ is a subtype of τ (denoted $\sigma \leq \tau$) and if `e` is an element of type σ (denoted `e`:σ), then every such `e` is also of type τ (denoted `e:`τ).

The subsumption rule provides with a formalization of the intuitive notion of subtyping, the so-called *subset semantics.* The subsumption rule is used to connect the typing relation (cf. Table 6.1) to the *subtyping relation* (cf. Table 6.2) for the simply typed lambda calculus.

The subtyping relation for the simply typed lambda calculus with subtypes is given in Figure 6.2. The subtyping relation is defined as a reflexive and transitive relation with a maximal element called `Top,` i.e. the type of all well-typed expressions. The subtyping rule for arrow types says that $\sigma_1 \to \sigma_2$ is a subtype of $\tau_1 \to \tau_2$ if τ_1 is a subtype of σ_1 and σ_2 a subtype of τ_2.

Note: For any type one may add to the simply typed lambda calculus with subtyping like records, variants, products, and unions one may add (simple) subtyping rules. Usually, one have to add a single subtyping rule for each type constructor, taking care that the subtyping rule is sound in conjunction with the subsumption rule. Good and exhaustive surveys on type systems with subtypes are given for example in [CW85, Car96, Pie02].

6.1.2 Subtyping Semantics

Giving semantics to a language with subtyping has been thoroughly investigated and various approaches to subtyping have been developed. For example in [OB88], subtyping has been expressed similarly to polymorphism in ML-style languages, in [CW85], subtyping has been expressed through explicit mechanisms as parts of the type checking system, and in [BTCGS91], subtyping has been expressed as implicit coercion. Good surveys on semantic models of subtyping can be found in [GM94, Mit96].

Very common and wide-spreadly used semantic models of subtyping, *inclusion polymorphism* and *implicit coercion*, are recalled in the following.

6.1.2.1 Inclusion Polymorphism

Inclusion polymorphism is one possibility that gives rise to deal with subtypes and inheritance. In inclusion polymorphism, an element can be viewed as belonging to many different types which do not need to be disjoint, i.e. there may be inclusions of types. Thus, subtyping and inheritance are forms of polymorphism: many different types may be given to a single object language expression.

Inclusion polymorphism achieves polymorphic behavior through an inclusion relation (in the common set-theoretic sense) between types or sets of values, introduced into a type system by means of the rule of *subsumption*:

$$\frac{\Gamma \vdash \texttt{e}{:}\sigma \qquad \sigma \leq \tau}{\Gamma \vdash \texttt{e}{:}\tau} \quad (\text{T-Sub})$$

This rule formalizes the intuition that some types are more informative than others: we say that σ is a subtype of τ, written $\sigma \leq \tau$, to mean that any object language expression e of type σ can be used in a context where an expression of type τ is expected. In fact, every value of type σ is also described by τ (i.e. the elements of σ are a subset of the elements of τ). This interpretation of a subtype relation is therefore often called *subset semantics*.

For example, a record type `e-mail-address` describing e-mail addresses with fields for name and e-mail address is more informative than a record type `name-address` with a single field for names. Thus, the former record type is a subtype of the later one. With the afore mentioned subsumption rule, any expression of type `e-mail-address` can be used in a context where an expression of type `name-address` is expected.

Inclusion polymorphism can be found in many programming languages, in particular in object-oriented programming languages, using a specific form of inclusion polymorphism, called inheritance. Simula 67 is the earliest example of an inclusion polymorphic language. Further developments in such languages are for example Smaltalk and Java.

6.1.2.2 Implicit Coercion

Using a subset semantics in terms of inclusion polymorphism follows the intuition that subtyping is “semantically insignificant”, i.e. it has no effect on the way programs are evaluated. But some “intuitively reasonable” inclusions between base types such as integers and floats may have effect on this semantics. For example, introducing the axiom $integer \leq float$ implies under the subset semantics that integer values must be a subset of the set of floats. However, integers and floats have entirely different representations in most language implementations. Similar problems also arise when record types are combined with subtyping.

To overcome such problems with the subset semantics, a different semantics of subtyping is adopted: *implicit coercion* (also called *coercion semantics (of subtyping)*). With such a coercion semantics of subtyping, subtyping is “compiled away” by replacing it with run-time coercion. Intuitively, coercion semantics for subtyping is expressed by a function that transforms expressions of the high-level language with subtyping into expressions of a lower-level language without subtyping. No evaluation rules are given for the high-level language. Instead, expressions of the high-level language are type checked (using the high-level typing and subtyping rules), their typing derivations are translated into the lower-level language, and finally, the evaluation relation of this lower-level language is used to obtain the high-level language expressions' operational behavior.

Implicit coercion is wide-spreadly used in object-oriented languages such as Java for base types such as integers and floats as well as for method invocation between subclasses.

6.1.3 Predicate Subtypes and Dependent Types

Predicate subtypes and dependent types are stronger forms of typing and subtyping that enable to encode more information in types. They have been investigated in type theory, logics, proof assistents, and theorem proving.

Predicate Subtypes. Using subtyping as in the simply type lambda calculus with subtyping, one can consider for example a value of type $\mathbb{N}$ also to be of type $\mathbb{Z}$, i.e. it allows one type to be a subtype of another type. But this means for subtyping does not solve problems of statically check violations such as division by zero or out-of-bound array references. *Predicate subtypes* [ROS98] allow for such kinds of checks.

Predicate subtypes allow for encoding more (compared to "usual" types) information in types. Therefore, they enable a stronger form of subtyping. A predicate subtype is created as a subtype from another type τ by a corresponding predicate p as follows. For any type τ, the predicate $p : \tau \rightarrow \mathbb{B}$ defines the set of those elements of τ that satisfy p. Examples of predicate subtypes are non-negative integers (i.e. in set notation $\{x : \mathbb{Z} \mid x > 0\}$), a predicate subtype of type `integer`, and integer lists with $n \in \mathbb{N}$ members (i.e. in set notation $\{l : \mathbb{Z}^* \mid \mathit{length}(l) = n\}$), a predicate subtype of type `list of integers`. In general, predicate subtyping allows type expressions to contain arbitrary predicates.

Predicate subtypes are particularly used in specification languages for proof assistents and theorem provers, for example in [ROS98]. In such systems, predicate subtypes are used

- to express side conditions of theorems,
- to statically check violations such as division by zero, and
- to allow for more consistency checks.

Although predicate subtypes seam a promising approach to enhance the advantages of static type checking, static type checking becomes undecidable in type systems with predicate subtyping. Consequently, the programmer has to annotate programs with hints and explanations to guide the type checker and/or theorem proving efforts are required for type checking.

Note: Predicate subtypes are powerful enough to express dependent types [SNP90, Tho91, Luo94, Hof97]. But in contrast to dependent types, predicate subtypes incorporate the concept of subtyping.

Dependent Types *Dependent types*, mostly investigated in theoretical computer science and logics [SNP90, Tho91, Luo94, Hof97], provide an abstraction mechanism for applying types to different object language expressions.

Dependent types provide a means to describe programs much more precisely than ordinary types. In the following, dependent types are exemplified. Assume a built-in type `IntList` (denoting the type of lists of integers) with some related functions:

```
nil    :  IntList
cons   :  Int → IntList → IntList
head   :  IntList → Int
isnil  :  IntList → Bool
```

Assuming a language with dependent types, the type `IntList` can be refined to a type of lists with `n` elements, denoted by `IntList n`. To take advantage of this refinement and to give more precise types to the functions, dependencies between function arguments and function results need to be expressed. This is achieved by a refinement of the arrow type to a *dependent function type*. Dependent function types are denoted with $\Pi\mathtt{x}{:}\tau_1.\tau_2$, precising the arrow type $\tau_1 \rightarrow \tau_2$. The variable `x` represents the function's argument (of type τ_1) such that it can be mention in the result type τ_2. This dependent function type captures the dependencies between the values of the arguments and that of the results of a function. The refined versions of the list functions using dependent types then have the following types:

```
nil   :  IntList 0
cons  :  Πn:Nat.Int → IntList n → IntList (succ n)
head  :  Πn:Nat.IntList (succ n) → Int
```

Note that a function `isnil` is no more needed since the type `IntList` can be now tested whether `n` is 0 or not.

Checking that functions with dependent types actually belong to the intended types yields in *proving* that a function meets its specification. The idea for such proofs is based on the observation that a constructive proof of a theorem of the form " for every x exists a y such that a predicate p holds" can be viewed as a function from x to y. That this function has the property p is of evidence for the type checker. Unfortunately, type checking with dependent types (which requires a combination with theorem proving) becomes computationally intractable.

Since dependent types lack in computational tractability, attempts have been made to restrict the power of those types. Dependent types (and also predicate types) are used in building proof assistants and automated theorem provers and work on theorem proving environments such as AutoMath [dB80], NuPRL [CAB$^+$84], LEGO [LP92], ELF [Pfe94], and PVS [ROS98]. A survey on such work is given in [Pfe96].

Combination of dependent types with subtyping has been initially investigated by Cardelli [Car88]. This work has been advanced and generalized in [Asp94, CL96, Zwa99, AC01].

6.2 Properties and Advantages of Calendric Types

For programming languages, type systems and their associated type checkers are intended to ensure the absence of certain undesirable behaviors during program execution. Such checks that obey the typing rules of the language are usually performed at compile time, i.e. statically before running a program. The undesired behaviors generally include untrapped errors such as adding a boolean to an integer. If the language is "type safe", then all programs that can exhibit these undesired behaviors will be rejected during type checking.

Execution is not a primary concern of the calendar type specification language CaTTS-DL. In fact, it is merely a language to specify what a day or a teaching term (in some specific context of the calendar used) is. But (meta-) type checking CaTTS-DL calendar specifications can still serve to reject calendar specifications that are erroneous or undesirable in some way. For example, a calendric type that is constructed by conjunction of hours and working weeks is not reasonable at all and should be rejected (statically) before using such a type declaration in some program.

Of course, we want to ensure that no CaTTS-CL program is executed that contains undesired behavior that can be syntactically determined such as shifting an activity by an activity rather than a duration. But the primary concern of a constraint problem is to check whether the problem is *consistent*. Calendric types can be used to make consistency checks of such programs more efficient, and they allow for earlier inconsistency detection during constraint propagation. For example, a CaTTS-CL constraint problem that contains two event variables X of type *day* and Y of type *week* and the time constraint $Y\ during\ X$ fails without any further constraint propagation taking place, because the duration of Y in terms of days, i.e. 7 is larger than the duration of X in terms of days, i.e. 1 such that Y cannot be during X. Such simplifications of CaTTS-CL constraints can be implemented into the constraint solver thanks to the calendric types' semantics. Since CaTTS is an environment that contains a constraint solver, it is feasible to contemplate that type checking CaTTS-CL can rely on constraint solving, and is thus not restricted to syntactic checks of untrapped errors.

Above all, calendric types and type checking in CaTTS-CL provide multi-calendar constraint solving with the semantics of the different calendric types. Calendric types enable earlier and larger domain reduction during constraint propagation (cf. Chapter 5) which makes multi-calendar constraint solving much more efficient. Furthermore, constraint problems that are (obviously) inconsistent can be rejected by the type checker.

6.2.1 Concise Modeling, Documentation, and Annotation

The definition language CaTTS-DL (cf. Chapter 4), as a means for modeling temporal and calendric data and calendars in a declarative way to allow for context-aware modeling and reasoning, is *open* inasmuch that the definition language is neither restricted to a particular calendar nor to specific calendric and temporal data. Instead, a programmer can specify any cultural and/or professional calendar his/her application might need. The definition language provides with a small, but powerful set of *type constructors* based on specifying predicate sets. Such types complement the temporal and calendric data with machine readable and processable semantics.

Calendric data such as dates are probably more than any other data domain a subject to user interpretation: for example, the date "12/02/2005" is interpreted in France as 12^{th} February 2005 while it is interpreted as 2^{nd} December 2005 in the US or the calendric expression "Friday evening" might either refer to the eve of Friday or to the eve of Thursday. Calendric types can be used to give such data their intended meanings. Thus, calendric types are essentially another form of program and document annotation than ontologies that hardly becomes outdated.

6.2.2 Multi-Calendar Support: Modularity, Reuse, and Maintenance

To model and query temporal data in information systems like the (Semantic) Web, the calendar used by some programmer along with his/her cultural, legal, business, or even private context must be taken into account. Having in mind the Web sites of a university, for example basing on temporal data of an *educational calendar* involving concepts like lectures, courses, examinations, consultation hours, and teaching terms, or of a personal appointment book, basing on temporal data of a *business calendar* involving business conferences, meetings as well as personal work out times, for example. Those kinds of calendars, which might be called *professional calendars,* are usually established over any *cultural calendar* used. Cultural calendars like the Gregorian calendar or the Islamic calendar base on the *cultural* context a person lives and works in. Since the (Semantic) Web is international and thus, inherently heterogeneous, the possibility to define and used arbitrary professional and cultural calendars in CaTTS for Web languages suggests itself and it is thus indispensable.

Having once defined a cultural calendar such as the Gregorian or Islamic calendars in CaTTS, it should be reusable several times, for example to be extended with an educational calendar in one case and with a business calendar in some other case. Reuse, and thus, modularity and maintenance of CaTTS-DL calendar specifications is supported since calendars are defined in terms of reusable, extensible, and parametric modules (cf. Chapter 4).

6.2.3 Calendar-Conversion Functionality

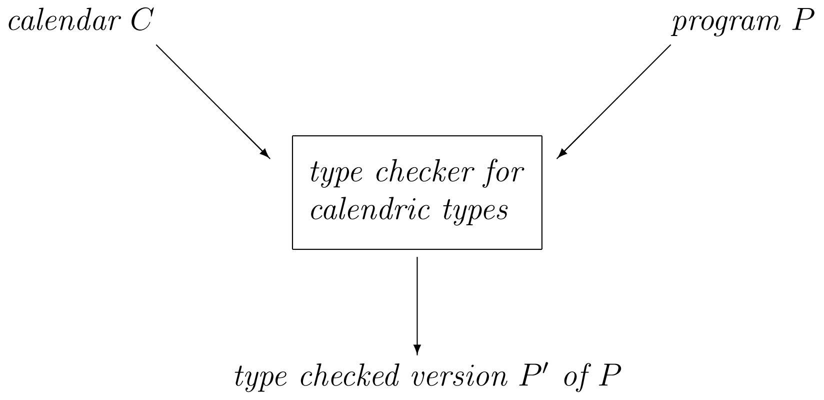
The various cultural and professional calendars in use today define miscellaneous concepts to capture calendric data like days, lessons, working shifts, a person's birthday date, consultation hours of a physician, etc. A computer implementation dealing with such data needs a way to compare and compute it in a uniform manner without loss of semantics inherent to the different calendric concepts. Fortunately, in CaTTS-DL, calendric types are always defined in terms of aggregation and/or inclusion subtypes of other calendric types by specifying predicate sets. Defining subtyping with such calendric types using a subtype semantics of implicit coercion introduces a means to convert values from one type to another whenever required in a program. In fact, this conversion functionality (based on subtyping) provides a means to perform multi-calendar constraint solving efficiently and without loss of semantics as it would be the case if any calendric data would be converted to a single reference type like seconds.

6.2.4 Multi-Calendar Constraint Solving

As mentioned in the previously stated objectives and as introduced with CaTTS' constraint solver (cf. Chapter 5), multi-calendar constraint solving is performed *independently, efficiently* and *without loss of semantics* of the data. The reason for this is that type predicates specify a conversion from values of one calendric type to those of another which are related to each other in CaTTS' conversion constraint. Efficiency of multi-calendar constraint solving is enhanced due to calendric types.

6.2.5 Use in Different Web Languages

CaTTS is in principle designed to be used for calendric data modeling and reasoning in *any* language, in particular, Web and Semantic Web languages such as XML Schema, XQuery, RDF, and OWL. Similar, to ensure the meaning of XML Schema base types like `float`, `string`, or `date` and `dateTime` in an XQuery program or an OWL ontology, arbitrary calendric data could be represented and processed in an XQuery program or an OWL ontology by using calendric data and date and time formats enriched with type annotations after some calendars specified in CaTTS-DL. Type checking such annotated data ensures the desired semantics and program behavior: consider a calendar C specified in CaTTS-DL and a program P in the language XQuery, OWL, or any other program P with type annotations referring to types and/or formats specified in the calendar C. An appropriate type checker verifies and/or extends the type annotations in P generating a type checked version P' of P:


calendar C
program P
type checker for calendric types
type checked version P′ of P

Note: Depending of the language in mind CaTTS should be used with, type checking calendric data typed after calendars specified in CaTTS-DL needs to be adapted to the requirements of the considered language. The subsequently introduced (simple) static type checking rules for CaTTS-CL programs and the coercion semantics of subtyping for CaTTS-CL meet the requirements of a constraint language with calendric reasoning approaches to express and solve multi-calendar appointment scheduling problems referring to calendric types defined in some CaTTS-DL calendar specifications.

6.3 Predicate Subtypes in CaTTS

Recall that CaTTS-DL is a programming language approach to modeling time and calendar data using calendars as types. One of CaTTS-DL's fundamental principles is to define time granularities by means of *predicate subtypes* as calendric types either by aggregation or inclusion of another calendric type. Predicate subtypes defined in a CaTTS-DL calendar specification are referred to by CaTTS-CL and its constraint solver. CaTTS-DL uses predicate types in a different manner than predicate types are used for both dependent types and specification languages of proof assistents and theorem provers (cf. Section 6.1):

1. CaTTS uses predicate types as a means to define time granularities as calendric types such that they can be manipulated by computer programs. CaTTS' predicate types represent time granularities including complex ones like Hebrew months. Calendric data types defined in CaTTS-DL belong to (possibly different) CaTTS-DL calendar specifications.

2. In CaTTS, the definition of predicate subtypes is syntactically restricted: predicate subtypes define either aggregations or inclusions of other calendric types (cf. Chapter 3 and Chapter 4). To this aim, type constructors for aggregation subtypes and for inclusion subtypes are supported which have been introduced in Chapter 4.

3. In CaTTS, predicate subtypes are used to define a coercion semantics for subtyping with calendric types, meeting the requirements of multi-calendar constraint solving

over arbitrary calendar domains that refer to calendric types defined in a CaTTS-DL calendar specification. In particular, predicate subtypes provide a means to relate calendric data represent in different calendric types to each other such as the set of weeks $\{1, 2\}$ to the set of intervals of days $\{\{1, 2, \dots, 7\}, \{8, 9, \dots, 14\}\}$ and to efficiently manipulate such related time sets during constraint solving. That is, predicate subtypes are used to generate conversions from values of one calendric type to values in some other calendric type which are applied during constraint solving.

4. Since CaTTS's typing and subtyping relations are developed not only for a calendar type definition language (i.e. CaTTS-DL) but also for a calendar constraint language (i.e. CaTTS-CL) some design choices can be considered that are not available for programming languages. In particular, CaTTS comes along with a constraint solver (cf. Chapter 5). Therefore, it is feasible to contemplate that type checking can be composed with constraint solving. In particular, CaTTS' predicate subtypes can be used for consistency checks in CaTTS-DL calendar specifications such as out-off bound violations of a finite type like the era of the current Japanese emperor, called "Heisei" as well as in CaTTS-CL programs such as the inconsistency of the constraint X *during* Y if X is of type *week* and Y is of type *day*. In CaTTS' current prototype implementation, such checks are performed dynamically during constraint solving.

Note: In CaTTS' current implementation consistency checks are exclusively performed *dynamically* during constraint solving. Dynamic checks fit perfectly well with CaTTS-CL's coercion subtype semantics. Furthermore, CaTTS constraint solver just implements a consistency test for time constraints on calendar domain constraints that refer to calendric types defined in CaTTS-DL. However, one might figure out at least a small set of CaTTS-CL language constructs for which consistency checks can be performed statically before running the constraint solver. This issue is further discussed in Section 7.2.1.3.

6.4 Conversion Function Generation from Type Predicates

From each (user-defined) predicate that defines a calendric type as a subtype of another calendric type either by aggregation or inclusion using CaTTS-DL language constructs in a CaTTS-DL calendar specification, a *conversion function* is automatically generated by CaTTS-DL's language processor. The conversion constraint of CaTTS' multi-calendar constraint solver (cf. Table 5.9, p.152) applies those conversion functions.

6.4.1 Definition of the Conversion Function

Formally, a conversion function $c_{\sigma\to\tau}$ for calendric types σ and τ defined in a CaTTS-DL calendar specification $\mathcal{C}$[5] where $\sigma \leq \tau$ (i.e. σ is a subtype of τ) and its inverse $c_{\tau\to\sigma}$ are defined as follows.

Definition 6.1 (Conversion Function). *Let σ and τ calendric types defined in calendar $\mathcal{C}$ and $\sigma \leq \tau$. The* **conversion function** *$c_{\sigma\to\tau}$ and its inverse $c_{\tau\to\sigma}$ are defined as follows for indices $i, j \in \mathbb{Z}$*

$$c_{\sigma\to\tau}(\sigma(i)) \quad := \quad [\tau(j_1), \ldots, \tau(j_k)] \qquad \textit{where} \quad \begin{array}{l}[\tau(j_1), \ldots, \tau(j_k)] = \sigma(i)\\ 1 \leq k\end{array}$$

$$c_{\tau\to\sigma}(\tau(i)) \quad := \left\{ \begin{array}{lll} \sigma(j) & \textit{if} & \tau(i) \subseteq \sigma(j) \\ & & \\ \{max(\{k \mid k < i \wedge \tau(k) \subseteq \sigma(l)\}), & & \\ min(\{k \mid k > i \wedge \tau(k) \subseteq \sigma(l)\})\} & \textit{otherwise} & \end{array}\right.$$

With Definition 6.1, $c_{week\to day}(1)$ yields the interval of days $[day(1), day(2), \ldots, day(7)]$, assumed that $week \leq day$ and that weeks are defined from days such that each week contains 7 days, always starting on Mondays and such that $day(1)$ refers to a Monday. Inversely, $c_{day\to week}(6)$ yields the week $week(1)$, i.e. the week that contains the day with index 6. To give another example, $c_{working\_day\to day}(6)$ yields the day $day(8)$, assumed that $working\_day \leq day$ and that working days are defined from days such that they refer to the weekdays Monday to Friday in each week and such that $day(1)$ refers to a Monday. Inversely, $c_{day\to working\_day}(6)$ yields the set of working days $\{working\_day(5), working\_day(6)\}$, i.e. the set of those working days which immediately preceed and succeed the day $day(6)$, since $day(6)$, a Saturday, is not a working day.

According to Definition 6.1, $[\tau(j_1), \ldots, \tau(j_k)]$ denotes a (finite) interval of values in the type τ and $\{max(\{k \mid k < i \vee \tau(k) \subseteq \sigma(l)\}, min(\{k \mid k > i \vee \tau(k) \subseteq \sigma(l)\}\}$ denotes the set of the immediate predecessor and the immediate successor according to $\tau(i)$ in type σ, if no corresponding value to $\tau(i)$ in σ exists.

Applying this conversion function during constraint propagation in CaTTS' conversion constraint requires in some cases a variation: for example, if a day d should explicitly refer to a starting time of a week w, this is only possible if d *starts* this week w; otherwise, the week that succeeds the week d is contained in must be chosen. Similarly, if a day d should explicitly refer to an ending time of a week w, this is only possible if d *finishes* this week w; otherwise the week that preceeds the week d is contained in must be chosen.[6]. For such

[5]Recall that $\mathcal{C}$ refers to an aligned set of calendars which is considered as a single calendar by CaTTS-DL's processor.

[6]in terms of the interval relations *starts* and *finishes*

cases, the conversion function from Definition 6.1 comes with two versions, denoted $c^-_{\sigma\to\tau}$ and $c^+_{\sigma\to\tau}$ (resp. $c^-_{\tau\to\sigma}$ and $c^+_{\tau\to\sigma}$), the former to compute the immediate predecessor and the latter to compute the immediate successor.

Definition 6.2 (Conversion Function for Predecessors and Successors). *Let σ and τ calendric types defined in calendar $\mathcal{C}$ and $\sigma \leq \tau$. The* **conversion function for predecessors** *$c^-_{\sigma\to\tau}$ and its inverse $c^-_{\tau\to\sigma}$ and the* **conversion function for successors** *$c^+_{\sigma\to\tau}$ and its inverse $c^+_{\tau\to\sigma}$ are defined as follows for indices $i, j \in \mathbb{Z}$:*

$$c^-_{\sigma\to\tau}(\sigma(i)) \quad := \quad \tau(j_1) \quad \textit{where} \quad [\tau(j_1), \ldots, \tau(j_k)] = \sigma(i), 1 \leq k$$

$$c^+_{\sigma\to\tau}(\sigma(i)) \quad := \quad \tau(j_k) \quad \textit{where} \quad [\tau(j_1), \ldots, \tau(j_k)] = \sigma(i), 1 \leq k$$

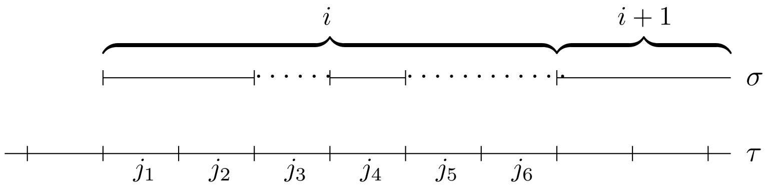
$$c^-_{\tau\to\sigma}(\tau(i)) \quad := \begin{cases} \sigma(j) & \textit{if} \quad \tau(i)\ \textit{equals}\ \sigma(j) \vee \tau(i)\ \textit{finishes}\ \sigma(j) \\ \begin{array}{l} \textit{max}(\{k \mid k < i \wedge \\ (\tau(k)\ \textit{equals}\ \sigma(l) \vee \\ \tau(k)\ \textit{finishes}\ \sigma(l))\}) \end{array} & \textit{otherwise} \end{cases}$$

$$c^+_{\tau\to\sigma}(\tau(i)) \quad := \begin{cases} \sigma(j) & \textit{if} \quad \tau(i)\ \textit{equals}\ \sigma(j) \vee \tau(i)\ \textit{starts}\ \sigma(j) \\ \begin{array}{l} \textit{min}(\{k \mid k > i \wedge \\ (\tau(k)\ \textit{equals}\ \sigma(l) \vee \\ \tau(k)\ \textit{starts}\ \sigma(l))\}) \end{array} & \textit{otherwise} \end{cases}$$

With Definition 6.1, $c^-_{week\to day}(1)$ yields the day $day(1)$ that starts the week $week(1)$, assumed that $week \leq day$ and that weeks are defined from days such that each week contains 7 days, always starting on Mondays and such that $day(1)$ refers to a Monday. And $c^+_{week\to day}(1)$ yields the day $day(7)$ that finishes the week $week(1)$. Inversely, $c^-_{day\to week}(6)$ yields the week $week(0)$, i.e. the week that preceeds the week that contains the day with index 6. And $c^+_{day\to week}(6)$ yields the week $week(2)$, i.e. the week that preceeds the week that contains the day with index 6. To give another example, $c^-_{working\_day\to day}(6)$ yields the day $day(8)$, assumed that $working\_day \leq day$ and that working days are defined from days such that they refer to the weekdays Monday to Friday in each week and such that $day(1)$ refers to a Monday as it is also the case for $c^+_{working\_day\to day}(6)$. Inversely, $c^-_{day\to working\_day}(6)$ yields the working day $working\_day(5)$, i.e. the working day that immediately preceeds the day $day(6)$ in terms of working days, since $day(6)$, a Saturday, is not a working day. And $c^+_{day\to working\_day}(6)$ yields the working day $working\_day(6)$, i.e. the working day that immediately succeeds the day $day(6)$ in terms of working days.

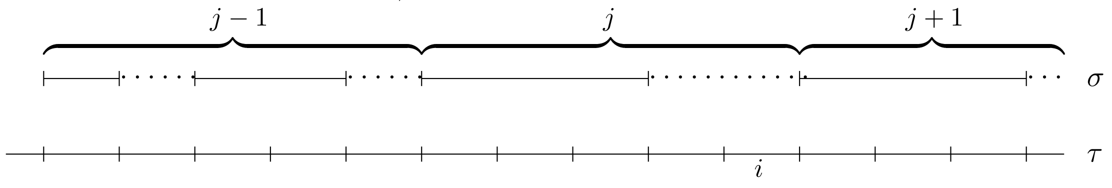
An illustration of the conversion function and its specialized forms for immediate predecessors and immediate successors is given in Figure 6.1.

The conversion function, $\sigma \leq \tau$:



$c_{\sigma\rightarrow\tau}(\sigma(i)) = [\tau(j_1), \tau(j_2), \tau(j_4)]$
$c^{-}_{\sigma\rightarrow\tau}(\sigma(i)) = \tau(j_1)$
$c^{+}_{\sigma\rightarrow\tau}(\sigma(i)) = \tau(j_4)$

Inverse of the conversion function, $\sigma \leq \tau$:



$c_{\tau\rightarrow\sigma}(\tau(i)) = \sigma(j)$
$c^{-}_{\tau\rightarrow\sigma}(\tau(i)) = \sigma(j-1)$
$c^{+}_{\tau\rightarrow\sigma}(\tau(i)) = \sigma(j+1)$

Figure 6.1: Illustration of the conversion function $c_{\sigma\rightarrow\tau}$ and its inverse $c_{\tau\rightarrow\sigma}$ for $\sigma \leq \tau$.

6.4.2 Conversion Function Generation from Aggregation Subtypes

In what follows, the different possibilities to define an aggregation subtype in a CaTTS-DL calendar specification using CaTTS' language constructs for aggregation subtype predicates are considered. The conversion function and its inverse generated from such a type predicate by CaTTS-DL's language processor are given together with an example for each case.

6.4.2.1 Periodic Aggregations

In CaTTS-DL, periodic aggregation subtypes can be declared using the following CaTTS-DL language construct.

$$\textbf{type}\ \sigma\ =\ \textbf{aggregate}\ \mathrm{d}_1\ \tau,\ \ldots,\ \mathrm{d}_k\ \tau\ @\ \tau(\mathrm{a});$$

where

- d_i, $i \in \{1, \ldots k\}$ is the duration of some value of type σ in terms of values of type τ,
- $k \in \mathbb{N}$ is the length of the ordered periodic pattern of k values of type σ according to values of type τ,
- $a \in \mathbb{Z}$ the anchor index of type σ in type τ,
- $\sigma \preceq \tau$, and thus, $\sigma \leq \tau$.

For example, the type `week` can be defined by a periodic aggregation of the type `day` since each week contains 7 days.

From such a periodic aggregation predicate, the conversion function $c_{\sigma\to\tau}$, its two variants $c^-_{\sigma\to\tau}$ and $c^+_{\sigma\to\tau}$ and its inverse $c_{\tau\to\sigma}$ with variants $c^-_{\tau\to\sigma}$ and $c^+_{\tau\to\sigma}$ are generated as follows:

$$c_{\sigma\to\tau}(i) := \textbf{let } i \textit{ mod } k = m \quad \textbf{in } \bigcup_{j=d_1+\ldots+d_{m-1}}^{(d_1+\ldots+d_m)-1} (d_1+\ldots+d_k) \times ((i-1) \textit{ div } k) + a + j$$

$$c^-_{\sigma\to\tau}(i) := \textbf{let } i \textit{ mod } k = m \quad \textbf{in } (d_1+\ldots+d_k) \times ((i-1) \textit{ div } k) + a + (d_1+\ldots+d_{m-1})$$

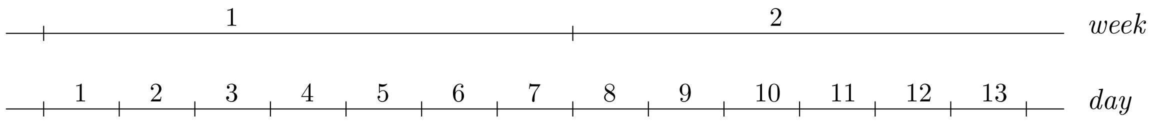
$$c^+_{\sigma\to\tau}(i) := \textbf{let } i \textit{ mod } k = m \quad \textbf{in } (d_1+\ldots+d_k) \times ((i-1) \textit{ div } k) + a + (d_1+\ldots+d_m - 1)$$

$$c_{\tau\to\sigma}(i) := (i-a) * k \textit{ div } (d_1+\ldots+d_k) + 1$$

$$c^-_{\tau\to\sigma}(i) := \textbf{if } (i-a) * k \textit{ mod } (d_1+\ldots+d_k) == d_1+\ldots+d_k - 1 \quad \textbf{then } (i-a) * k \textit{ div } (d_1+\ldots+d_k) + 1 \quad \textbf{else } (i-a) * k \textit{ div } (d_1+\ldots+d_k)$$

$$c^+_{\tau\to\sigma}(i) := \textbf{if } (i-a) * k \textit{ mod } (d_1+\ldots+d_k) == 0 \quad \textbf{then } (i-a) * k \textit{ div } (d_1+\ldots+d_k) + 1 \quad \textbf{else } (i-a) * k \textit{ div } (d_1+\ldots+d_k) + 2$$

Assume an internal indexing of type `week` and type `day` as given in the following illustration:



Further assume that the type `week` is defined as an aggregation subtype of type `day` by the following predicate subtype in a CaTTS-DL calendar specification.

type week = **aggregate** 7 day @ day(1);

Then the following conversions can be computed by applying the previously given patterns to generate conversions for periodic aggregation subtypes.

$c_{week\to day}(2)$	$=$	$\{8,\ldots,15\}$	the interval of days that define $week(2)$
$c^-_{week\to day}(2)$	$=$	8	the day that starts $week(2)$
$c^+_{week\to day}(2)$	$=$	14	the day that finishes $week(2)$
$c_{day\to week}(9)$	$=$	2	the week which contains $day(9)$
$c^-_{day\to week}(9)$	$=$	1	the week that preceeds the week containing $day(9)$
$c^+_{day\to week}(9)$	$=$	3	the week that succeeds the week containing $day(9)$

Note: $c^{-}_{day \rightarrow week}(8)$ yields $week(2)$ because $day(8)$ starts this week according to Definition 6.2, p.175.

6.4.2.2 Periodic Aggregations with finite many Exceptions

In CaTTS-DL, periodic aggregation subtypes with finite many exceptions can be declared using the following CaTTS-DL language construct.

```
type σ = aggregate
                        alternate σ(i)
                              cond_11 → d_11 τ
                              ...
                              cond_1l → d_1l τ
                        end
                        ...
                        alternate σ(i)
                              cond_k1 → d_k1 τ
                              ...
                              cond_kl → d_km τ
                        end
                    @ τ(a);
```

where

- $D_{i \in \{1,\dots k\}}$, where every D_i is the set of possible durations for a specific phase i, and $D = \bigcup D_{i \in \{1,\dots k\}}$, i.e. $D = \{\{d_{11}, \dots, d_{1l}\}, \dots, \{d_{k1}, \dots, d_{km}\}\}$
- k is the length of the ordered pattern of values of type σ in terms of intervals of values of type τ,
- a the anchor index of type σ in τ,
- *cond* a condition formulated using arithmetic expressions supported in CaTTS-DL,
- $\sigma \preceq \tau$, and thus, $\sigma \leq \tau$.

For example, the type (Gregorian) `month` can be defined by a periodic aggregation with finite many exceptions from type (Gregorian) `day` since each month contains either 28, 30 or 31 days with the exception that the February contains from time to time an extra day (due to leap year regulations).

Let $d_{1_i} \in D_1, \dots, d_{k_i} \in D_k$ the durations of $\sigma(i)$ satisfying the conditions $cond_{1_i}$ to $cond_{k_i}$ specified in the predicate of σ for the value of type σ with index i. Then the conversion function is defined as follows for starting times and ending times.

$$\begin{array}{lll} c_{\sigma\to\tau}(i) & := & \textbf{let}\ \ i\ mod\ k = m \\ & & \textbf{in}\ \ \bigcup_{j=d_{1_i}+\ldots+d_{m_i-1}}^{(d_{1_i}+\ldots+d_{m_i})-1} (d_{1_i}+\ldots+d_{k_i}) \times ((i-1)\ div\ k) + a + j \end{array}$$

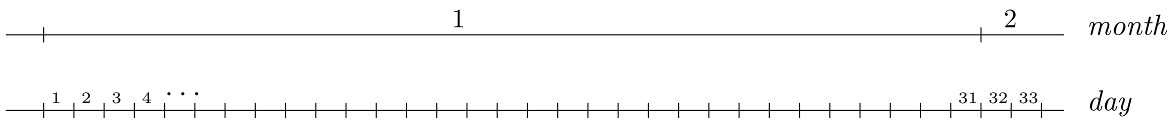
$$\begin{array}{lll} c^{-}_{\sigma\to\tau}(i) & := & \textbf{let}\ \ i\ mod\ k = m \\ & & \textbf{in}\ \ (d_{1_i}+\ldots+d_{k_i}) \times ((i-1)\ div\ k) + a + (d_{1_i}+\ldots+d_{m_i-1}) \end{array}$$

$$\begin{array}{lll} c^{+}_{\sigma\to\tau}(i) & := & \textbf{let}\ \ i\ mod\ k = m \\ & & \textbf{in}\ \ (d_{1_i}+\ldots+d_{k_i}) \times ((i-1)\ div\ k) + a + (d_{1_i}+\ldots+d_{m_i}-1) \end{array}$$

Inversely, the conversion function is defined by approximation to the minimal and the maximal length of the k elements in a cycle. Let for $i \in \{1,\ldots,m\}$ $min(d_{1_i}+...+d_{k_i})$ the minimal length of one cycle and for $i \in \{1,\ldots,m\}$ $max(d_{1_i}+...+d_{k_i})$ the maximal cycle. Let further K denote the number of cycles of length k i is contained in. Then the inverse of the conversion function is defined as follows.

$$\begin{array}{lll} c_{\tau\to\sigma}(i) & := & min(i)..max(i) \\[1em] min(i) & := & (i-a)*k*K\ div\ max(d_{1_i}+\ldots+d_{k_i})*K+1 \\[1em] max(i) & := & (i-a)*k*K\ div\ min(d_{1_i}+\ldots+d_{k_i})*K+1 \\[1em] min^{-}(i) & := & \textbf{if}\ \ (i-a)*k*K\ mod\ (max(d_{1_i}+\ldots+d_{k_i})*K) == \\ & & max(d_{1_i}+\ldots+d_{k_i})*K \\ & & \textbf{then}\ \ (i-a)*k*K\ div\ (max(d_{1_i}+\ldots+d_{k_i})*K) \\ & & \textbf{else}\ \ (i-a)*k*K\ div\ (max(d_{1_i}+\ldots+d_{k_i})*K)-1 \\[1em] max^{-}(i) & := & \textbf{if}\ \ (i-a)*k*K\ mod\ (min(d_{1_i}+\ldots+d_{k_i})*K) == \\ & & min(d_{1_i}+\ldots+d_{k_i})*K \\ & & \textbf{then}\ \ (i-a)*k*K\ div\ (min(d_{1_i}+\ldots+d_{k_i})*K) \\ & & \textbf{else}\ \ (i-a)*k*K\ div\ (min(d_{1_i}+\ldots+d_{k_i})*K)-1 \\[1em] c^{-}_{\tau\to\sigma}(i) & := & min^{-}(i)..max^{-}(i) \\[1em] min^{+}(i) & := & \textbf{if}\ \ (i-a)*k*K\ mod\ (max(d_{1_i}+\ldots+d_{k_i})*K) == 0 \\ & & \textbf{then}\ \ (i-a)*k*K\ div\ (max(d_{1_i}+\ldots+d_{k_i})*K)-1 \\ & & \textbf{else}\ \ (i-a)*k*K\ div\ (max(d_{1_i}+\ldots+d_{k_i})*K)+1 \\[1em] max^{+}(i) & := & \textbf{if}\ \ (i-a)*k*K\ mod\ (min(d_{1_i}+\ldots+d_{k_i})*K) == 0 \\ & & \textbf{then}\ \ (i-a)*k*K\ div\ (min(d_{1_i}+\ldots+d_{k_i})*K)-1 \\ & & \textbf{else}\ \ (i-a)*k*K\ div\ (min(d_{1_i}+\ldots+d_{k_i})*K)+1 \\[1em] c^{+}_{\tau\to\sigma}(i) & := & min^{+}(i)..max^{+}(i) \end{array}$$

Assume an indexing of type `month` and type `day` as given in the following illustration:



and assume that `month` is defined from `day` by the following predicate subtype in a CaTTS-DL calendar specification.

```
type month = aggregate
      31 day named january ,
      alternate month(i)
        | ((i div 12) mod 4 == 0 &&
          ((i div 12) mod 400 != 100 &&
           (i div 12) mod 400 != 200 &&
           (i div 12) mod 400 != 300)    -> 29 day
        | otherwise                      -> 28 day
      end named february ,
      31 day named march,
      30 day named april ,
      31 day named may,
      30 day named june ,
      31 day named july ,
      31 day named august ,
      30 day named september ,
      31 day named october ,
      30 day named november,
      31 day named december @ day(1);
```

Then the following conversions can be computed. Where $min(d_{1_i} + ... + d_{k_i})$ is 365 and $max(d_{1_i} + ... + d_{k_i})$ is 366 and K is 1 (when $month(2)$ is considered).

$c_{month \to day}(2)$	$=$	$\{32, \ldots, 59\}$	the interval of days that define $month(2)$
$c^-_{month \to day}(2)$	$=$	32	the day that starts $month(2)$
$c^+_{month \to day}(2)$	$=$	59	the day that finishes $month(2)$
$c_{day \to month}(33)$	$=$	2..2	the month which contains $day(33)$
$c^-_{day \to month}(33)$	$=$	1..1	the month that preceeds the month containing $day(33)$
$c^+_{day \to month}(33)$	$=$	3..3	the month that succeeds the month containing $day(33)$

Note: Non-periodic aggregations need additional facilities of constraint solving and memorization. This makes them not fit into the above mentioned pattern. They are not considered in CaTTS' current prototype implementation. An example of a non-periodic aggregation is the specification of Hebrew months from Hebrew days.

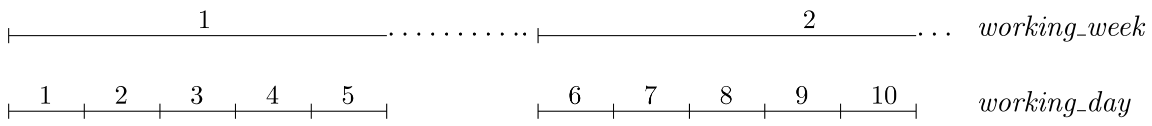
6.4.2.3 Restricted Aggregations

In CaTTS-DL, restricted aggregation subtypes (e.g. the type `working_week` can be defined by a restricted aggregation from type `working_day` according to type `week`). Such a type can be declared using the following CaTTS-DL language construct.

type $\sigma = \rho \; \#\!< \tau$;

From such a restriction predicate to specify an aggregation subtype, the conversion function (and its two variants) is generated by concatenation of the conversion functions from ρ to τ and its inverse is generated from concatenation of the inverse conversion functions form τ to ρ.[7]

Assume an indexing of type `working_week` and `working_day` as given in the following illustration:



and assume that `working_week` is defined from `working_day` (assuming that working days are specified as Monday to Friday in each week) according to type `week` by the following predicate subtype in a CaTTS-DL calendar specification.

type working_week = week #< working_day;

Then the following conversions can be computed.

[7]Recall that the internal indexing of σ and ρ is the same.

$c_{working\_week \to day}(2)$	$=$	$\{6, \ldots, 10\}$	the interval of working days that define $working\_week(2)$
$c^{-}_{working\_week \to day}(2)$	$=$	6	the working day that starts $working\_week(2)$
$c^{+}_{working\_week \to day}(2)$	$=$	10	the working day that finishes $working\_week(2)$
$c_{working\_day \to working\_week}(7)$	$=$	2	the working week which contains $working\_day(7)$
$c^{-}_{working\_day \to working\_week}(7)$	$=$	1	the working week that preceeds the working week containing $working\_day(7)$
$c^{+}_{working\_day \to working\_week}(7)$	$=$	3	the working week that succeeds the working week containing $working\_day(7)$

In this case, the conversion function $c_{working\_week \to day}$ is defined by concatenation of the conversion functions $c_{week \to day}$ and $c_{day \to working\_day}$ (denoted $c_{week \to day} \cdot c_{day \to working\_day}$). Analogously, the inverse conversion function $c_{working\_day \to working\_week}$ is defined by concatenation of the inverse conversion functions $c_{working\_day \to day}$ and $c_{day \to week}$ (denoted $c_{working\_day \to day} \cdot c_{day \to week}$).

6.4.3 Conversion Function Generation from Inclusion Subtypes

In what follows, the different possibilities to define an inclusion subtype in a CaTTS-DL calendar specification using CaTTS' language constructs for inclusion subtype predicates are considered. The conversion functions generated from such inclusion subtype predicates by CaTTS-DL's language processor are given.

6.4.3.1 Selections

An inclusion subtype predicate can be specified using the CaTTS-DL selection language construct that has the following general syntactic form.

type σ = **select** $\tau(i)$ **where** P;

where P can be specified using the following CaTTS-DL language constructs.

- *relative i in[to]* $\tau \odot k$, $\odot \in \{=, \leq, <, >, \geq, \neq\}$,
- *q(relative i in[to]* τ*)*, $q \in \{min, max\}$, and
- conjunctions (denoted &&) and disjunctions (denoted ||) of such constructs.

Since the patterns for conversion function generations for selections using (disjunctions and conjunctions of) the relative in/to language constructs are very similar, only two cases are given for demonstration purposes.

Case 1:

type ρ = **select** σ(i) **where relative** i **in** τ == k;

Defining ρ as an inclusion subtype of σ, i.e. $\rho \subseteq \sigma$.

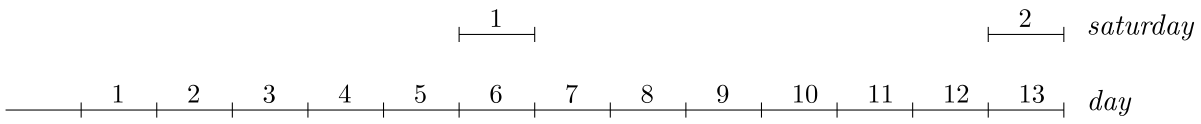
From such an inclusion type declaration in CaTTS-DL the following pattern for conversion function generation is derived.

```
$c_{\rho\to\sigma}(i)$   :=  $d \times (i-1) + a + k - 1$
$c^-_{\rho\to\sigma}(i)$  :=  $c_{\rho\to\sigma}(i)$
$c^+_{\rho\to\sigma}(i)$  :=  $c_{\rho\to\sigma}(i)$

$c_{\sigma\to\rho}(i)$   :=  let $\tau\_of\_\sigma = ((i-1)\ div\ d) + 1$ in
               if $c_{\rho\to\sigma}(\tau\_of\_\sigma) == i$
               then $\tau\_of\_\sigma$
               else $\{c^-_{\sigma\to\rho}(i), c^+_{\sigma\to\rho}(i)\}$

$c^-_{\sigma\to\rho}(i)$  :=  let $\tau\_of\_\sigma = ((i-1)\ div\ d) + 1$ in
               if $(((i-1)\ mod\ d) + 1) < k$
               then $\tau\_of\_\sigma - 1$
               else $\tau\_of\_\sigma$

$c^+_{\sigma\to\rho}(i)$  :=  let $\tau\_of\_\sigma = ((i-1)\ div\ d) + 1$ in
               if $(((i-1)\ mod\ d) + 1) <= k$
               then $\tau\_of\_\sigma$
               else $\tau\_of\_\sigma + 1$
```

where

- $\sigma \preceq \tau$
- d denotes the duration of $\sigma(i)$ in τ which is computed by converting $\sigma(i)$ to τ using $c_{\sigma\to\tau}(i)$ and then the duration of the corresponding interval in terms of σ in τ is determined
- a denotes the anchor of σ in τ (according to the aggregation subtype definition of τ from σ)
- $k \leq d$

Assume an indexing of type `saturday` and type `day` as given in the following illustration:



Further assume that `saturday` is defined from `day` by the following predicate subtype in a CaTTS-DL calendar specification.

```
type saturday = select day(i) where relative i in week == 6;
```

Then the following conversions can be computed.

$$\begin{array}{lcll} c_{saturday\to day}(2) & = & 13 & \text{the day that corresponds to } saturday(2) \\ c^{-}_{saturday\to day}(2) & = & 13 & \\ c^{+}_{saturday\to day}(2) & = & 13 & \\ & & & \\ c_{day\to saturday}(9) & = & \{1,2\} & \text{the Saturdays that preceed and succeed } day(9) \\ c^{-}_{day\to saturday}(9) & = & 1 & \text{the Saturday that preceed } day(9) \\ c^{+}_{day\to saturday}(9) & = & 2 & \text{the Saturday that succeed } day(9) \end{array}$$

Case 2:

```
type ρ = select σ(i) where
                 relative i in τ >= l && relative i in τ <= k;
```

Defining ρ as an inclusion subtype of σ, i.e. $\rho \subseteq \sigma$.

From such an inclusion type declaration in CaTTS-DL the following pattern for conversion function generation is derived:

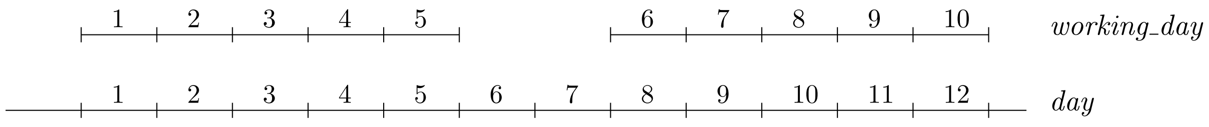
$$\begin{array}{lcl} c_{\rho\to\sigma}(i) & := & d \times ((i-1)\ div\ length([l..k])) + a + ((i-1)\ mod\ length([l..k])) \\ c^{-}_{\rho\to\sigma}(i) & := & c_{\rho\to\sigma}(i) \\ c^{+}_{\rho\to\sigma}(i) & := & c_{\rho\to\sigma}(i) \end{array}$$

```
$c_{\sigma\to\rho}(i)$ := let $\tau\_of\_\sigma = ((i-1)\ div\ d) + i$ in
         if $c_{\rho\to\sigma}(\tau\_of\_\sigma) == i$
         then $\tau\_of\_\sigma$
         else $\{c^{-}_{\sigma\to\rho}(i), c^{+}_{\sigma\to\rho}(i)\}$

$c^{-}_{\sigma\to\rho}(i)$ := let $\tau\_of\_\sigma = ((i-1)\ div\ d) + i$ in
         if $(((i-1)\ mod\ d) + 1) >= l$ and $(((i-1)\ mod\ d) + 1) <= k$
         then $(length([l..k]) \times \tau\_of\_\sigma) - length([l..k]) + (i\ mod\ d)$
         elseif $(((i-1)\ mod\ d) + 1) < l$ then $length([l..k]) \times (\tau\_of\_\sigma) - 1)$
         else $length([l..k]) \times \tau\_of\_\sigma$

$c^{+}_{\sigma\to\rho}(i)$ := let $\tau\_of\_\sigma = ((i-1)\ div\ d) + i$ in
         if $(((i-1)\ mod\ d) + 1) >= l$ and $(((i-1)\ mod\ d) + 1) <= k$
         then $(length([l..k]) \times \tau\_of\_\sigma) - length([l..k]) + (i\ mod\ d)$
         elseif $(((i-1)\ mod\ d) + 1) < l$ then $(length([l..k]) \times (\tau\_of\_\sigma) - 1)) + 1$
         else $(length([l..k]) \times \tau\_of\_\sigma) + 1$
```

where

- $\sigma \preceq \tau$
- d denotes the duration of $\sigma(i)$ in τ which is computed by converting $\sigma(i)$ to τ using $c_{\sigma\to\tau}(i)$ and then the duration of the corresponding interval in terms of σ in τ is determined
- a denotes the anchor of σ in τ (according to the aggregation subtype definition of τ from σ)
- $length([l..k])$ denotes the length of the list $[l..k]$ (from element l to element k), $l \leq k$
- $k \leq d$

Assume an indexing of type `working_day` and type `day` as given in the following illustration:



Further assume that `working_day` is defined from `day` by the following predicate subtype.

```
type working_day = select day(i) where
                   relative i in week >= 1 && relative i in week <=6;
```

Then the following conversions can be computed.

$c_{working\_day\to day}(7)$	$=$	9	the day that corresponds to $working\_day(7)$
$c^{-}_{working\_day\to day}(7)$	$=$	9	
$c^{+}_{working\_day\to day}(7)$	$=$	9	
$c_{day\to working\_day}(6)$	$=$	$\{5,6\}$	the working days that preceed and succeed $day(6)$
$c^{-}_{day\to working\_day}(6)$	$=$	6	the working day that preceeds $day(6)$
$c^{+}_{day\to working\_day}(6)$	$=$	5	the working day that succeeds $day(6)$

Note: Selections specified by conditions using time constraints other than *relative in/to* such as *after* additionally need facilities of constraint solving. They are not considered in CaTTS' current prototype implementation. The time constraint *relative in/to* (with conjunctions and disjunctions of it) has been chosen as the first constraint to be implemented since with such predicates most inclusion subtypes that might appear in a calendar specification can be expressed as illustrated by the examples given Chapter 4.

Note: Approximations to lower and upper bounds for conversions between inclusion subtypes defined using the *relative* construct are used whenever the type within which the values of some other type are related is defined by aggregation with finite many exceptions. For example, the type `first_week_in_month` defined for `week(i)` by the predicate `relative i in month == 1`. In this case, the join of types `week` and `month` is computed which is `day`. For the types `week` and `month` the minimal and maximal number of days that could be included are computed. The conversion function is then approximated by a range from the number of times the minimal length of week in terms of days is included in the minimal length of month in terms of days to the number of times the maximal length of week in terms of days is included in the maximal length of month in terms of days.

6.4.3.2 Conjunctions

An inclusion subtype predicate can be specified using the CaTTS-DL type constructor for conjunctions that has the following general syntactic form.

type σ = τ_1 & τ_2;

The conversion function for conjunction types is defined by (set-theoretic) conjunction of the type predicates defining types τ_1 and τ_2 such that the previously defined patterns for conversion function generation of selections can be used.

For example, a CaTTS-DL type that specifies those days which are Mondays and the first days in a month can be defined by conjunction of the type `monday` and `first_day_in_month` as follows in CaTTS-DL.

```
type monday_and_first_in_month = monday & first_day_in_month;
```

Assuming that the type `monday` is defined by selection with

```
type monday = select day(i) where relative i in week == 1;
```

and `first_day_in_month` is defined by selection as follows

```
type first_day_in_month = select day(i) where
                                 relative i in month == 1;
```

Then the conversion function for the type `monday_and_first_in_month` to type `monday` and the conversion function for the type `monday_and_first_in_month` to type `first_day_in_month` can be defined by conjunction of those two predicates using the previously introduced pattern for conversion function generation for inclusion subtypes defined by the selection constructor.

6.4.3.3 Disjunctions

An inclusion subtype predicate can be specified using the CaTTS-DL type constructor for disjunctions that has the following general syntactic form.

type σ = τ_1 | τ_2;

The conversion function for disjunction types is defined by (set-theoretic) disjunction of the type predicates defining types τ_1 and τ_2 such that the previously defined patterns for conversion function generation of selections can be used.

Note: Defining an inclusion type by disjunctions results in a supertype of the two types the disjuncted type is constructed from. Thus, conversion functions are generated, from each of the subtypes to the (new) supertype and from this supertype to its possible subtypes.

For example, a CaTTS-DL type that specifies those work days which are working days (as previously defined) or Saturdays (as previously defined) as follows using the disjunction type constructor.

```
type work_day = working_day | saturday;
```

Then the conversion function from the type `working_day` to type `work_day` and the conversion function from type `saturday` to type `work_day` (as well as the inverse functions) can be defined by disjunction of those two predicates using the previously introduced pattern for conversion function generation for inclusion subtypes defined by the selection constructor.

6.4.3.4 Exceptions

An inclusion subtype predicate can be specified using the CaTTS-DL type constructor for exceptions that has the following general syntactic form.

type $\sigma = \tau_1 \setminus \tau_2$;

The conversion function for exception types is defined by (set-theoretic) conjunction type predicates defining types τ_1 and τ_2 where the predicate of type τ_2 is replaced by its inverse. Thus, as it is the case for conjunctions and disjunctions, the conversion function for exceptions can be defined by the previously defined conversion function patterns for selections.

For example, a CaTTS-DL type that specifies those days which are not Saturdays (as previously defined) as follows using the exception type constructor.

```
type not_saturday = day\saturday;
```

Then the conversion function for the type `not_saturday` to type `day` can be defined by conjunction of the predicate specifying days and the inverse of the predicate specifying Saturdays (i.e. `select day(i) where relative i in week != 6`). The previously introduced pattern for conversion function generation for inclusion subtypes defined by the selection constructor can be now used.

τ ::=		*type expressions:*
	$\mathrm{r}_{\mathcal{C}}$	*reference type of* $\mathcal{C}$
	aggregate τ $\mathrm{te}_i^{i\in 1..n}$ @ te	*(abs. anchored) aggregation*
	aggregate τ $\mathrm{te}_i^{i\in 1..n}$ ~@ z	*(rel. anchored) aggregation,* $z \in \mathbb{Z}$
	select τ X:τ where te	*selection*
	$\tau \& \tau$	*conjunction*
	$\tau \mid \tau$	*disjunction*
	$\tau \setminus \tau$	*exception*
	$\tau\# < \tau$	*restriction*
K ::=		*kinding expressions:*
	τ	*point*
	τ^*	*interval*
	τ^n	*duration*
te ::=		*CaTTS-TDL expressions:*
	X	*variable*
	i	*index*
	τ(i)	*part,* $i \in \mathbb{Z}$*, index*
	index te	*index of te*
	n τ	*duration,* $n \in \mathbb{N}$
	relative te in/to $\tau \odot$ i	*relative related values in/to* τ
	m(relative te in/to τ)	*minimal/maximal value of values relatively related in/to* τ
	te && te	*conjunction*
	te || te	*disjunction*
	alternate te | te $\rightarrow$ te {| te $\rightarrow$ te}	*alternate*

where $\odot \in \{==, <=, <, >, >=, !=\}$
$m \in \{min, max\}$

Table 6.3: The (abstract) syntax of CaTTS-DL.

6.5 Well-Formed CaTTS-DL Calendar Specifications

In what follows, a set of inference rules is introduced to ensure well-formedness of CaTTS-DL calendar specifications. Those rules specify how calendric types can be defined from other calendric types in a CaTTS-DL calendar specification to yield new calendric types. Once a CaTTS-DL calendar specification has been checked for well-formedness, it can be used without hesitation in any language such as XQuery, XSLT, RDF, or OWL, and, in particular, in CaTTS-CL where constraint expressions refer to calendric types defined in such a CaTTS-DL calendar specification. The section concludes with an example testing well-formedness of a CaTTS-DL calendar specification.

6.5.1 Syntax

The (abstract) syntax of CaTTS-DL expressions is given in Table 6.3. This syntax is the output from parsing a CaTTS-DL calendar specification. It is used to define the typing relation for CaTTS-DL type and object language expressions. Thus, the abstract syntax is used by the type checker of CaTTS-DL.

The following notational conventions are used in the syntax in Table 6.3 and throughout this text: the symbol τ on the right-hand side of the CaTTS-DL syntax is metavariable, i.e. it is a placeholder for some particular calendric type. Throughout this text, we use the metavariable τ and nearby letters such as σ and ρ and variants such as τ_1 or τ'. Similarly, other letters are used as metavariables, standing for expressions drawn from other syntactic categories such as te, a metavariable for CaTTS-TDL expressions (cf. Table 6.3). To refer to a type of another kind than a (time) point, we use a *kinding statement* of the form $\forall\alpha.\ \alpha :: K$ read as "the type α is assumed to have kind K"), i.e. α is a type that is a placeholder for some particular calendric type (e.g. *day* with values such as "20^{th} April 2005"), or for some particular interval type over a calendric type (e.g. day^* with values such as "from 20^{th} April 2005 to 22^{nd} May 2005"), or for some particular duration type over a calendric type (e.g. day^n with values such as "3 day"). Writing $\forall\alpha.\ \alpha :: \tau \vee \alpha :: \tau^*$, α is a placeholder for a point type or an interval type over some particular calendric type τ.

Apart from that, the syntactic forms given in Table 6.3 are a subset of the CaTTS-TDL expressions introduced in Chapter 4, as they appear after parsing a CaTTS-DL calendar specification. This subset is chosen such that the language remains expressive enough to define cultural calendars such as the Gregorian as well as professional calendars such as an academic calendar of a university. This subset corresponds to those CaTTS-DL expressions to construct type predicates for which conversion function generation is defined, yet.

Further differences from the CaTTS-DL syntax given in Appendix A appear from some parser outputs: a (finite set of aligned) CaTTS-DL calendar specification(s) $\mathcal{C}$ may include at most one calendric type defined by the `refinement` constructor (cf. Chapter 4) as supertype of $\mathcal{C}$'s reference type `ref`. During parsing, this type is translated to the type $r_{\mathcal{C}}$ and `ref` to an aggregation subtype of $r_{\mathcal{C}}$. Therefore, `refinement` and `ref` do not appear in the abstract syntax used to check calendar specifications for well-formedness. The type constructor for aggregations `aggregate` as well as the type constructor for selections `select` have a type attached from the parser according to their declarations.

6.5.2 Typing Relation

The rules to check well-formedness of CaTTS-DL calendric type constructors are defined straightforward on the syntactic forms of CaTTS-DL type expressions. The rules to check for well-formedness of the type declarations in a calendar specification $\mathcal{C}$ are given in Table 6.4. Those rules are completed by a collection of rules to check the correctness of CaTTS-DL expressions that can be used to declare calendric types. The rules for CaTTS-DL expressions are given in Table 6.5. To the collection of rules given in Table

6.4 and Table 6.5 to check for well-formedness of calendric types in a (finite set of aligned) calendar specification(s) $\mathcal{C}$, we add a collection of rules for an *aggregation relation* between calendric types σ and τ, written $\sigma \preceq \tau$ and an *inclusion relation* between calendric types σ and τ, written $\sigma \subseteq \tau$. Those two relations are used to check well-formedness of some of the CaTTS-DL type constructors. Recall that in a CaTTS-DL calendar specification a calendric type is always defined either by aggregation of by inclusion of another, already defined calendric type. This design principle of CaTTS-DL is now exploited to enhance concise modeling of calendar specifications. The *aggregation relation* is given in Table 6.6 and the *inclusion relation* is given in Table 6.7.

The rules used to test for well-formedness of some CaTTS-DL calendar specification are defined as *inference rules* commonly used in "natural deduction style" presentations of logical systems. Such an inference rule has the form

$$\frac{p_1 \ldots p_n}{c}$$

Read as: if the statements (often called premises) p_1 , $\ldots p_n$ listed above the line have been established, then the conclusion c below the line may be derived. Rules without any premises are often called *axioms*.

Before turning attention to CaTTS-DL's typing relation for well-formedness of calendar specifications, we need to introduce *typing statements*. A typing statement has the form $\forall \alpha :: K. \, \Gamma \vdash e : \alpha$, read as "the expression e is assumed to have type α of kind K". Γ is called a *typing context* (*context*, for short). It is defined as follows:

$\Gamma ::=$		*contexts:*
	$\emptyset$	*empty contexts*
	$\Gamma, \texttt{x}{:}\tau$	*variable binding*

A context is a sequence of variables and their types. The "comma" operator extends the context Γ by adding a new binding (i.e. a variable and its type) on the right. The empty context is written $\emptyset$. Note that always "fresh" variables are chosen to avoid naming conflicts between a new binding and any bindings that may be in the context Γ.

Formation Rules. The formation rules (denoted with an F) followed by the name of the inference rule to form a (new) calendric type from another, already defined calendric type in a CaTTS-DL calendar specification are given in Table 6.4. The formation rule for aggregations (F-Aggr) is read as follows: if for all expressions $\texttt{te}_i$ where $i \in 1..n$ have type τ^n and if the expression $\texttt{te}$ has type τ, then $\texttt{aggregate te}_i^{i\in 1..n}$ `@ te` forms a new calendric type. The rules (F-Aggr˜) and (F-Sel) are defined analogously. The expressions $\texttt{te}_i$ and $\texttt{te}$ are type checked using the rules for checking correctness of CaTTS-DL expressions which are given in Table 6.5. The formation rule for conjunction types (F-And) restricts the construction of conjunction types such that the conjunction $\tau_1\&\tau_2$ is well formed, if types τ_1 and τ_2 have a common inclusion supertype τ, denoted $\tau_1 \subseteq \tau$ and $\tau_2 \subseteq \tau$. This rules prevents from construction unreasonable types such as the conjunction

$$\frac{\text{for all } i,\, \Gamma \vdash te_i : \tau^n \quad \Gamma \vdash te : \tau}{\Gamma \vdash (aggregate\ \tau\ te_i^{i\in 1..n} @te)_{type}}\ (\textsc{F-Aggr}) \qquad \frac{\text{for all } i,\, \Gamma \vdash te_i : \tau^n \quad \Gamma \vdash z : \mathbb{Z}}{\Gamma \vdash (aggregate\ \tau\ te_i^{i\in 1..n}\ \tilde{}@z)_{type}}\ (\textsc{F-Aggr}\tilde{})$$

$$\frac{\Gamma, X : \tau \vdash te : \mathbb{B}}{\Gamma \vdash (select\ \tau\ X : \tau\ where\ te)_{type}}\ (\textsc{F-Sel}) \qquad \frac{\Gamma \vdash \tau_1 \subseteq \tau \quad \Gamma \vdash \tau_2 \subseteq \tau}{\Gamma \vdash (\tau_1 \& \tau_2)_{type}}\ (\textsc{F-And})$$

$$\frac{\Gamma \vdash \tau_1 \subseteq \tau \quad \Gamma \vdash \tau_2 \subseteq \tau}{\Gamma \vdash (\tau_1 \mid \tau_2)_{type}}\ (\textsc{F-Or}) \qquad \frac{\Gamma \vdash \tau_1 \subseteq \tau \quad \Gamma \vdash \tau_2 \subseteq \tau}{\Gamma \vdash (\tau_1 \setminus \tau_2)_{type}}\ (\textsc{F-Ex})$$

$$\frac{\Gamma \vdash \tau_1 \preceq \tau \quad \Gamma \vdash \tau_2 \subseteq \tau}{\Gamma \vdash (\tau_1 \# < \tau_2)_{type}}\ (\textsc{F-Res})$$

Table 6.4: Formation rules for CaTTS-DL type declarations.

$$\frac{X : \tau \in \Gamma}{\Gamma \vdash X : \tau}\ (\textsc{T-Var}) \qquad \frac{\Gamma \vdash i : \mathbb{Z}}{\Gamma \vdash \tau(i) : \tau}\ (\textsc{T-Part})$$

$$\frac{\Gamma \vdash te_1 : \tau_1}{\Gamma \vdash index\ te_1 : \mathbb{Z}}\ (\textsc{T-Index}) \qquad \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash n\ \tau_1 : \tau_1^n}\ (\textsc{T-Dur})$$

$$\frac{\text{for all } i,\, \Gamma, x : \tau \vdash c_i : \mathbb{B}\Gamma \vdash te_i : \tau^n}{\Gamma \vdash alternate\ x : \tau \mid c_i^{i\in 1..n} \rightarrow te_i : \tau^n}\ (\textsc{T-Alter}) \qquad \frac{\Gamma \vdash te_1 : \mathbb{Z} \quad \Gamma \vdash i : \mathbb{Z} \quad \tau_{2type}}{\begin{array}{c}\Gamma \vdash relative\ te_1\ in/to\ \tau_2 \odot i : \mathbb{B}\ / \\ \Gamma \vdash m(relative\ te_1\ in/to\ \tau_2) : \mathbb{B}\end{array}}\ (\textsc{T-Rel})$$

$$\frac{\Gamma \vdash te_1 : \mathbb{B} \quad \Gamma \vdash te_2 : \mathbb{B}}{\Gamma \vdash te_1\ \&\&\ te_2 : \mathbb{B}}\ (\textsc{T-Conj}) \qquad \frac{\Gamma \vdash te_1 : \mathbb{B} \quad \Gamma \vdash te_2 : \mathbb{B}}{\Gamma \vdash te_1\ \ ||\ \ te_2 : \mathbb{B}}\ (\textsc{T-Disj})$$

Table 6.5: Typing rules for expressions used in CaTTS-DL type declarations.

of months and holidays. The subtyping statements used in this rule are checked by the inclusion relation (cf. Table 6.7). The formation rules for disjunction types and exception types are defined analogously. Finally, the formation rule for restriction types (F-Res) prevents from constructing unreasonable restrictions like the restriction of hours by working weeks as follows: the type $\tau_1 \# < \tau_2$ is well-formed only if there exists a type τ which is an aggregation supertype of τ_1, denoted $\tau_1 \preceq \tau$ an an inclusion supertype of τ_2, denoted $\tau_2 \subseteq \tau$. The subtyping statements used in this rule are checked by the aggregation relation (cf. Table 6.6) and the inclusion relation (cf. Table 6.7).

Typing Rules for CaTTS-DL expressions. The inference rules that check for correct CaTTS-DL expressions which can be used to defined calendric types in a CaTTS-DL calendar specification are given in Table 6.5. According to the rule (T-Var) a variable has whatever type one might currently assume it to have. The premise `X`$:\tau \in \Gamma$ is read: "the type assumed for `X` in the context Γ is τ". A part $\tau(i)$ has type τ, if i is an integer according

$\sigma \preceq \sigma$	(AS-REFL)	$\dfrac{\rho \preceq \sigma \quad \sigma \preceq \tau}{\rho \preceq \sigma}$	(AS-TRANS)
$(aggregate\ \tau\ te_i^{i \in 1..n}\ \tilde{}@te) \preceq \tau$	(AS-AGGR)	$(aggregate\ \tau\ te_i^{i \in 1..n}\ \tilde{}@z) \preceq \tau$	(AS-AGGR˜)
$\sigma\# < \tau \preceq \tau$	(AS-RES)		

Table 6.6: The aggregation relation over calendric types.

to the rule (T-PART). The axiom (T-INDEX) states that an index is of type integer.[8] A duration `n` τ has type τ^n, if `n` is a natural number according to rule (T-DUR). The rule (T-REL) for extracting relative indices of some activity's values formulates the condition that the language construct `relative te`$_1$ `in/to` $\tau_2 \odot$ `i` (resp. `min/max(relative te`$_1$ `in/to` τ_2`))` is of type boolean (denoted $\mathbb{B}$), if expressions `te`$_1$ and `i` are integers and τ_2 must be a calendric type defined in the considered CaTTS-DL calendar specification. The conjunction (resp. disjunction) of relations of type boolean has again type boolean according to rule (T-CONJ) (resp. (T-DISJ)). Finally, to express exceptions in the periodic pattern of aggregations, CaTTS-DL provides with the language construct `alternate` to specify alternative matches. The typing rule (T-ALTER) is read as follows: if variable `X` is assumed to have type τ, if the conditions `c`$_i$ in each of the branches have type $\mathbb{B}$, and if each of the results in the branches has type duration of τ (denoted τ^n), then the whole alternate expression has type τ^n.

The Aggregation Relation. The aggregation relation between (some) calendric types defined in a CaTTS-DL calendar specification that is called by some formation rule when testing for well-formedness is given in Table 6.6. The first two rules (AS-REFL) and (AS-TRANS) state that the aggregation relation between calendric types is a pre-order. The other three rules are mere axioms. The rules (AS-AGGR) and (AS-AGGR˜) state that calendric type defined by aggregation is an aggregation of the type it is defined from. Similarly, the rule (AS-RES) states that a calendric type defined by restriction is an aggregation of the type its values are restricted to.

The Inclusion Relation. The inclusion relation between (some) calendric types defined in a CaTTS-DL calendar specification that is called by some formation rule when testing for well-formedness is given in Table 6.7. The first two rules (IS-REFL) and (IS-TRANS) state that the inclusion relation between calendric types is a pre-order. The axiom (IS-SEL) states that a calendric type defined by selection is an inclusion of the type it is defined from. CaTTS' inclusion constructor for disjunctions has two different subtyping rules (IS-DJ$_1$)

[8]The basic types for natural number, integers, characters, strings, booleans, and lists are provided together with their usual operations for CaTTS' implementation.

$\sigma \subseteq \sigma$	(IS-Refl)	$\dfrac{\rho \subseteq \sigma \quad \sigma \subseteq \tau}{\rho \subseteq \tau}$	(IS-Trans)
$(select\ \tau\ X : \tau\ where\ te) \subseteq \tau$	(IS-Sel)	$\tau \setminus \sigma \subseteq \tau$	(IS-Ex)
$i \in \{1,2\},\ \tau_i \subseteq \tau_1 \mid \tau_2$	(IS-Dj$_1$)	$\dfrac{i \in \{1,2\},\ \sigma_i \subseteq \tau}{\sigma_1 \mid \sigma_2 \subseteq \tau}$	(IS-Dj$_2$)
$i \in \{1,2\},\ \tau_1\&\tau_2 \subseteq \tau_i$	(IS-Cj$_1$)	$\dfrac{i \in \{1,2\},\ \sigma \subseteq \tau_i}{\sigma \subseteq \tau_1\&\tau_2}$	(IS-Cj$_2$)

Table 6.7: The inclusion relation over calendric types.

and (IS-Dj$_2$). The former states that the types τ_1 and τ_2, the type $\tau_1 \mid \tau_2$ is constructed from, are both inclusion of $\tau_1 \mid \tau_2$. The latter states that if either σ_1 or σ_2 is an inclusion of τ, then the disjunction of σ_1 and σ_2 (denoted $\sigma_1 \mid \sigma_2$) is also an inclusion of τ. The two rules for the conjunction constructor (IS-Cj$_1$) and (IS-Cj$_2$) formulate the conditions that (1) a conjunction $\tau_1\&\tau_2$ is an inclusion of the types τ_1 and τ_2 it is constructed from, and that (2) an inclusion σ of either τ_1 or τ_2 is also an inclusion of the types' conjunction $\tau_1\&\tau_2$. CaTTS' constructor for exception types has a single rule (IS-Ex) stating that the exception $\tau \setminus \sigma$ is an inclusion of τ.

6.5.3 Example: Checking Well-Formedness of a CaTTS-DL calendar specification

The following example shows the application of the previously introduced inference rules to check for well-formedness of a (finite set of aligned) CaTTS-DL calendar specification(s). Given a CaTTS-DL calendar specification `CalendarExample` specifying types for days, weeks, working days, and working weeks:

```
calendar CalendarExample:Sig =
  cal
    type day;
    type week = aggregate 7 day @ day(1);
    type working_day = select day(i) where
              relative i in week >= 1 && relative i in week <= 5;
    type working_week = week #< working_day;
  end
```

The rules to check well-formedness of this calendar are applied to each of the defined types. Note that a type defined by aggregation as well a type defined by selections get a type attached from the parser (according to its syntactic form). Furthermore, a CaTTS-DL expression like `relative i in week >= 1` as it appears in the type constructor for type `working_day` is a syntactic sugaring for the user not having to write `relative (index` `day(i)) in week >= 1`. The parser performs this replacement. The type `day` is parsed to the calendar's reference type $r_{\mathcal{C}}$. The rules to check for well-formedness operate on this (abstract) syntax (cf. Table 6.3) output from the parser. Thus, the type checker operates on the following syntactic forms.

$$\begin{array}{l} day_{type} := r_{\mathcal{C}} \\ week_{type} := \mathit{aggregate\ 7\ day\ @\ day(1)} \\ working\_day_{type} := \mathit{select\ day\ day(i)\ where\ relative\ (index\ day(i))\ in\ week >= 1}\ \&\& \\ \qquad\qquad\qquad\quad \mathit{relative\ (index\ day(i))\ in\ week <= 5} \\ working\_week_{type} := week_{type}\# < working\_day_{type} \end{array}$$

Checking well-formedness of type $working\_day_{type}$ yields the following *typing derivation.*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{day(i) : day \in \{day(i) : day\}}{\{day(i) : day\} \vdash day(i) : day}\;(\text{T-Var})}{\{day(i) : day\} \vdash \mathit{index}\ day(i) : \mathbb{Z}}\;(\text{T-Index}) \qquad \{day(i) : day\} \vdash 1 : \mathbb{Z} \qquad \overset{*}{week_{type}}}{\{day(i) : day\} \vdash \mathit{relative\ (index\ day(i))\ in\ week >= 1} : \mathbb{B}}\;(\text{T-Rel}) \qquad \vdots}{\{\}, day(i) : day \vdash \mathit{relative\ (index\ day(i))\ in\ week >= 1}\ \&\&\ \mathit{relative\ (index\ day(i))\ in\ week <= 5} : \mathbb{B}}\;(\text{T-Conj})}{\{\} \vdash working\_day_{type}}\;(\text{F-Sel})$$

where $*$

$$\dfrac{\dfrac{\{day(i) : day\} \vdash 7 : \mathbb{N}}{\{day(i) : day\} \vdash 7\ day : day^n}\;(\text{T-Dur}) \qquad \dfrac{\{day(i) : day\} \vdash 1 : \mathbb{Z}}{\{day(i) : day\} \vdash day(1) : day}\;(\text{T-Part})}{\{day(i) : day\} \vdash week_{type}}\;(\text{F-Aggr})$$

And checking well-formedness of type $working\_week_{type}$ yields the following typing derivation.

$$\dfrac{\dfrac{}{\{\} \vdash week_{type} \preceq day_{type}}\;(\text{AS-Aggr}) \qquad \dfrac{}{\{\} \vdash working\_day_{type} \subseteq day_{type}}\;(\text{IS-Sel})}{\{\} \vdash working\_week_{type}}\;(\text{F-Res})$$

6.6 Note: Equivalence of Calendric Type Definitions

When talking about the semantics of the type language CaTTS-DL, we are interested in subtype relations (in terms of aggregation and inclusion) between calendric types that can be defined in a CaTTS-DL calendar specification as previously introduced. For some programming languages one might be also interested in a more basic relation among type expressions, *type equivalence.* Type equivalence gives the possibility to write the same type in different ways.

CaTTS-DL is expressive enough to define the same calendric type using different language constructs, i.e. two types which specify the same set of values, however in different ways. For example, New Year can be specified either as the first day in a year or as the first day of January as follows:

```
type new_year1 = select day(i) where relative i in year == 1;
type new_year2 = select day(i) where relative i in january == 1;
```

Although obvious for humans, types `new_year1` and `new_year2` define the same set of days, and, thus, equivalent calendric types, this equivalence cannot be detected by CaTTS-DL's type checker. To detected equivalence during type checking, we can formulate the equivalence of two types like `new_year1` and `new_year2` by a *definitional equivalence* relation on calendric types, written $\sigma \equiv \tau$. This relation can be defined by the following clause:

$$\forall \sigma, \tau \leq \iota.\ \sigma \equiv \tau \qquad \text{iff} \qquad \forall x \in \iota . p_\sigma(x) \Leftrightarrow p_\tau(x)$$

That is, two types σ and τ with a common supertype ι[9] are equivalent if for every value x that satisfies the predicate specified with type σ, denoted p_σ, the predicate of type τ, denoted p_τ, is satisfied, as well. To perform such "theorem proving" problems generated during type checking equivalence of CaTTS-DL type definitions, additional constraint solving techniques are required which are currently not supported in CaTTS. In particular, constraint solving on predicate sets (in fact, an interesting open research problem) would become necessary.

6.7 Typing and Subtyping in CaTTS-CL

So far, a set of inference rules has been specified that defines a system to (statically) check well-formedness of a (finite set of aligned) CaTTS-DL calendar specification(s). To check well-formedness of such calendar specifications, each of the syntactic forms to construct a calendric type is considered by the system. This is done be specifying a rule for each syntactic CaTTS-DL language expression. Such a rule describes how the corresponding calendric type can be constructed from other expressions of some particular syntactic form. Additionally, well-formedness exploits CaTTS' property that any calendric type is either defined by inclusion or by aggregation of another (already) defined calendric type. This property is used to statically check unreasonable calendric type declarations such as the union of Saturdays and weeks.

In what follows, a set of inference rules is specified that defines a *type system* to (statically) check correctness of CaTTS-CL programs. In this way, calendric types defined in some CaTTS-DL calendar specification can be referred to in CaTTS-CL (or any other program using calendric data typed after CaTTS-DL type definitions). Furthermore, the calendric types can be then applied by CaTTS' multi-calendar constraint solver to solve multi-calendar appointment scheduling problems expressed in CaTTS-CL.

Although calendric types and type checking is absolutely necessary for CaTTS-CL or any other language using calendric data typed after CaTTS-DL type definitions, typing CaTTS-CL should concern the user as little as possible. That means, the typing rules for

[9]Recall that $\leq$ defines a subtype relation on calendric types.

CaTTS-CL expressions must be specified as flexible as possible, however without violating the correctness of some CaTTS-CL program. This flexibility is achieved in terms of subtyping between calendric types.

Since CaTTS-CL is a constraint language for calendric data, erroneous or undesirable CaTTS-CL programs always refer to the problem whether they are *consistent*. Consistency is the matter tested by CaTTS' constraint solver. However, calendric types can be used to check statically for consistency of CaTTS-CL programs – at least in some cases – that can be drawn from the syntactic forms of the constraints.

6.7.1 Syntax

The (abstract) syntax of CaTTS-CL expressions is given in Table 6.8. This syntax is used by CaTTS-CL' s type system, typing CaTTS-CL after calendric types defined in CaTTS-DL (cf. Table 6.3, p.189 for the syntactic forms of CaTTS-DL types). The syntax corresponds to set of syntactic forms that have been defined for CaTTS-CL in Chapter 4. The syntax is given in a BNF-like notation. The syntax given in Table 6.8 corresponds to the output of CaTTS' parser. Type checking is performed on this syntax.

6.7.2 Subsumption

For the present[10], the goal of subtyping in CaTTS-CL is to type CaTTS-CL expressions such that we can write a CaTTS-CL program like

```
X is 2 working_week && Y is 1 month && X during Y
```

where the CaTTS-CL constraint `X during Y` is typeable without demanding explicit type casts. Since `X` and `Y` are both variables that refer to time, explicit type casts would be very unnatural: with common-sense understanding of time, working weeks and months can be related and compared. In fact, any pair of temporal activities independent of their time granularities (i.e. calendric types) should be comparable.

Specifying a typing rule for the CaTTS-CL time constraint `during`, it might look like as follows:

$$\frac{\Gamma \vdash ce_1 : \tau_1^* \;\; \Gamma \vdash ce_2 : \tau_1^*}{\Gamma \vdash ce_1 \; during \; ce_2 : \mathbb{B}}$$

This rule is read as follows: if the CaTTS-CL expressions ce_1 and ce_2 both have kind interval of some calendric type τ_1, denoted τ_1^*, then the CaTTS-CL time constraint has type boolean, denoted $\Gamma \vdash ce_1 \, during \, ce_2 : \mathbb{B}$. Without subtyping, this rule would reject the CaTTS-CL expression given above, because the type of `X` is `working_week`* and the type of `Y` is `month`[11] while the typing rule for the time constraint `during`, given above, demands

[10]In the subsequent Section 6.8, we will see that there is an additional goal of subtyping in CaTTS.

[11]Recall that events like `Y` (specified by duration 1) in a CaTTS-CL program refer to time points according to some calendric type, and tasks like `X` refer to time intervals according to some calendric type.

ce ::=	*CaTTS-CL expressions:*
true	
false	
k	*constant*
X	*variable*
n τ	*duration,* $n \in \mathbb{N}$
[ce..ce]	*endpoint interval*
ce to ce	*duration interval*
X is 1 τ	*event*
X is τ	*task*
X is n τ	*task with duration* $n \in \mathbb{N}$
ce $\mathcal{R}$ ce	*interval constraint*
ce $\odot$ ce	*metric constraint*
duration ce	*duration of ce*
index ce	*index of ce*
b/e ce	*starting or ending point of ce*
shift ce f/b ce	*shift in time*
e/s ce by ce	*extension/shortening of an interval*
relative ce in/to $\tau \odot$ i	relate values relatively in/to τ
m(relative ce in/to τ)	*min/max values related relatively in/to* τ
ce && ce	*conjunction*

$\mathcal{R} \in$	{equals, before, after, starts, started_by, finishes, finished_by, during, contains, meets, met_by, overlaps, overlapped_by, within, on_or_before, on_or_after}
$\odot \in$	{==, <=, <, >, >=, ! =}
to $\in$	{upto, downto}
$b/e \in$	{begin, end}
$f/b \in$	{forward,backward}
$e/s \in$	{extend, shorten}
$m \in$	{min,max}

Table 6.8: The (abstract) syntax of CaTTS-CL.

the expressions $\texttt{e}_1$ and $\texttt{e}_2$ having the same type τ_1^*. But, clearly, the time constraint `during` just requires that its two expressions are activities (i.e. events and/or tasks), independent of their underlying calendric types. To propagate this constraint, we just have to be sure that we can find some type τ_1^*, `X` and `Y` can be represented in. For example, one might express both the interval of working days `X` and the month `Y` in terms of intervals of days, because both an interval of working weeks like "calendar week 20 and 21 in 2005" is a specific interval of days and a month like "July 2005" is a specific interval of days.

Using subtyping, we can refine this typing rule for the CaTTS-CL time constraint `during` such that it can accept constraints like the one above. This is accomplished by formulating the intuition that some calendric type is coarser (and thus, more informative) than another calendric type. For example, `working_week`* is coarser and more informative than `day`* and `month` is coarser and more informative than `day`*. For example, an interval of 31 days must not necessarily refer to a month like January. That is, `month` is a *subtype of* `day`*, written `month` $\leq$ `day`*. This intuitive subtype relation between calendric types (possible of different kinds, e.g. point or interval) refers to the subtype relation (cf. Definition 3.7) over time granularities (calendric types) as defined in Chapter 3.

This intuition that values of a "finer" calendric type (i.e. a supertype) can be used instead of values of some corresponding "coarser" calendric type (i.e. a subtype), expressing an equivalent extend of time however in another granularity of temporal precision, can be formulated by adding the following typing rule of *subsumption*:

$$\frac{\forall\alpha,\beta :: K \quad \Gamma \vdash ce : \alpha \quad \alpha \leq \beta}{\Gamma \vdash ce : \beta} \quad (\text{T-S{\scriptsize UB}})$$

In CaTTS-CL, the rule of subsumption (T-Sub) is read as follows: if for every calendric types α and β independent of their kinds, written $\forall\alpha,\beta :: K$ (i.e. α and β might be points, intervals, or durations over some calendric type τ defined in a CaTTS-DL calendar specification) α is a subtype of β, then every value of of type α can be expressed in terms of some value of type β. For example, if CaTTS-CL's subtyping relation is defined such that `month` $\leq$ `day`* and `working_week`* $\leq$ `day`*, then the rule (T-Sub) can be used to derive $\vdash$ `Y: day`* (e.g. `Y = "July 2005"`) and $\vdash$ `X: day`* (e.g. `X = ["W20 2005".."W21 2005"]`) which is what is needed to make the motivating example type checked.

6.7.3 The Subtype Relation

CaTTS' *subtype relation* is defined by a finite collection of inference rules deriving statements of the form $\forall\alpha,\beta :: K.\ \alpha \leq \beta$, read as "for all types α and β of kind K, i.e. either of kind point, interval, or duration, α is a subtype of β". The subtype relation considers each form of calendric type and kind of calendric type (e.g. `aggregate`, `#<`, |, interval types, etc.) separately. For each syntactic construct one or more inference rules are defined for situations when it is allowed to uses values of one type of this syntactic form where another type is expected. CaTTS-CL's subtype relation is summarized in Table 6.9. The different inference rules are introduced in the following.

$\forall\alpha :: K.\ \alpha \le \alpha$	(S-Refl)	$\dfrac{\forall\alpha,\beta,\gamma :: K.\ \alpha \le \beta \quad \beta \le \gamma}{\alpha \le \gamma}$	(S-Trans)
$\sigma \le \sigma^*$	(S-ICoer)	$\sigma \le r_{\mathcal{C}}$	(S-Ref)
$\dfrac{\sigma \le \tau}{\sigma^n \le \tau^n}$	(S-Dur)	$\dfrac{\sigma \le \tau}{\sigma^* \le \tau^*}$	(S-Int)
$(select\ \tau\ X : \tau\ where\ te) \le \tau$	(S-Sel)	$\tau \setminus \sigma \le \tau$	(S-Ex)
$i \in \{1,2\},\ \tau_i \le \tau_1 \mid \tau_2$	(S-Dj$_1$)	$\dfrac{i \in \{1,2\},\ \sigma_i \le \tau}{\sigma_1 \mid \sigma_2 \le \tau}$	(S-Dj$_2$)
$i \in \{1,2\},\ \tau_1 \& \tau_2 \le \tau_i$	(S-Cj$_1$)	$\dfrac{i \in \{1,2\},\ \sigma \le \tau_i}{\sigma \le \tau_1 \& \tau_2}$	(S-Cj$_2$)
$(aggregate\ \tau\ te_i^{i\in 1..n}\ \tilde{}@te) \le \tau$	(S-Aggr)	$(aggregate\ \tau\ te_i^{i\in 1..n}\ \tilde{}@z) \le \tau$	(S-Aggr˜)
$\tau_1\# < \tau_2 \le \tau_2$	(S-Res)		

Table 6.9: The subtype relation of CaTTS-CL.

Before the rules for the different syntactic forms are introduced, we stipulate that the subtype relation $\le$ defines a pre-order on (possibly different kinds of) calendric types defined in a CaTTS-DL calendar specification $\mathcal{C}$[12]. That $\le$ is a pre-order is stated by the following two rules, i.e. that $\le$ is reflexive (S-Refl) and that $\le$ is transitive (S-Trans):

$$\forall\alpha :: K.\ \alpha \le \alpha \quad \text{(S-Refl)} \qquad\qquad \frac{\forall\alpha,\beta,\gamma :: K.\ \alpha \le \beta \quad \beta \le \gamma}{\alpha \le \gamma} \quad \text{(S-Trans)}$$

Having stated that $\le$ is a pre-order, we make some further general stipulations on subtyping in CaTTS: the subtype relation of CaTTS-CL refers to the composition of the aggregation subtype relation (cf. Definition 3.5) and the inclusion subtype relation (cf. Definition 3.6) as formalized in Definition 3.7. That means in particular, if for example `week` is defined by aggregation from `day` in a CaTTS-DL calendar specification, i.e. `week` $\preceq$ `day`, then `week` $\le$ `day`. Similarly, if for example `working_day` is defined by inclusion from `day` in a CaTTS-DL calendar specification, i.e. `working_day` $\subseteq$ `day`, then `working_day` $\le$ `day`.

Defining subtyping in CaTTS-CL by composition of aggregation and inclusion requires some basic coercion between the kinds of points over calendric types, i.e. σ and the kinds of intervals over calendric types, i.e. σ^*:

$$\sigma \le \sigma^* \quad \text{(S-ICoer)}$$

[12]Recall that $\mathcal{C}$ refers to a finite set of aligned CaTTS-DL calendar specifications.

Meaning, that any calendric type σ (defined in some CaTTS-DL calendar specification) is a subtype of the type of intervals over the same type, i.e. σ^*. This coercion is very natural for calendric types since a point is nothing but an interval of duration 1.[13] Furthermore, this coercion makes explicit what is implicitly hidden in the aggregation relation: inferring that `week` is a subtype of `day` means that if at some point in a CaTTS-CL program a value of type `week` is expected, one might use an interval of days instead, i.e. a value of type `day`*. With the rule (S-ICoer), we can now use intervals whenever points are expected.

Now we can consider the rules for the different syntactic forms to construct aggregation types:

$$(aggregate\ \tau\ te_i^{i\in 1..n}\ \tilde{}@te) \leq \tau \quad \text{(S-Aggr)} \qquad (aggregate\ \tau\ te_i^{i\in 1..n}\ \tilde{}@z) \leq \tau \quad \text{(S-Aggr}\tilde{}\text{)}$$

$$\tau_1\# < \tau_2 \leq \tau_2 \qquad\qquad \text{(S-Res)}$$

Those three rules are mere axioms. The rules (S-Aggr) and (S-Aggr˜) state that a calendric type defined by aggregation is a subtype of the type it is defined from. Similarly, the rule (S-Res) states that a calendric type defined by restriction is a subtype of the type its values are restricted to.

The rules for the different syntactic forms to construct inclusion types are the following.

$$(select\ \tau\ X : \tau\ where\ te) \leq \tau \quad \text{(S-Sel)} \qquad \tau \setminus \sigma \leq \tau \quad \text{(S-Ex)}$$

$$i \in \{1,2\},\ \tau_i \leq \tau_1 \mid \tau_2 \quad \text{(S-Dj}_1\text{)} \qquad \frac{i \in \{1,2\},\ \sigma_i \leq \tau}{\sigma_1 \mid \sigma_2 \leq \tau} \quad \text{(S-Dj}_2\text{)}$$

$$i \in \{1,2\},\ \tau_1\&\tau_2 \leq \tau_i \quad \text{(S-Cj}_1\text{)} \qquad \frac{i \in \{1,2\},\ \sigma \leq \tau_i}{\sigma \leq \tau_1\&\tau_2} \quad \text{(S-Cj}_2\text{)}$$

The axiom (S-Sel) states that a calendric type defined by selection is a subtype of the type it is defined from. CaTTS' inclusion constructor for disjunctions has two different subtyping rules (S-Dj$_1$) and (S-Dj$_2$). The former states that the types τ_1 and τ_2 the type $\tau_1 \mid \tau_2$ is constructed from are both subtypes of $\tau_1 \mid \tau_2$. The latter states that if either σ_1 or σ_2 is a subtype of τ, then the disjunction of σ_1 and σ_2 (denoted $\sigma_1 \mid \sigma_2$) is also a subtype of τ. The two rules for the conjunction constructor (S-Cj$_1$) and (S-Cj$_2$) formulate the conditions that (1) a conjunction $\tau_1\&\tau_2$ is a subtype of the types τ_1 and τ_2 it is constructed from, and that (2) a subtype σ of either τ_1 or τ_2 is also a subtype of the types' conjunction $\tau_1\&\tau_2$. CaTTS' constructor for exception types has a single axiom (S-Ex) stating that the exception $\tau \setminus \sigma$ is a subtype of τ.

Furthermore, we need subtyping statements for the forms to construct kinds of intervals (S-Int) and kinds of durations (S-Dur):

$$\frac{\sigma \leq \tau}{\sigma^n \leq \tau^n} \quad \text{(S-Dur)} \qquad \frac{\sigma \leq \tau}{\sigma^* \leq \tau^*} \quad \text{(S-Int)}$$

[13]Recall that CaTTS is based on an interval-based time model, i.e. durationless time points do not exist.

That means, if σ is a subtype of τ, then duration of σ is also a subtype of duration of τ (denoted $\sigma^n \leq \tau^n$), and then interval of σ is also a subtype of interval of τ (denoted $\sigma^* \leq \tau^*$), respectively.

Note: In CaTTS' current prototype implementation, subtyping is not support with kinds of durations over calendric types, i.e. the rule (S-Dur) is not implemented with the subtype checker. Subtyping with durations is not supported in CaTTS current implementation because it would introduce a form of imprecision into the language. For example, shifting a day d forward by 1 month would result in a set of possible days. Similar kinds of imprecision would appear when comparing for example the duration of 4 weeks to that of 1 month. So far, CaTTS only supports modeling and processing of precise calendric data and constraints. Section 7.2.1.3 provides with a discussion on dealing with imprecision caused by duration subtyping in CaTTS.

Finally, each (finite set of) CaTTS-DL calendar specification(s) $\mathcal{C}$ has a calendric type that is a supertype of any other calendric type defined in $\mathcal{C}$, denoted $r_{\mathcal{C}}$, read as "r is the reference type of the calendar specification $\mathcal{C}$". This type comes with an inference rule that makes $r_{\mathcal{C}}$ a maximum element of the calendric types defined in $\mathcal{C}$ according to the subtype relation $\leq$.

$$\sigma \leq r_{\mathcal{C}} \quad \text{(S-Ref)}$$

Note: The subtype relation (i.e. the composition of calendric type aggregation and inclusion) is the corner stone of CaTTS-CL. This relation defines the central feature of some calendric type being either coarser or finer than some other calendric type. In fact, this relation defines a means to compare *any* pair of calendric types defined in a (finite set of aligned) CaTTS-DL calendar specifications $\mathcal{C}$, thus, reflecting a widespread form of common-sense reasoning with calendric data.

6.7.4 The Typing Relation

This section introduces a *typing relation* for the CaTTS-CL expressions given in Table 6.8, p.198. CaTTS-CL expressions are typed after types defined in some CaTTS-DL calendar specification. The syntax of CaTTS-DL types is given in Table 6.3, p.189. CaTTS-CL's typing relation is defined by a finite collection of inference rules deriving statements of the form $\forall\alpha :: K\ \Gamma \vdash ce : \alpha$, read as "the CaTTS-CL expression ce has type α of kind K in the context Γ". The typing relation considers each form of CaTTS-CL expressions (e.g. shift expressions, interval relations, interval construction) separately. For each syntactic construct one or more rules are defined for sub-expressions, a CaTTS-CL expression like a shift can be constructed from. CaTTS-CL's typing relation is summarized in Table 6.10. The different inference rules are introduced in the following.

$$\frac{\forall \alpha, \beta :: K.\ \ \Gamma \vdash ce : \alpha \quad \alpha \leq \beta}{\Gamma \vdash ce : \beta} \qquad \text{(T-Sub)}$$

$$\Gamma \vdash k : \tau \qquad \text{(T-Const)} \qquad\qquad \frac{\forall \alpha :: K.\ \ X : \alpha \in \Gamma}{\Gamma \vdash X : \alpha} \qquad \text{(T-Var)}$$

$$\frac{\Gamma \vdash n : \mathbb{Z}}{\Gamma \vdash n\ \tau_1 : \tau_1^n} \qquad \text{(T-Dur)} \qquad\qquad \frac{\Gamma \vdash ce_1 : \tau_1}{\Gamma \vdash index\ ce_1 : \mathbb{Z}} \qquad \text{(T-Index)}$$

$$\frac{\Gamma \vdash ce_1 : \tau_1^*}{\Gamma \vdash duration\ ce_1 : \tau_1^n} \qquad \text{(T-Duration)} \qquad\qquad \frac{\Gamma \vdash ce_1 : \tau_1^*}{\Gamma \vdash b/e\ ce_1 : \tau_1} \qquad \text{(T-EndP)}$$

$$\frac{\Gamma \vdash ce_1 : \tau_1 \quad \Gamma \vdash ce_2 : \tau_1}{\Gamma \vdash [ce_1..ce_2] : \tau_1^*} \qquad \text{(T-EndpI)} \qquad\qquad \frac{\Gamma \vdash ce_1 : \tau_1^n \quad \Gamma \vdash ce_2 : \tau_1}{\Gamma \vdash ce_1\ to\ ce_2 : \tau_1^*} \qquad \text{(T-DurI)}$$

$$\frac{\Gamma \vdash ce_1 : \tau_1^* \quad \Gamma \vdash ce_2 : \tau_1^n}{\Gamma \vdash shift\ ce_1\ f/b\ ce_2 : \tau_1^*} \qquad \text{(T-Shift)} \qquad\qquad \frac{\Gamma \vdash ce_1 : \tau_1^* \quad \Gamma \vdash ce_2 : \tau_1^n}{\Gamma \vdash e/s\ ce_1\ by\ ce_2 : \tau_1^*} \qquad \text{(T-ExSh)}$$

$$true : \mathbb{B} \qquad \text{(T-True)} \qquad\qquad false : \mathbb{B} \qquad \text{(T-False)}$$

$$\frac{\Gamma, X : \tau \vdash ce : \mathbb{B}}{\Gamma \vdash (X\ is\ 1\ \tau)\ \&\&\ ce : \mathbb{B}} \qquad \text{(T-Event)} \qquad\qquad \frac{\Gamma, X : \tau^* \vdash ce : \mathbb{B}}{\Gamma \vdash (X\ is\ <n>\ \tau)\ \&\&\ ce : \mathbb{B}} \qquad \text{(T-Task)}$$

$$\frac{\Gamma \vdash ce_1 : \tau_1^* \quad \Gamma \vdash ce_2 : \tau_1^*}{\Gamma \vdash ce_1\ \mathcal{R}\ ce_2 : \mathbb{B}} \qquad \text{(T-Interval)} \qquad\qquad \frac{\Gamma \vdash ce_1 : \tau_1^n \quad \Gamma \vdash ce_2 : \tau_1^n}{\Gamma \vdash ce_1\ \odot\ ce_2 : \mathbb{B}} \qquad \text{(T-Metric)}$$

$$\frac{\Gamma \vdash ce_1 : \mathbb{Z} \quad \Gamma \vdash i : \mathbb{Z} \quad \tau_{2_{type}}}{\begin{array}{c}\Gamma \vdash relative\ ce_1\ in/to\ \tau_2 \odot i : \mathbb{B}\ / \\ \Gamma \vdash m(relative\ ce_1\ in/to\ \tau_2) : \mathbb{B}\end{array}} \qquad \text{(T-Rel)} \qquad\qquad \frac{\Gamma \vdash ce_1 : \mathbb{B} \quad \Gamma \vdash ce_2 : \mathbb{B}}{\Gamma \vdash ce_1\ \&\&\ ce_2 : \mathbb{B}} \qquad \text{(T-Conj)}$$

Table 6.10: The typing relation of CaTTS-CL.

Initially, the intuition that values of a "finer" calendric type (i.e. a supertype) can be used instead of values of some corresponding "coarser" calendric type (i.e. a subtype) has been already formulated in Section 6.7.2 with the rule of subsumption (T-SUB). This inference rule defines a connection between subtyping in CaTTS-CL and the typing rules for the different syntactic forms of CaTTS-CL expressions.

Let us now consider the further rules given in Table 6.10. Those rules enforce some simple syntactic restrictions on CaTTS-CL expressions. The rules does not include any particularities. But note that the rules are carefully defined whether the expressions refer to kind of points, intervals, or durations over some calendric type τ. Further note that whenever a rule requires an expression of type interval of τ, i.e. τ^*, an expression of type τ may also be used due to the subtype rule for interval coercion (S-ICOER), inferring $\tau \leq \tau^*$.

In the following, the meaning of the different typing rules in the typing relation is addressed.

According to the rule (T-VAR), a CaTTS-CL variable has whatever type one might currently assume it to have. The premise $\forall\alpha :: K.\ \mathtt{X}{:}\alpha \in \Gamma$ is read: " for every kind of calendric type, the type assumed for `X` in the context Γ is α". The axiom (T-CONST) states that a (date) constant has any calendric type (according to its format definition).

A duration `n` τ has type τ^n, if `n` is a natural number according to rule (T-DUR).

Three rules to extract duration, end points, or the index from activities are defined: the duration `duration` $\mathtt{ce}_1$ of an expression $\mathtt{ce}_1$ of type τ_1^* has type τ_1^n according to rule (T-DURATION). The index `index` $\mathtt{ce}_1$ of an expression $\mathtt{ce}_1$ of type τ_1 has type $\mathbb{Z}$ according to rule (T-INDEX). The begin (resp. end) point `b/e` $\mathtt{ce}_1$ of an expression $\mathtt{ce}_1$ of type τ_1^* has type τ_1 according to rule (T-ENDP).

Two different forms of task constructions are supported. (T-ENDPI) allows for constructing tasks from ending points ce_1 and ce_2 of some type τ_1. Such tasks have the form $[\mathtt{ce}_1..\mathtt{ce}_2]$ of type interval of τ_1, τ_1^*. The rule (T-DURI) permits the construction of a tasks from a duration ce_1 of some type τ_1, i.e. τ_1^n and an event ce_2 over the same calendric type, however of kind point, i.e. τ_1. Such tasks has the form $\mathtt{ce}_1$ `to` $\mathtt{ce}_2$ of type τ_1^* (i.e. interval of τ_1).

The rule for shifting activities (T-SHIFT) formulates the condition that the expression `shift` $\mathtt{ce}_1$ `f/b` $\mathtt{ce}_2$ has type τ_1^*, if $\mathtt{ce}_1$ is of type τ_1^* and $\mathtt{ce}_2$ of type duration of τ_1, τ_1^n.

The rule for extending or shortening an activity (T-EXSH) formulates the condition that the expression `e/s` $\mathtt{ce}_1$ `by` $\mathtt{ce}_2$ has type τ_1^*, if $\mathtt{ce}_1$ is of type τ_1^* and $\mathtt{ce}_2$ of type duration of τ_1, τ_1^n.

Calendar domain constraints to specify activities (in particular events and tasks) are constructed using CaTTS-CL's `is` constraint followed by a duration. The typing rules for such domain constraints, (T-EVENT) and (T-TASK), state the following: the intended type of the event (resp. task) is given by its associated duration, written `X is n` τ where `n` must be a natural number greater than zero (as stated in the premise of the rules). The associated duration tells the type checker to assume that the variable `X` is of type τ (resp. τ^* if it is a task, i.e. $n > 1$). Knowing the type of the variable which is τ (resp. τ^*), the type of the time constraint ce the calendar domain expression is associated with is boolean (denoted $\mathbb{B}$).

Time constraints over activities according to rule (T-INTERVAL) as well as time constraints over durations of activities according to rule (T-METRIC) are of type boolean, if their participating expressions $\texttt{ce}_1$ and $\texttt{ce}_2$ have the same type τ_1^* of intervals (resp. τ_1^n of durations).

The rule (T-REL) for extracting relative indices of some activity's values formulates the condition that the language construct `relative` $\texttt{ce}_1$ `in/to` $\tau_2 \odot$ `i` (resp. `min/max(relative` $\texttt{ce}_1$ `in/to` τ_2`))` has type boolean (denoted $\mathbb{B}$), if expressions $\texttt{ce}_1$ and `i` are integers and τ_2 must be a well-formed calendric type defined in the underlying CaTTS-DL calendar specification.

Finally, the conjunction of constraints of type boolean has again type boolean according to rule (T-CONJ).

6.7.5 Example: Type Checking a CaTTS-CL Program

In the following, the typing derivation for the CaTTS-CL program

```
X is 2 working_week && Y is 1 month && X during Y
```

is given. Assume that the used calendric types are specified in a CaTTS-DL calendar specification as follows:

```
...
type day;
type week = aggregate 7 day @ day(1);
type month = aggregate
      31 day,
      alternate month(i)
        | (i div 12) mod 4 == 0 &&
          ((i div 12) mod 400 != 100 ||
           (i div 12) mod 400 == 0)     -> 29 day
        | otherwise                     -> 28 day
      end,
      ...
      31 day @ day(1);
type working_day = select day(i) where
          relative i in week >= 1 && relative i in week <= 5;
type working_week = week #< working_day;
...
```

The typing derivation is given by a derivation tree applying some of the previously introduced typing and subtyping rules according to Table 6.9, p.200 and Table 6.10, p.203. The tree has to be read from bottom to top.

$$\dfrac{\dfrac{Y : month \in \{X : w\_week^*, Y : month\}}{\{X : w\_week^*, Y : month\} \vdash Y : month}\,(\text{T-}\textsc{Var}) \qquad \overset{**}{month \leq day^*}}{\uparrow}\,(\text{T-}\textsc{Sub})$$

$$\dfrac{\dfrac{X : w\_week^* \in \{X : w\_week^*, Y : month\}}{\{X : w\_week^*, Y : month\} \vdash X : w\_week^*}\,(\text{T-}\textsc{Var}) \qquad \overset{*}{w\_week^* \leq day^*}}{\uparrow}\,(\text{T-}\textsc{Sub})$$

$$\dfrac{\dfrac{\dfrac{\{X : w\_week^*, Y : month\} \vdash X : day^* \qquad \{X : w\_week^*, Y : month\} \vdash Y : day^*}{\{X : w\_week^*, Y : month\} \vdash X\ during\ Y : \mathbb{B}}\,(\text{CT-}\textsc{Interval})}{\{X : w\_week^*\} \vdash Y\ is\ 1\ month\ \ \&\&\ \ X\ during\ Y : \mathbb{B}}\,(\text{T-}\textsc{Event})}{\{\} \vdash X\ is\ 2\ w\_week\ \ \&\&\ \ Y\ is\ 1\ month\ \ \&\&\ \ X\ during\ Y : \mathbb{B}}\,(\text{T-}\textsc{Task})$$

where at the derivation step denoted with $*$, CaTTS-CL's subtype checker is called. The subtype checker computes the following derivation tree.

$$\dfrac{\dfrac{\dfrac{}{w\_week \leq w\_day}\,(\text{S-}\textsc{Res})}{w\_week^* \leq w\_day^*}\,(\text{S-}\textsc{Int}) \qquad \dfrac{\dfrac{}{w\_day \leq day}\,(\text{S-}\textsc{Sel})}{w\_day^* \leq day^*}\,(\text{S-}\textsc{Int})}{w\_week^* \leq day^*}\,(\text{S-}\textsc{Trans})$$

as well as at the derivation step denoted with $**$. At this step, the following derivation tree is computed by the subtype checker.

$$\dfrac{\dfrac{}{month \leq day}\,(\text{S-}\textsc{Aggr}) \qquad \dfrac{}{day \leq day^*}\,(\text{S-}\textsc{ICoer})}{month \leq day^*}\,(\text{S-}\textsc{Trans})$$

6.7.6 Consistency Checks based on Calendric Types

So far, some very basic type checking rules testing for correctness of CaTTS-CL programs are defined. Those rules does not consider possible differences inherent to events (i.e. points in time) and tasks (i.e. intervals in time). Reconsidering the typing rule for interval relations between activities:

$$\dfrac{\Gamma \vdash ce_1 : \tau_1^* \;\; \Gamma \vdash ce_2 : \tau_1^*}{\Gamma \vdash ce_1\, \mathcal{R}\, ce_2 : \mathbb{B}} \quad (\text{T-}\textsc{Interval})$$

This rule merely tells the type checker that a time constraint like `X equals Y` is well-typed, if he can infer a common type τ_1^* for the expressions `X` and `Y`. Assume that `X` is declared by the constraint `X is 1 day`, i.e. `X` has type `day` and `Y` is declared by the constraint `Y is 1 week`, i.e. `Y` has type `week`. According to the typing rules in Table 6.10, p.203 and the subtyping rules in Table 6.9, p.200, the type checker infers that `X equals Y` of type $\mathbb{B}$ is well-typed as follows:

$\Gamma := \{X : day, Y : week\}$

$$\dfrac{\dfrac{\dfrac{X : day \in \Gamma}{\Gamma \vdash X : day}\,(\text{T-}\textsc{Var}) \quad \overset{*}{day \leq day^*}}{\Gamma \vdash X : day^*}\,(\text{T-}\textsc{Sub}) \quad \dfrac{\dfrac{X : week \in \Gamma}{\Gamma \vdash Y : week}\,(\text{T-}\textsc{Var}) \quad \overset{**}{week \leq day^*}}{\Gamma \vdash Y : day^*}\,(\text{T-}\textsc{Sub})}{\Gamma \vdash X\ equals\ Y : \mathbb{B}}\,(\text{T-}\textsc{Interval})$$

where

$$* \qquad \dfrac{}{day \leq day^*}\ (\text{S-ICoer}) \qquad ** \qquad \dfrac{\dfrac{}{week \leq day}\ (\text{S-Aggr}) \qquad \dfrac{}{day \leq day^*}\ (\text{S-ICoer})}{week \leq day^*}\ (\text{S-Trans})$$

assuming that $week := aggregate\ day\ \ 7\ day\ @\ day(1)$.

From the stand point of type checking used to test whether a CaTTS-CL program is syntactically correct, this typing derivation for the CaTTS-CL time constraint `X equals Y` where `X:day` and `Y:week` is perfectly well. But from the stand point of CaTTS-CL which is a constraint language to express multi-calendar appointment scheduling problems, this CaTTS-CL time constraint is obviously erroneous or undesirable in some way: the constraint is inherently *inconsistent* (which would be inferred by CaTTS's constraint solver), because a day `X` can never equal a week `Y` (in the common-sense definition of days and weeks according to the Gregorian calendar).

Carefully considering the typing derivation for the constraint `X equals Y, X:day, Y:week`, we make the following observation: while `X` can be represented in kind point of `day`, `Y` must be represented in kind interval of `day`. That is, if we can infer a common calendric type τ_1 for the expressions ce_1 and ce_2 in the rule (T-Interval) (whenever $\mathcal{R}$ refers to the time constraint `equals`) where one of those expressions can be represented in τ_1 and the other must be represented in type τ_1^*, then the constraint ce_1 `equals` ce_2 is inconsistent. To test such kinds of consistency statically during type checking a CaTTS-CL program, an additional rule for the time constraint `equals` that has the supplement to fail need to be introduced:

$$\dfrac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_1^*}{\Gamma \vdash e_1\ equals\ e_2 : \mathbb{B} \quad | \ fail} \quad (\text{CT-EqualFail})$$

Using this rule instead of the rule (T-Interval) in the typing derivation given above, the CaTTS-CL constraint `X equals Y, X:day, Y:week` is rejected by the type checker. That means in particular, that we can check consistency of CaTTS-CL constraints (at least in some cases) statically, before running CaTTS' constraint solver, making reasoning with time constraints in CaTTS more efficient. In the following, some further rules for specific CaTTS-CL time constraints that check for consistency of CaTTS-CL programs statically during type checking are given. Those rules are similar to the previously discussed rule (CT-EqualFail) for the time constraint `equals`.

$$\dfrac{\Gamma \vdash e_1 : \tau_1^* \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ \mathcal{I}\ e_2 : \mathbb{B} \quad | \ fail} \quad (\text{CT-IntFail}) \qquad \dfrac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_1^*}{\Gamma \vdash e_1\ \mathcal{I}'\ e_2 : \mathbb{B} \quad | \ fail} \quad (\text{CT-Int'Fail})$$

$$\dfrac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ overlaps\ e_2 : \mathbb{B} \quad | \ fail} \quad (\text{CT-OverlapsFail})$$

where $\mathcal{I} \in \{starts, finishes, during, within\}$ and $\mathcal{I}' \in \{started\_by, finished\_by, cotains, \}$.

Note: Such consistency checks of CaTTS-CL programs are currently not implemented with CaTTS' type checker. In CaTTS' prototype implementation, any form of consistency checks are left to CaTTS' constraint solver, and are thus performed dynamically at runtime.

6.8 Coercion Semantics for Subtyping in CaTTS-CL

The subtyping mechanism introduced for CaTTS so far is just a way to obtain additional flexibility in typing CaTTS-CL expressions. In particular, the user is not obliged to annotate expressions with additional type cast informations in a program that refers to calendric data and constraints typed after calendric types defined in some CaTTS-DL calendar specification like

```
X is 2 working_week && Y is 1 month && X during Y
```

That is, the CaTTS-CL constraint `X during Y` can be specified without explicit type cast declarations although `X` and `Y` have different types.

This flexibility in CaTTS-CL programs is achieved by subsumption in terms of the rule (T-Sub). Recall that the subsumption rule merely tells the type checker that some set of values represented by one type are a subset of the set of values represented by some other (super)type such that the supertype also types the subset. This interpretation of the subsumption rule for CaTTS-CL expressions is farely simple and natural when talking about time: according to the CaTTS-CL program given above, CaTTS-CL's type checker infers (by applying the rules given in Table 6.10, p.203) the subtyping statements that $\texttt{working\_week}^*$, i.e. the type of `X` is a subtype of intervals of days, $\texttt{day}^*$ (denoted $\texttt{working\_week}^* \leq \texttt{day}^*$) and that `month`, i.e. the type of `Y` is a subtype of $\texttt{day}^*$ (denoted $\texttt{month} \leq \texttt{day}^*$), as well. Those two statements are then tested by CaTTS-CL's subtype checker that applies the rules given in Table 6.9, p.200. The subtype checker is called by the type checker whenever a subtyping statement is inferred within a typing derivation. Thus, a CaTTS-CL user may specify the time constraints mentioned above in a CaTTS-CL program without hesitation about the different calendric types used.

From the subtyping statements $\texttt{working\_week}^* \leq \texttt{day}^*$ and $\texttt{month} \leq \texttt{day}^*$ the type checker knows *that* $\texttt{working\_week}^*$ and `month` are subtypes of $\texttt{day}^*$, but not *why* this is the case. However, to efficiently solve the above specified CaTTS-CL program using CaTTS' constraint solver, the constraint solver needs to know why some type is a subtype of another type such that values of the one type can be represented in terms of values of another type. Recall that this representation of equivalent sets of values of different types is required to perform multi-calendar constraint reasoning in CaTTS. In particular, this equivalent representation of calendric values in different types is the basis of CaTTS' conversion constraints, making multi-calendar constraint solving both reasonable and efficient. For example, the month `month(1)` is represented by the calendar domain $1..1, month$ in terms of type `month`, whereas it is represented by the calendar domain $1..1 + 31..31, day$ in terms of type `day`*, assuming that the month with index 1 refers to a January in the Gregorian calendar.

To relate these two different representations for the same amount of time to each other and to translate one representation into the other whenever required by the constraint solver, CaTTS' conversion constraint (cf. Section 5.4.3) is used. To reconcile the conversion constraint (without burdening the programmer with conversions between different representations of the variables' associated domains) with the subset semantics of subtyping in CaTTS-CL, each subtyping statement inferred during type checking some CaTTS-CL program is translated into a conversion constraint that can be obtained by CaTTS' constraint solver. Such a translation is achieved by adapting a *coercion semantics for subtyping* in CaTTS-CL.

This coercion semantics is introduced in the following. Subsequently, a transformation of CaTTS-CL programs into CL$_{catts}$ programs (cf. Table 5.1, p.136 to recall the syntax of CL$_{catts}$) that can be solved by CaTTS' constraint solver is defined. This transformation is based on the coercion semantics. Finally, a translation of a CaTTS-CL program into an equivalent CL$_{catts}$ program is exemplified. What is meant by "equivalence" is formalized in terms of *coherence* of the transformation.

6.8.1 Coercion Semantics

To formally relate CL$_{catts}$'s conversion constraint with subtyping in CaTTS-CL, and thus to evaluate CaTTS-CL programs using CaTTS' multi-calendar constraint solver, a different semantics for subtyping is adopted. This semantics is called *coercion semantics* [BTCGS91, Mit96, Pie02]. Coercion semantics replaces the subset semantics of subtyping introduced with the rule of subsumption (T-SUB).

In principle, a coercion semantics "compiles away" subtyping by replacing it with run-time coercion. For example, if in the CaTTS-CL program

```
X is 2 working_week && Y is 1 month && X during Y
```

the interval of working weeks `X` is promoted to an interval of days and the month `Y` is promoted to an interval of days during type checking this CaTTS-CL program, i.e. type checking infers the subtyping statements `working_week`$^* \leq$ `day`* and `month` $\leq$ `day`*, then (at run time) CaTTS' processor transforms this time constraint into an equivalent time constraint `X' during Y'` where `X'` and `Y'` both have type `day`* and conversion constraints `X` $\simeq$ `X'` and `Y` $\simeq$ `Y'`. The conversion constraints then obtain the different representations for `X` and `Y` in terms of intervals of working weeks and intervals of days (resp. months and intervals of days) during constraint solving using CaTTS' constraint solver.

CaTTS-CL's coercion semantics for subtyping is expressed by a function that transforms CaTTS-CL expressions (cf. Table 6.8, p.198 for the syntactic forms of CaTTS-CL) into CL$_{catts}$ expressions (cf. Table 5.1, p.136 for the syntactic forms of CL$_{catts}$), the language used by CaTTS' multi-calendar constraint solver. CL$_{catts}$ essentially is a language without subtyping that, instead, contains the conversion constraint. In accordance with this translation function, no evaluation rules for the language CaTTS-CL are defined. Rather, CaTTS-CL expressions are evaluated by type checking them using the afore introduced typing and subtyping relations (cf. Table 6.10, p.203 and Table 6.9, p.200). Subsequently, their typing derivations are translated into the language CL$_{catts}$, and then the evaluation relation of this language (i.e. CaTTS' constraint propagation algorithm, cf. Section 5.5) is used to obtain the operational behavior of these CaTTS-CL expressions.

Formally, the coercion semantics, i.e. the compilation from CaTTS-CL expressions to CL$_{catts}$ expressions consists of three functions, one for calendric types, one for subtyping, and one for typing. The transformation function for CaTTS-DL types and CaTTS-CL expressions is called *translation*, and the function for subtyping in CaTTS-CL is called *coercion*.

Transformation of Calendric Types. Calendric types (cf. Table 6.3 for the syntactic forms of CaTTS-DL types and kinds of types) remain unaffected by the translation. The translation function is written $trans(\cdot)$.

$$\begin{array}{lcl}
trans(r_{\mathcal{C}}) & = & r_{\mathcal{C}} \\
trans(aggregate\ \tau\ te_i^{i\in 1..n}@te) & = & aggregate\ \tau\ te_i^{i\in 1..n}@te \\
trans(aggregate\ \tau\ te_i^{i\in 1..n\,\tilde{}}@z) & = & aggregate\ \tau\ te_i^{i\in 1..n\,\tilde{}}@z \\
trans(select\ \tau\ X : \tau\ where\ te) & = & select\ \tau\ X : \tau\ where\ te \\
trans(\tau_1 \& \tau_2) & = & trans(\tau_1)\ \&\ trans(\tau_2) \\
trans(\tau_1 \mid \tau_2) & = & trans(\tau_1) \mid trans(\tau_2) \\
trans(\tau_1 \setminus \tau_2) & = & trans(\tau_1) \setminus trans(\tau_2) \\
trans(\tau_1 \# < \tau_2) & = & trans(\tau_1) \# < trans(\tau_2) \\
trans(\tau^*) & = & trans(\tau)^* \\
trans(\tau^n) & = & trans(\tau)^n
\end{array}$$

Coercion of the Subtype Relation. For subtyping, the translation generates a *coercion* from one calendric type to another calendric type. To generate such a coercion, and, thus, to transform values of some type σ to values of some other type τ where $\sigma \leq \tau$, CaTTS-CL's language processor needs to know (i) *that* σ is a subtype of τ and (ii) *why* σ is a subtype of τ. The first condition is obtained by inferring a subtyping derivation for the subtyping statement $\sigma \leq \tau$ using CaTTS-CL's subtyping relation given in Table 6.9, p.200. The second condition is accomplished by generating coercions from such a subtyping derivation. Such a coercion is expressed in terms of a CL$_{catts}$ conversion constraint. The following notation is used to formalize this coercion: write $C :: \sigma \leq \tau$ to mean "C is the subtyping derivation tree whose conclusion is $\sigma \leq \tau$". Such a derivation generates a coercion $c_\sigma^\tau(\cdot)$. The coercion is then defined as follows.

Definition 6.3 (Coercion). *Let C a derivation for the subtyping statement $\sigma \leq \tau$ derived using CaTTS-CL's subtyping relation (cf. Table 6.9, p.200). The translation generates a* **coercion** *c_σ^τ, i.e. a function from type σ to type τ in the language CL$_{catts}$. This coercion is defined by case on the final rule used in the derivation C as follows.*

(S-Refl)	$\forall\alpha :: K\ \ c_{\alpha}^{\alpha}$	$\overset{def}{=}$	$X : \alpha, Y : \alpha,\ X \simeq Y$
(S-Trans)	$\forall\alpha,\beta,\gamma :: K\ c_{\alpha}^{\gamma}$	$\overset{def}{=}$	$X : \alpha,\ c_{\beta}^{\gamma}(c_{\alpha}^{\beta}(X))$
(S-ICoer)	$c_{\sigma}^{\sigma^*}$	$\overset{def}{=}$	$X : \sigma, Y : \sigma^*,\ X \simeq Y$
(S-Ref)	c_{σ}^{re}	$\overset{def}{=}$	$X : \sigma, Y : \mathbb{Z}, X \simeq Y$
(S-Dur)	$c_{\sigma^n}^{\tau^n}$	$\overset{def}{=}$	$X : \sigma^n,\ (c_{\sigma}^{\tau})^n(X)$
(S-Int)	$c_{\sigma^*}^{\tau^*}$	$\overset{def}{=}$	$X : \sigma^*,\ (c_{\sigma}^{\tau})^*(X)$
(S-Sel)	$c_{(select\ \tau\ X:\tau\ where\ te)}^{\tau}$	$\overset{def}{=}$	$X : (select\ \tau\ X : \tau\ where\ te), Y : \tau,\ X \simeq Y$
(S-Dj$_1$)	$c_{\tau_i}^{\tau_1 \mid \tau_2}$	$\overset{def}{=}$	$X : \tau_i, Y : \tau_1 \mid \tau_2,\ X \simeq Y$
(S-Dj$_2$)	$c_{\sigma_1 \mid \sigma_2}^{\tau}$	$\overset{def}{=}$	$X : \sigma_1 \mid \sigma_2,\ c_{\sigma_1}^{\tau}(X) \mid c_{\sigma_2}^{\tau}(X)$
(S-Cj$_1$)	$c_{\tau_1 \& \tau_2}^{\tau_i}$	$\overset{def}{=}$	$X : \tau_1 \& \tau_2, Y : \tau_i,\ X \simeq Y$
(S-Cj$_2$)	$c_{\sigma}^{\tau_1 \& \tau_2}$	$\overset{def}{=}$	$X : \sigma,\ c_{\sigma}^{\tau_1}(X) \& c_{\sigma}^{\tau_2}(X)$
(S-Ex)	$c_{\tau \setminus \sigma}^{\tau}$	$\overset{def}{=}$	$X : \tau \setminus \sigma, Y : \tau,\ X \simeq Y$
(S-Aggr)	$c_{(aggregate\ \tau\ te_i^{i \in 1..n} @ te)}^{\tau}$	$\overset{def}{=}$	$X : (aggregate\ \tau\ te_i^{i \in 1..n} @ te), Y : \tau,\ X \simeq Y$
(S-Aggr˜)	$c_{(aggregate\ \tau\ te_i^{i \in 1..n} \sim @ z)}^{\tau}$	$\overset{def}{=}$	$X : (aggregate\ \tau\ te_i^{i \in 1..n} \sim @z), Y : \tau,\ X \simeq Y$
(S-Res)	$c_{\sigma\# < \tau}^{\tau}$	$\overset{def}{=}$	$X : \sigma\# < \tau, Y : \tau,\ X \simeq Y$

Transformation of the Typing Relation. For typing, the translation transforms CaTTS-CL expressions into CL$_{catts}$ expressions. This translation refers to the previously defined coercion. To translate a CaTTS-CL expression into a CL$_{catts}$ expression without subtyping, CaTTS-CL's language processor needs to know where the subsumption rule (T-Sub) is used in type checking a CaTTS-CL expression. The translation then inserts (run-time) coercion (i.e. it applies the coercion according to Definition 6.3) whenever this subsumption rule is applied during type checking a CaTTS-CL expression. A convenient way to formalize this observation is to define the translation form CaTTS-CL expressions to CL$_{catts}$ expressions as a function on derivations of typing statements inferred during type checking a CaTTS-CL expression using the typing relation given in Table 6.10, p.203. The following notation is used to formalize this translation: write $D :: \forall\alpha.\ \alpha :: K.\ \Gamma \vdash ce : \alpha$ to mean "D is the typing derivation tree whose conclusion is $\forall\alpha.\ \alpha :: K.\ \Gamma \vdash ce : \alpha$". Such a derivation generates a translation $trans(\forall\alpha.\ \alpha :: K.\ \Gamma \vdash ce : \alpha)$ into a CL$_{catts}$ expression ce of type α in CL$_{catts}$. The translation is then defined as follows.

Definition 6.4 (Translation Function). *Let D a derivation for the typing statement $\forall\alpha.\ \alpha :: K\ \Gamma \vdash ce : \alpha$ derived using CaTTS-CL's typing relation (cf. Table 6.10, p.203). Then its translation $trans(\forall\alpha.\ \alpha :: K\ \Gamma \vdash ce : \alpha)$ is a CL$_{catts}$ expression of type α in CL$_{catts}$. The translation is defined by case on the final rule used in the derivation D as follows.*

(T-Sub)	$\mathit{if}\ \forall\alpha,\beta :: K\ \Gamma \vdash ce : \beta$ *is derived from* $\Gamma \vdash ce : \alpha$ *using* $\alpha \leq \beta$, *then* $\mathit{trans}(\Gamma \vdash ce : \beta) = c_{\alpha}^{\beta}(\mathit{trans}(\Gamma \vdash ce : \alpha))$
(T-Var)	$\mathit{if}\ \forall\alpha :: K\ \Gamma \vdash X : \alpha)$ *is derived from* $X : \alpha \in \Gamma$, *then* $\mathit{trans}(\forall\alpha :: K\ \Gamma \vdash X : \alpha) = X$
(T-Const)	$\mathit{trans}(\Gamma \vdash k : \tau) = k$
(T-Dur)	$\mathit{if}\ \Gamma \vdash n\ \tau_1 : \tau_1^n$ *is derived from* $\Gamma \vdash n : \mathbb{N}$, *then* $\mathit{trans}(\Gamma \vdash n\ \tau_1 : \tau_1^n) = n\ \tau_1$
(T-Duration)	$\mathit{if}\ \Gamma \vdash \mathit{duration}\ ce_1 : \tau_1^n$ *is derived from* $\Gamma \vdash ce_1 : \tau_1^*$, *then* $\mathit{trans}(\Gamma \vdash \mathit{duration}\ ce_1 : \tau_1^n) = \mathit{duration}\ \mathit{trans}(\Gamma \vdash ce_1 : \tau_1^*) = X$, X *fresh*
(T-EndP)	$\mathit{if}\ \Gamma \vdash b/e\ ce_1 : \tau_1$ *is derived from* $\Gamma \vdash ce_1 : \tau_1^*$, *then* $\mathit{trans}(\Gamma \vdash b/e\ ce_1 : \tau_1) = b/e\ \mathit{trans}(\Gamma \vdash ce_1 : \tau_1^*) = X$, X *fresh*
(T-Index)	$\mathit{if}\ \Gamma \vdash \mathit{index}\ ce_1 : \mathbb{Z}$ *is derived from* $\Gamma \vdash ce_1 : \tau_1$, *then* $\mathit{trans}(\Gamma \vdash \mathit{index}\ ce_1 : \mathbb{Z}) = \mathit{index}\ \mathit{trans}(\Gamma \vdash ce_1 : \tau_1) = X$, X *fresh*
(T-EndpI)	$\mathit{if}\ \Gamma \vdash [ce_1..ce_2] : \tau_1^*$ *is derived from* $\Gamma \vdash ce_1 : \tau_1$ *and* $\Gamma \vdash ce_2 : \tau_1$, *then* $\mathit{trans}(\Gamma \vdash [ce_1..ce_2] : \tau_1^*) = [\mathit{trans}(\Gamma \vdash ce_1 : \tau_1)..\mathit{trans}(\Gamma \vdash ce_2 : \tau_1)]$
(T-DurI)	$\mathit{if}\ \Gamma \vdash ce_1\ \mathit{to}\ ce_2 : \tau_1^*$ *is derived from* $\Gamma \vdash ce_1 : \tau_1^n$ *and* $\Gamma \vdash ce_2 : \tau_1$, *then* $\mathit{trans}(\Gamma \vdash ce_1\ \mathit{to}\ ce_2 : \tau_1^*) = \mathit{trans}(\Gamma \vdash ce_1 : \tau_1^n)\ \mathit{to}\ \mathit{trans}(\Gamma \vdash ce_2 : \tau_1)$
(T-Shift)	$\mathit{if}\ \Gamma \vdash \mathit{shift}\ ce_1\ f/b\ ce_2\ \tau_1^*$ *is derived from* $\Gamma \vdash ce_1 : \tau_1^*$ *and* $\Gamma \vdash ce_2 : \tau_1^n$, *then* $\mathit{trans}(\Gamma \vdash \mathit{shift}\ ce_1\ f/b\ ce_2 : \tau_1^*) =$ $\mathit{shift}\ \mathit{trans}(\Gamma \vdash ce_1 : \tau_1^*)\ f/b\ \mathit{trans}(\Gamma \vdash ce_2 : \tau_1^n) = X$, X *fresh*
(T-ExSh)	$\mathit{if}\ \Gamma \vdash e/s\ ce_1\ \mathit{by}\ ce_2 : \tau_1^*$ *is derived from* $\Gamma \vdash ce_1 : \tau_1^*$ *and* $\Gamma \vdash ce_2 : \tau_1^n$, *then* $\mathit{trans}(\Gamma \vdash e/s\ ce_1\ \mathit{by}\ ce_2 : \tau_1^*) =$ $e/s\ \mathit{trans}(\Gamma \vdash ce_1 : \tau_1^*)\ \mathit{by}\ \mathit{trans}(\Gamma \vdash ce_2 : \tau_1^n) = X$, X *fresh*
(T-Rel)	$\mathit{if}\ \Gamma \vdash \mathit{relative}\ ce_1\ \mathit{in/to}\ \tau_2 \odot i : \mathbb{B}\ (\Gamma \vdash \mathit{min/max}(\mathit{relative}\ ce_1\ \mathit{in/to}\ \tau_2) : \mathbb{B})$ *is derived from* $\Gamma \vdash ce_1 : \mathbb{Z}$, $\Gamma \vdash i : \mathbb{Z}$, *and* τ_2, *then* $\mathit{trans}(\Gamma \vdash \mathit{relative}\ ce_1\ \mathit{in/to}\ \tau_2 \odot i : \mathbb{B}) = \mathit{relative}\ \mathit{trans}(\Gamma \vdash ce_1 : \mathbb{Z})\ \mathit{in/to}\ \tau_2 \odot i$ $(\mathit{trans}(\Gamma \vdash m(\mathit{relative}\ ce_1\ \mathit{in/to}\ \tau_2) : \mathbb{B}) = m(\mathit{relative}\ \mathit{trans}(\Gamma \vdash ce_1 : \mathbb{Z})\ \mathit{in/to}\ \tau_2))$
(T-True)	$\mathit{trans}(\Gamma \vdash \mathit{true} : \mathbb{B}) = \mathit{true}$
(T-False)	$\mathit{trans}(\Gamma \vdash \mathit{false} : \mathbb{B}) = \mathit{false}$
(T-Event)	$\mathit{if}\ \Gamma \vdash X\ \mathit{is}\ 1\ \tau\ \&\&\ ce : \mathbb{B}$ *is is derived from* $\Gamma, X : \tau \vdash ce : \mathbb{B}$, *then* $\mathit{trans}(\Gamma \vdash X\ \mathit{is}\ 1\ \tau\ \&\&\ ce : \mathbb{B}) = \mathit{trans}(\Gamma, X : \tau \vdash ce : \mathbb{B})$
(T-Task)	$\mathit{if}\ \Gamma \vdash X\ \mathit{is}\ \langle n \rangle\ \tau\ \&\&\ ce : \mathbb{B}$ *is is derived from* $\Gamma, X : \tau^* \vdash ce : \mathbb{B}$, *then* $\mathit{trans}(\Gamma \vdash X\ \mathit{is}\ \langle n \rangle\ \tau\ \&\&\ ce : \mathbb{B}) = \mathit{trans}(\Gamma, X : \tau^* \vdash ce : \mathbb{B})$
(T-Interval)	$\mathit{if}\ \Gamma \vdash ce_1\ \mathcal{R}\ ce_2 : \mathbb{B}$ *is derived from* $\Gamma \vdash ce_1 : \tau_1^*$ *and* $\Gamma \vdash ce_2 : \tau_1^*$, *then* $\mathit{trans}(\Gamma \vdash ce_1\ \mathcal{R}\ ce_2 : \mathbb{B}) = \mathit{trans}(\Gamma \vdash ce_1 : \tau_1^*)\ \mathcal{R}\ \mathit{trans}(\Gamma \vdash ce_2 : \tau_1^*)$
(T-Metric)	$\mathit{if}\ \Gamma \vdash ce_1\ \odot\ ce_2 : \mathbb{B}$ *is derived from* $\Gamma \vdash ce_1 : \tau_1^n$ *and* $\Gamma \vdash ce_2 : \tau_1^n$, *then* $\mathit{trans}(\Gamma \vdash ce_1\ \odot\ ce_2 : \mathbb{B}) = \mathit{trans}(\Gamma \vdash ce_1 : \tau_1^n)\ \odot\ \mathit{trans}(\Gamma \vdash ce_2 : \tau_1^n)$
(T-Conj)	$\mathit{if}\ \Gamma \vdash ce_1\ \&\&\ ce_2 : \mathbb{B}$ *is derived from* $\Gamma \vdash ce_1 : \mathbb{B}$ *and* $\Gamma \vdash ce_2 : \mathbb{B}$, *then* $\mathit{trans}(\Gamma \vdash ce_1\ \&\&\ ce_2 : \mathbb{B}) = \mathit{trans}(\Gamma \vdash ce_1 : \mathbb{B}),\ \mathit{trans}(\Gamma \vdash ce_2 : \mathbb{B})$

6.8.2 Example: Transforming a CaTTS-CL Program into a CL$_{catts}$ Program

Let us turn attention back to the CaTTS-CL program

```
X is 2 working_week && Y is 1 month && X during Y
```

We assume that the used calendric types are specified in a CaTTS-DL calendar specification as follows:

```
...
type day;
type week = aggregate 7 day @ day(1);
type month = aggregate
      31 day,
      alternate month(i)
        | (i div 12) mod 4 == 0 &&
          ((i div 12) mod 400 != 100 ||
           (i div 12) mod 400 == 0)     -> 29 day
        | otherwise                     -> 28 day
      end,
      ...
      31 day @ day(1);
type working_day = select day(i) where
          relative i in week >= 1 && relative i in week <= 5;
type working_week = week #< working_day;
...
```

The following transformation from this CaTTS-CL program into a CL$_{catts}$ program is then performed on the typing derivation for this program according to Definition 6.4:

$$
\begin{array}{ll}
 & \mathit{trans}(\{\} \vdash X \text{ is 2 } w\_week \;\&\&\; Y \text{ is 1 } month \;\&\&\; X \text{ during } Y{:}\mathbb{B}) \\
\mapsto_{(T-Task)} & \mathit{trans}(\{X : w\_week^*\} \vdash Y \text{ is 1 } month \;\&\&\; X \text{ during } Y{:}\mathbb{B}) \\
\mapsto_{(T-Event)} & \mathit{trans}(\{X : w\_week^*, Y : month\} \vdash X \text{ during } Y{:}\mathbb{B}) \\
\mapsto_{(T-Interval)} & \mathit{trans}(\{X : w\_week^*, Y : month\} \vdash X : day^*) \text{ during} \\
 & \mathit{trans}(\{X : w\_week^*, Y : month\} \vdash Y : day^*) \\
\mapsto_{(T-Sub)} & c^{day^*}_{w\_week^*}(\mathit{trans}(\{X : w\_week^*, Y : month\} \vdash X : w\_week^*)) \\
\mapsto_{(T-Sub)} & c^{day}_{month}(\mathit{trans}(\{X : w\_week^*, Y : month\} \vdash Y : month)) \\
\mapsto_{(T-Var)} & X \\
\mapsto_{(T-Var)} & Y
\end{array}
$$

Where $c^{day^*}_{w\_week^*}$ and $c^{day^*}_{month}$ are then defined as follows on the subtyping derivations according to Definition 6.3:

Let $\Gamma := \{X : w\_week^*, Y : month\}$ in

$$
\begin{array}{ll}
c^{day^*}_{w\_week^*}(\Gamma \vdash X : w\_week^*) = & \\
\mapsto_{(S-Trans)} & c^{day^*}_{w\_day^*}(c^{w\_day^*}_{w\_week^*}(\Gamma \vdash X : w\_week^*)) \\
\mapsto_{(S-Int)} & c^{day^*}_{w\_day^*}((c^{w\_day}_{w\_week})^*(\Gamma \vdash X : w\_week^*)) \\
\mapsto_{(T-Res)} & X \simeq X' \;,\; c^{day^*}_{w\_day^*}(\Gamma, X' : w\_day^* \vdash X' w\_day^*) \\
\mapsto_{(S-Int)} & (c^{day}_{w\_day})^*(\Gamma, X' : w\_day^* \vdash X' : w\_day^*) \\
\mapsto_{(S-Sel)} & X'' : day^*,\; X' \simeq X'' \\
c^{day^*}_{month}(\Gamma \vdash Y : month) = & \\
\mapsto_{(S-Trans)} & c^{day^*}_{day}(c^{day}_{month}(\Gamma \vdash Y : month)) \\
\mapsto_{(S-Aggr)} & Y \simeq Y',\; c^{day^*}_{day}(\Gamma, Y' : day \vdash Y' : day) \\
\mapsto_{(S-ICoer)} & Y'' : day^*,\; Y' \simeq Y''
\end{array}
$$

Then the CL$_{catts}$ program that corresponds to the CaTTS-CL program given above has the following form:

$$X \simeq X',\ X' \simeq X'',\ Y \simeq Y',\ Y' \simeq Y'',\ X''\ during\ Y''$$

where $\Gamma = \{X : w\_week^*, Y : month, X'' : day^*, Y'' : day^*, X' : w\_day^*, Y' : day\}$.

Note: In CaTTS' implementation, the translation replaces a sequence of conversion constraints computed from the typing and subtyping derivations by one constraint. For example, $X \simeq X'$, $X' \simeq X''$ is replaced by the conversion constraint $X \simeq X''$. The subtyping derivation is maintained in the coercion by concatenation of each of the conversion functions used in each of the corresponding conversion constraints.

6.8.3 Coherence

Having defined a coercion semantics for subtyping in CaTTS-CL, a problem arises that we need to be carful to avoid when implementing this coercion semantics. Suppose, the following CaTTS-DL calendar specification.

```
...
type day;
...
type sunday = select day(i) where relative i in week == 7;
type birthday = select day(i) where relative i in april == 8;
type sunday_and_birthday = sunday & birthday;
type sundayAbroad = select sunday(i) where
   sunday(i) within ["1992".."1999"];
type birsthdayDuringStudies = select birthday(i) where
   birthday(i) within  ["1996".."2001"];
type sundayAbroad_and_birsdayDuringStudies =
   sundayAbroad & birsthdayDuringStudies
...
```

This calendar specification generates the following (directed) graph of subtypes.

$$\begin{tikzcd}
 & \text{day} \arrow[dl] \arrow[dr] & \\
\text{sunday} \arrow[dr] \arrow[dd] & & \text{birthday} \arrow[dl] \arrow[dd] \\
 & \text{sunday\_and\_birthday} & \\
\text{sundayAbroad} \arrow[dr] & & \text{birthdayDuringStudies} \arrow[dl] \\
 & \text{sundayAbroad\_and\_birsdayDuringStudies} &
\end{tikzcd}$$

Further assume that this calendar specification is referred to in the following CaTTS-CL program.

```
...
X is 1 sunday\_and\_birthday &&
Y is 1 sundayAbroad\_and\_birsdayDuringStudies &&
X equals Y
...
```

Now suppose to evaluate this CaTTS-CL program using the coercion semantics for CaTTS-CL introduced above. In fact, this program is typeable in distinct ways:

Translating this CaTTS-CL program using the afore defined translation (cf. Definition 6.4) and coercion (cf. Definition 6.3), one might produce several different translations: recall the typing rule for the interval constraints:

$$\frac{\Gamma \vdash ce_1 : \tau_1^* \;\; \Gamma \vdash ce_2 : \tau_1^*}{\Gamma \vdash ce_1 \; \mathcal{R} \; ce_2 : \mathbb{B}} \quad (\text{T-Interval})$$

This rule merely tells the type checker that a time constraint like `X equals Y` is well-typed and has type boolean, if a common type τ_1^* for the expressions `X` and `Y` can be inferred. Using subsumption one might either promote `sunday_and_birthday` to `sunday` and then to `sunday`* and `sundayAbroad_and_birsdayDuringStudies` to `sundayAbroad` to `sunday` and then to `sunday`* or `sunday_and_birthday` to `birthday` and then to `birthday`* and `sundayAbroad_and_birsdayDuringStudies` to `birsdayDuringStudies` to `birthday` and then to `birthday`*. Translating the first typing derivation to CL$_{catts}$ would yield

$X^{sunday\_and\_birthday} \simeq X^{sunday^*}$, $Y^{sundayAbroad\_and\_birsdayDuringStudies} \simeq Y^{sunday^*}$, $X^{sunday^*} \; equals \; Y^{sunday^*}$.

Translating the second derivation would yield the following CL$_{catts}$ constraint.

$X^{sunday\_and\_birthday} \simeq X^{birthday^*}$, $Y^{sundayAbroad\_and\_birsdayDuringStudies} \simeq Y^{birthday^*}$, $X^{birthday^*} \; equals \; Y^{birthday^*}$

But $X^{sunday^*} \; equals \; Y^{sunday^*}$ and $X^{birthday^*} \; equals \; Y^{birthday^*}$ are very different CL$_{catts}$ constraints; they do not even have the same type. In other words, the choice of how to prove $\Gamma \vdash X \; equals \; Y : \mathbb{B}$ affects the way the translated program behaves.

The appropriate response to such problems is to impose an additional requirement on the translation which is called *coherence* [BTCGS91]. Coherence means that for any pair of typing derivation D_1 and D_2 with the same conclusion $\forall\alpha.\, \alpha :: K \;\; \Gamma \vdash ce : \alpha$, the translations for D_1 and D_2 behave equivalent in terms of the language CL$_{catts}$. That means in particular that the CaTTS-CL expression $\forall\alpha.\, \alpha :: K \;\; \Gamma \vdash ce : \alpha$ must evaluate to the same values in CL$_{catts}$ independed of any specific typing derivation for this expression.

To recover coherence for an implementation of the transformation of CaTTS-CL into CL$_{catts}$ a deterministic choice is made that does not restrict generality of the transformation: for any pair of expressions ce_1 of type α and ce_2 of type β that appear in a derivation of the rule (T-Interval) ce_1 and ce_2 are transformed into the *join* of α and β according to Proposition 3.1. Such a join always exists: the calendric types over which kinds α and β are defined must be specified within the same (finite set of aligned) calendar specification(s) $\mathcal{C}$. $\mathcal{C}$ has by definition a largest element (w.r.t. the subtype relation). With Proposition 3.1, this join is unique. This choice is made within any typing rule that implies a subtyping derivation. Thus, the translation fro CaTTS-CL to CL_{catts} is shown to be coherent.

6.9 Note: Typing CaTTS-DL Calendar Specifications

So far, this chapter has introduced the well-formedness relation for CaTTS-DL calendar specifications as well as typing and subtyping relations for CaTTS-CL programs typed after calendric types defined in CaTTS-DL. Furthermore, to evaluate typed CaTTS-CL programs by means of constraint solving (using CaTTS' multi-calendar temporal reasoner as introduced in Chapter 5), a coercion semantics for subtyping in CaTTS-CL has been defined .

As already mentioned in Chapter 4, CaTTS provides with means to define *calendars as types*, "typed" after CaTTS-DL calendar signatures. CaTTS-DL calendar specifications (and calendar functions which are nothing but parameterized calendar specifications) are typed after calendar signatures by means of *calendar instantiation* and *calendar matching* analogously as it is done in programming languages with module systems such as Standard ML [MTH90]. For calendar matching and instantiation one would define an environment, storing the pairs of identifiers and associated syntactic forms for the different kinds of declarations and bindings (cf. Appendix A). Declarations and bindings are check during parsing CaTTS-DL calendar specifications (resp. CaTTS-CL programs). In particular, the environment handles the non context-free identifiers in the CaTTS-syntax.

Calendar instantiation and calendar matching can be performed (in principle) straightforward in CaTTS analogous to module matching and instantiation in Standard ML, it is not further considered in this thesis.

Chapter 7

Conclusion

"Menschen, die wie wir an die Physik glauben, wissen, daß die Unterscheidung zwischen Vergangenheit, Gegenwart und Zukunft nur eine besonders hartnäckige Illusion ist."
(Albert Einstein, 1879 – 1955)

This thesis has introduced the calendar type language CaTTS, a programming language approach to data modeling and reasoning with calendars. CaTTS is a type language for calendar definitions that provides with a constraint-based reasoning approach to manipulate temporal and calendric data of calendric types defined in this language.

CaTTS consists of

- CaTTS-DL, a definition language, itself consisting of
  - CaTTS-TDL, a type definition language and
  - CaTTS-FDL, a date format definition language
- CaTTS-CL, a constraint language typed after CaTTS-DL definitions.

CaTTS provides with syntactic forms, called type constructors, to define arbitrary calendric types in some calendar specification. These type constructors rely on the concept of predicate subtypes. Using predicate subtype constructors one can define arbitrary time granularities as types in CaTTS-DL. The two subtype relations *aggregation of* and *inclusion of*, and, in particular, their (natural) composition, *subtype of*, provide means for conversions between values of such types during multi-calendar temporal reasoning. In CaTTS, multi-calendar temporal reasoning is based on constraint programming techniques. This multi-calendar constraint solver refers to and relies on calendric types defined in CaTTS-DL. It solves reasoning problems, in particular, appointment scheduling problems expressed in the language CaTTS-CL. CaTTS' constraint solver provides with an efficient solution to multi-calendar temporal reasoning. Multi-calendar temporal reasoning in CaTTS is linear in the number of constraints and in the number of variables according to the size of the variables' domains. Furthermore, CaTTS' multi-calendar constraint solver is complete.

It has been shown that defining calendars as types and their parameterization ensures maintenance and reuse of calendars. The "type" of a calendar provides a summary of that calendar itself.

Furthermore, it has been shown that using a "theory reasoning" approach to time and calendars that refers to such types allows for both, efficient multi-calendar temporal reasoning and user-friendly modeling of temporal and calendric data.

CaTTS facilitates the modeling and efficient processing of calendar and time data for several applications, in particular for solving multi-calendar appointment scheduling problems.

The usability of CaTTS' approach to calendar data modeling and reasoning has been exemplified.

Research on time and calendar models and reasoning approaches to time and calendars that has influenced the work on CaTTS, both from theory and from practice has been thoroughly surveyed, always in comparison to CaTTS' approach to this problem.

7.1 Results

In the following, the twofold thesis underlying the research reported about in this work that has been formulated in the introduction is recalled:

- *"Calendars as Types":* calendars are more conveniently expressed with dedicated language constructs. Types and type checking are as useful and desirable with calendric data types as with whatever other data types. Types complement data with machine readable and processable semantics. Type checking is a very popular and well established "lightweight formal method" to ensure program and system behavior and to enforce high-level modularity properties yielding in abstraction. Types and type checking enhance efficiency and consistency of any language.
- *"Theory Reasoning":* calendars are more efficiently processed with dedicated reasoning methods than with "axiomatic reasoning" of ontology languages like RDF and OWL. This makes search space restrictions possible that would not be possible if calendars and temporal notions would be specified in a generic formalism such as first-order logic and processed with generic reasoning methods such as first-order logic theorem provers.

This thesis is supported by documentary evidence: the calendar type language CaTTS has been designed and prototypically implemented. CaTTS allows for user-friendly modeling of arbitrary calendars and calendric data as well as for reasoning on such data. Defining time granularities as types in a CaTTS calendar specification using predicate subtyping allows for declarative and user-friendly modeling of such types, for annotation of calendric data in any document or program using CaTTS, for modularity of calendar specifications, and thus, for reuse and maintenance of calendars. Furthermore, predicate subtyping provides a means to automatically generate conversion functions between the values of such types. The novel conversion constraint, introduced with CaTTS' constraint solver refers to the predicate subtypes, in particular to the conversion functions generated from the type predicates. CaTTS' constraint solver used to solve multi-calendar temporal reasoning problems, and, in particular, appointment scheduling problems refers to and relies on the calendric types defined by predicate subtypes. This makes constraint reasoning more efficient (in fact linear in both the number of constraints and the number of variables with respect to the size of the domains associated with the variables). Furthermore, multi-calendar constraint reasoning is performed without loss of the semantics of the different calendric types.

7.1.1 Underlying Problem

As motivated in the introduction to this thesis, temporal and calendric data are diverse and heterogeneous. Such data often depend on cultural, professional, locational, and/or legal contexts, frequently involving rather subtle differences, even if the same cultural calendar is used. Undoubtly, time and calendars play an important role in many information systems in Artificial Intelligence as well as in Database research. Several applications involving often rather complex temporal and calendric data have been addressed in this thesis. Today's internationalization efforts in the World Wide Web as well as the vision of the Semantic Web give rise to further research on time and calendars. In particular, since the Web is, in general, not exclusively used by computer scientists but rather by people who do not want to tackle with computers, a tool to concisely modeling temporal and calendric data and constraints in a declarative, and, thus, user-friendly manner that can be efficiently manipulated and processed by computers turns out to be extremely useful.

Having surveyed a large amount of research done on time and calendar modeling and reasoning in Artificial Intelligence, in Database Systems, and in the Web and Semantic Web, it turns out that a practically applicable approach to time and calendars including declarative modeling and efficiently processing approaches that can also be used by the (heterogenous) "Web and Semantic Web clientele" is missing.

7.1.2 CaTTS: A Programming Language Approach to Time and Calendars

CaTTS' underlying time model is based on different concepts that have been defined within (theoretical) research on time and calendar formalisms. CaTTS' underlying time model is purely interval-based. It includes the concept of time granularities and two subtype relations, aggregation of and inclusion of (as well as a composition of these two relations), which are very useful in modeling calendars. Indeed, they reflect widespread forms of common-sense modeling and reasoning with calendric data.

CaTTS consists of two language parts, a type definition language, CaTTS-DL and a constraint language, CaTTS-CL.

CaTTS-DL is a language to specify arbitrary cultural and professional calendars. As examples given in this work show, CaTTS-DL is expressive enough to specify complex cultural calendars such as the Hebrew calendar. CaTTS-DL calendars are specified by finite collections of time granularities defined as types. Such calendric types are defined using predicate subtyping.

CaTTS-CL is a language to specify multi-calendar temporal reasoning problems, and, in particular, appointment scheduling problems over calendric data and constraints that refer to and rely on calendric types defined in CaTTS-DL.

7.1.3 CaTTS' Language Processors

CaTTS-DL's language processor is a type checker that is based on predicate subtyping. Predicate subtyping turns out to be extremely useful for calendric data, providing a means to define time granularities as types as well as conversions between values of such types. Conversions are required for (efficient) constraint solving on such data. Predicate subtyping introduces a semantics for the constraint language CaTTS-CL, and it allows for consistency checks of CaTTS-CL programs. Furthermore, some basic static type checks of

CaTTS-CL programs can be made. Beyond, predicate subtyping introduces a means to generate conversion functions from CaTTS-DL calendric type definitions. Those functions are accessed by CaTTS-CL's constraint solver.

Predicate subtyping is a novel approach to data modeling and processing of time and calendars which turns out to be extremely natural, allowing for user-friendly data modeling and efficient reasoning, keeping and, even, exploiting the semantics of different calendric types.

CaTTS-CL's language processor is a constraint solver used in order to solve multi-calendar temporal reasoning problems, and, in particular, appointment scheduling problems. It provides with a novel constraint, called conversion constraint, that allows for constraint reasoning over arbitrary calendar domains (that refer to the calendric types defined in CaTTS-DL). The constraint solver is defined in the constraint system typed finite domains, an extension of the existing constraint system finite domains with calendric types attached to the domains as well as the CaTTS conversion constraint. Multi-calendar constraint reasoning is performed without loss of semantics of the different calendric types. It thus increases efficiency of constraint solving involving calendric data. In fact, since the conversion functions accessed by those conversion constraints are automatically generated from CaTTS-DL type definitions before any constraint solving is taking place, CaTTS-CL's constraint solver is linear in the number of constraints as well as in the number of variables according to the size of the variables' associated domains. Furthermore, CaTTS' constraint solver is proved to be complete.

CaTTS' constraint solver goes beyond proposals in the literature for constraint reasoning with different time granularities.

Due to calendric types, the constraint solver can process arbitrary calendric data as long as it is typed after types defined in CaTTS-DL.

7.2 Perspectives for Future Research

This section mentions and discusses (possible) future research directions concerning CaTTS and its underlying ideas, i.e. to define calendars as types and to use theory reasoning approaches to manipulate and process the data of such types. The perspectives for future research which are discussed in the following point out the high potential of CaTTS' approach to data modeling and reasoning for specific application domains such as time and calendars.

7.2.1 Possible Extensions of the Type Language CaTTS

In what follows, some possible complements, enhancing CaTTS' data modeling, reasoning, and type checking facilities for calendars are addressed.

The subsequently discussed possible extensions of CaTTS and its two language processors, i.e. its constraint solver and its type checker, give evidence that CaTTS is a generic modeling language for time and calendars, provided with thoroughly chosen language constructs: CaTTS' language constructs can be easily combined as to support additional modeling facilities. Interdependent with CaTTS' modeling facilities, its language processors can be (rather) easily extended. In particular, CaTTS gives rise to be used in arbitrary applications that refer to time and calendars, easily adapting CaTTS' language constructs and processing tools to the application's needs.

7.2.1.1 Further Directions to Calendric Data Modeling

This section addresses possible complements to what has been introduced in Chapter 4. Recall that Chapter 4 has introduced the type language CaTTS consisting of the type definition language CaTTS-DL and the constraint language CaTTS-CL. So far, calendric data modeling in CaTTS is restricted to precise temporal and calendric data that can be expressed in terms of finite convex intervals in a calendric type defined in a CaTTS-DL calendar specification. Some possible complements to CaTTS' calendric data modeling features are addressed in the following.

Activities with Variable Duration. In CaTTS, activities (that represent finite convex intervals in a calendric type) can be either specified by a precise duration, e.g.

```
X is 2 working_day
```

specifying an activity X that represents intervals of working days of length 2, i.e. intervals that last for 2 working days or by specifying no duration at all, e.g.

```
X is working_day
```

specifying an activity X that represents intervals of working days of any length, i.e. intervals that last for an unknown number of working days.

Specification of activities might be easily extended to allow for specifying activities whose durations vary between a minimal and a maximal amount of time, e.g.

```
X is 2 to 4 working_day
```

specifying an activity X that represents intervals of working days of minimal length 2 (working days) and maximal length 4 (working days), i.e. X represents intervals that last at least for 2 working days and at most for 4 working days. Such an extension of activity specifications might be useful for several applications: for example, someone is looking for a flight from Munich to Tokyo and back where his/her stay in Tokyo might vary for some days depending on the dates of the most suitable flight.

Activities Representing Generalized Intervals. As already mentioned, in CaTTS, activities represent finite convex intervals with a calendric type. For several application such as travel planning involving correlated flight and hotel bookings, activities that represent (finite) non-convex intervals or even (infinite) periodic intervals with calendric type might be useful. Activities that refer to finite non-convex intervals could be represented by conjunctions of activity specifications such as

```
X is 2 working_day && X is 4 working_day
```

specifying an activity `X` that represents intervals of working days that consists of two components where the first refers to convex intervals of length 2 (working days) and the second component of this generalized interval refers to convex intervals of length 4 (working days). Thus `X` refers to non-convex intervals of working days, containing two components where the first has a duration of 2 (working days) followed by a gap in time followed by a second component of duration 4 (working days). One might further assume additional language constructs to relate the different (convex) components of such non-convex intervals to each other, using CaTTS-CL's time constraints such as `before`, `meets`, and `overlaps`.

Activities that refer to (infinite) periodic intervals might be useful in appointment scheduling, e.g. when someone wants to keep 1 hour on each Monday for private concerns within his/hers schedule. An activity `X` that represents intervals of 1 hour duration on each Monday might be represented as follows in CaTTS:

```
X is 1 hour during monday
```

Where `monday` is a calendric type defined in the considered CaTTS-DL calendar specification.

Preferences on Time Constraints. Assuming that some persons want to schedule a meeting. It might be natural that some of them prefer specific time slots. That is, it might be useful to provide the user of CaTTS-CL with the ability to specify priorities on his/her time constraints by adding a number (e.g. between 1 and 5 to a time constraint from weak to strong preference). For example,

```
X is 1 hour && X within "W20.2005" && X during monday(4)
```

could specify an activity `X` of 1 hour that must be within the 20^{th} week of 2005 and (if possible at all) during a Monday with preference 4 of 5.

Further extensions to express time constraints in CaTTS-CL might be, in addition to conjunctions of (primitive) time constraints, disjunctions of time constraints, negations of time constraints, and conditionals for time constraints.

Note: If the set of activities that can be represented in CaTTS-CL is extended, then some extensions become necessary that support additional time constraints to express conditions between those activities.

All those complements to calendric data and constraint modeling in CaTTS that are addressed above require (often rather complex) extensions of CaTTS' constraint solver.

7.2.1.2 Further Directions to Multi-Calendar Constraint Solving

This section addresses possible complements to what has been introduced in Chapter 5. Recall that Chapter 5 has introduced a constraint-based reasoner to solve multi-calendar temporal reasoning problems, in particular, appointment scheduling problems modeled in

CaTTS-CL. Such problems refer to calendric types defined in CaTTS-DL calendar specifications. First, some (possible) complements to CaTTS' current constraint solver are addressed. Subsequently, further research directions on constraint solving involving different domains in some discourse universe such as calendars are addressed. This points out that CaTTS' approach to typed CSPs using a conversion constraint to manipulate calendric data of different types is rather generic, and might be applied to other, similar, constraint problems.

Optimizations of CaTTS' Constraint Solver. As discussed in Chapter 5, CaTTS' constraint propagation algorithm is defined and implemented on interval domains. Using interval domains based on the consistency technique bounds consistency has the advantage that the implementation is rather efficient, i.e. linear, however, possibly with a large constant. Using interval domains has the disadvantage that only domains without gaps can be represented. This drawback particularly concerns CaTTS' conversion constraint: for example, converting the activity $X :: 1..2 + 1..1$ of type *week* to days using such an interval representation would result in the activity $Y :: 1..8 + 7..7$ of type *day*. Assume that weeks are defined from days such that each week has a duration of 7 days always starting on Mondays, and such that the day with index 1 is a Monday. The conversion from the domain associated with Y to that of X contains several time intervals which are not solutions, e.g. the interval from day 3 (Wednesday) to day 9 (Tuesday). This mismatch results from the fact that only interval boundaries of the domains of X and Y are considered by the bounds consistency algorithm. Instead using enumeration domains, and, thus, arc consistency [FA97, MS98, Apt03], X would be represented by a list of its values, i.e. $X :: [1 + 1, 2 + 1]$ and a conversion to days would yield $Y :: [1 + 7, 8 + 7]$, i.e. only those intervals of 7 days (the time intervals from day 1 to day 7 and from day 8 to day 14) that actually represent weeks. But using enumeration domains that enforce arc consistency is less efficient. An algorithm implementing arc consistency is at least cubic in the number of variables as well as in the number of constraints [FA97]. Since CaTTS deals with activities representing time intervals as well as conversion constraints, the size of the domains as well as the number of constraints inherently increase. Arc consistency is essentially only tractable for sufficiently small domains. Thus, an improvement of CaTTS' constraint solver might be a specification and implementation on interval domains combined with enumeration domains such that the required domain representation can be chosen at hand according to the problem considered. Such an implementation would require additional predicates to switch between the different representations.

Extending the Reasoning Facilities. As already addressed in the previous Section 7.2.1.1, CaTTS only supports activities which can be represented by finite convex intervals with calendric types. On the one hand, this restriction to finite intervals over a calendric type allows for an efficient implementation of a constraint solver for multi-calendar temporal reasoning problems. On the other hand, this restriction considerably limits the set of temporal reasoning problems that can be modeled and solved using CaTTS: several problems such as travel planning involve calendric data that refers to generalized, i.e. not necessarily convex and finite intervals. The time constraints provided with CaTTS-CL need to be extended such that they can propagate activities that refer to generalized intervals. In addition, further time constraints (e.g. particular relations that often appear between non-convex time intervals, or time constraints used in order to relate periodic intervals such as "periodically after") need to be specified and implemented.

Another extension that has been addressed in Section 7.2.1.1 concerns the specification of preferences on CaTTS-CL time constraints. A possible way to deal with preferences is the definition and propagation of so-called *soft-constraints* which have to be specified for CaTTS' particular needs.

The addressed extensions (cf. Section 7.2.1.1) to disjunctions and negations of constraints as well as conditionals, require general extensions of the constraint solver underlying CaTTS. Some of those extensions are discussed in [FA97].

Conversion Constraints for other Domains. Reasoning problems involving specific theories other than time and calendars that similarly refer to differently "grained" data such as location and topologies, measurements, physical sizes as well as trust data are appropriate candidates to adapt the idea of CaTTS' constraint solver for typed domains manipulated using a specific conversion constraint that refers to and relies on (user-defined) predicate types.

7.2.1.3 Further Directions to Type Checking with Calendric Types

This section addresses possible complements to what has been introduced in Chapter 6. Recall that Chapter 6 has introduced predicate subtyping in CaTTS to check well-formedness of CaTTS-DL calendar specifications and it introduces a coercion semantics of subtyping for CaTTS-CL programs. In the following, possible advanced type checking approaches with predicate subtypes in CaTTS are addressed.

Subtyping with Durations. The current implementation of CaTTS' type checker would reject a CaTTS-CL program such as

X **is** 1 day && Y **is** 1 day && (**shift** X **forward** 1 month) **equals** Y

The reason that this program would be rejected is that subtyping with durations of activities is not supported. Currently, the typing rule (T-Shift) for the constraint `shift X forward 1 month` requires the same calendric types for the kind of `X` (a point) and the kind of `1 month` (a duration). In this example, however `X` has type `day` whereas `1 month` has type duration of month, i.e. $\texttt{month}^n$. This restriction is built into CaTTS due to avoid any kind of imprecision: the result of shifting a day forward by 1 month is not clear at all. Rather the result might refer to the same day number one month later (this becomes critical when shifting the 31st January forward 1 month), to some day one month later, or to that day that is exactly shifted by the duration of 1 month in terms of days, unfortunately, this duration is variable (e.g. a Gregorian month might have 28 up to 31 days).

A possible solution to such kinds of problems, allowing for subtyping with durations that provides with sufficient flexibility for the user is the following implementation: convert the duration of `1 month` to an equivalent duration in terms of days that defines a range from the minimal to the maximal duration of 1 month in terms of days, i.e. 28 to 31 days and add the minimal duration to the minimal starting time of `X`, yielding the minimal starting time of `Y`, and the maximal duration to the maximal starting time of `X`, yielding the maximal starting time of `Y`. Thus, the day `X` is shifted forward by a duration that ranges from 28 to 31 days.

Type Checking with Predicate Subtypes. In programming languages, type checkers are usually intended to ensure the absence of certain undesirable behavior during program execution. CaTTS-DL is a specification language for calendars. Although program execution is not a primary concern for specification languages, type checking CaTTS-DL still serves to reject calendar specifications that are erroneous or undesirable in some way. Beyond the static type checks supported currently for CaTTS-DL calendar specifications there is further potential to investigate to what extent erroneous or undesirable calendar specifications might be rejected statically. For example, a calendar specification that contains the specification of a type whose set of values is empty should be rejected, because any program, and in particular, any CaTTS-CL program that refers to such an empty type is inconsistent. Similarly, one could check calendar specifications for different type specifications, however referring to some equivalent set of values (cf. Section 6.7). To perform such kinds of "reliable checks" of CaTTS-DL calendar specifications, predicate subtypes can be used by virtue of constraint solving (in principle) available for CaTTS' type checker. Type checking according to predicate subtypes is then performed by proof obligation generation from the type predicates using constraint reasoning.

Similarly, out-off bound checks of finite calendric types defined in some CaTTS-DL calendar specification and referred to in some CaTTS-CL program might be performed statically by providing CaTTS' type checker a means to call CaTTS' constraint solver during type checking the considered CaTTS-CL program. Currently, out-off bound checks are performed dynamically, during constraint solving.

Note: CaTTS' constraint solver needs to be extended with some further specific constraints to perform such "reliable checks" of CaTTS-DL calendar specifications. Additionally, a possibility for the type checker to call the constraint solver during (statically) type checking CaTTS-DL calendar specifications as well as CaTTS-CL programs is required.

7.2.2 Topologies as Types

The idea to model time granularities as types defined by specifying predicate sets in some calendar specification that can be referred to for reasoning purposes might also be applied to other theories such as topologies [BLS05].

7.2.2.1 Granularities

For this purpose, initially, the concept of time granularity as introduced in Chapter 3 need to be extended to some multi-dimensional case. Such a general definition of the concept of *granularity* is given in the following.

Definition 7.1 *An* **n-dimensional space** *is a pair ($\mathcal{A}^n$,$<_{\mathcal{A}^n}$) where $\mathcal{A}^n$ is an infinite set (isomorphic to $\mathbb{R}^n$) and $<_{\mathcal{A}^n}$ is a total order on $\mathcal{A}^n$ such that $\mathcal{A}^n$ is not bounded for $<_{\mathcal{A}^n}$. An element $a = (a_1, ..., a_n)$ of $\mathcal{A}^n$ is called* **n-point***.*

Note that since $\mathcal{A}$ is totally ordered (recall that it is isomorphic to $\mathbb{R}$), the total order is preserved over the Cartesian product of $\mathcal{A} \times \ldots \times \mathcal{A}$.

Definition 7.2 *Let ($\mathcal{A}^n$,$<_{\mathcal{A}^n}$) be an n-dimensional space.*
Let $G = \{g_i \mid i \in \mathbb{Z}\}$ be a set isomorphic to $\mathbb{Z}$. Let call the elements of G **granules***. A* **granularity** *is a (non-necessarily total) function $\mathcal{G}$ from G into the power set of $\mathcal{A}^n$, $\mathcal{P}(\mathcal{A}^n)$ such that for all $i, j \in \mathbb{Z}$ with $i < j$*

1. *if $\mathcal{G}(g_i) \neq \emptyset$ and $\mathcal{G}(g_j) \neq \emptyset$, then for all $a_i = (a_{i_1}, ..., a_{i_n}) \in \mathcal{G}(g_i)$ and for all $a_j = (a_{j_1}, ..., a_{j_n}) \in \mathcal{G}(g_j)$, then $a_i <_{\mathcal{A}^n} a_j$, and*
2. *If $\mathcal{G}(g_i) = \emptyset$, then $\mathcal{G}(g_j) = \emptyset$.*

According to Definition 7.2, granules of the same granularity are totally ordered and non-overlapping. The first condition of Definition 7.2 induces from the ordering of the n-point (of the n-dimensional space) the common-sense ordering on granules. The second condition of Definition 7.2 is purely technical: it makes it possible to refer to the *infinite* set $\mathbb{Z}$ also for *finite* sets of granules.

Examples of granularities are *time granularities* over ($\mathcal{A}$,$<_{\mathcal{A}}$) and *location granularities* over ($\mathcal{A}^2$,$<_{\mathcal{A}^2}$).

Granularities can be defined by specifying subtype relations (in terms of predicates). Two subtype relations, *aggregation of* and *inclusion of*, have been defined for (one-dimensional) time granularities (cf. Chapter 3). For example a type "working day" is an inclusion (in the common set-theoretic sense) of the type "day" since the set of working days is a subset of the set of days; the type "week" is an aggregation of the type "day" since each week can be defined as a time interval of days.

Similar to the subtype relations, aggregation and inclusion, between time granularities, aggregations and inclusions are defined between location granularities as follows.

Definition 7.3 *Let $\mathcal{G}$ and $\mathcal{H}$ be location granularities.*
$\mathcal{G}$ is an **inclusion subtype of** *$\mathcal{H}$, denoted $\mathcal{G} \subseteq \mathcal{H}$, i.e. every granule of $\mathcal{G}$ is a granule of $\mathcal{H}$.*

For example, the location granularity "subway station" is an inclusion subtype of the location granularity "station", selecting only those stations with subway connection.

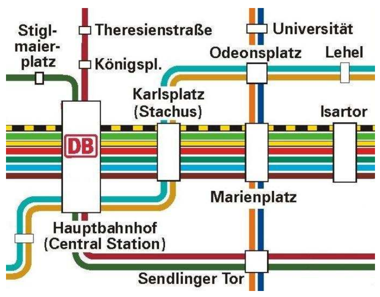


Figure 7.1: A part of Munich's subway and suburban train network.

Definition 7.4 *Let $\mathcal{G}$ and $\mathcal{H}$ be location granularities.*
$\mathcal{G}$ is an **aggregation-enclosure subtype of** *$\mathcal{H}$, denoted $\mathcal{G} \preceq \mathcal{H}$, if every granule of $\mathcal{G}$ is a 2-dimensional space over $\mathcal{H}$ and every granule of $\mathcal{H}$ is included in (exactly) one granule of $\mathcal{G}$.*
$\mathcal{G}$ is an **aggregation-connection subtype of** *$\mathcal{H}$, denoted $\mathcal{G} \prec \mathcal{H}$, if every granule of $\mathcal{G}$ is a 2-dimensional space over $\mathcal{H}$ and every granule of $\mathcal{H}$ is included in (at least) one granule of $\mathcal{G}$.*

For example, the location granularity "train network" is an aggregation-enclosure subtype of the location granularity "train connection", aggregating a set of train connections into a network. And a "train connection" itself is an aggregation-connection subtype of "station", connecting several stations to a train line. Note that connections are not necessarily disjoint: the same station might participate in different train lines, for example.

7.2.2.2 Topological Data Modeling

Having defined such a formal framework of granularities, one might start to develop a topology modeling language similar to the calendar modeling language CaTTS, also based on predicate subtyping.

In many cases, location reasoning pertains to routing and navigation tasks which rely on network infrastructures. In what follows, networks are used as a straightforward example for a topology, knowing that more complicated topologies might require further modeling features not addressed in this example. However, this example shows that a holistic model of the real world is hardly necessary for several applications involving locational data. For example, a journey involving the public underground system can be planned without any information about the *geographic* composition of the subway network. Knowledge about schedules and the *topological* structure suffices to find an optimal connection between two stations. Expressing such information in a location type language would provide both an abstraction from geographic coordinates and a means to enable reasoning on location data.

Let us consider a part of Munich's subway and suburban train network (cf. Figure 7.1) defined in a topology type system:

```
type subway_station = collect
     Sendlinger_Tor, Hauptbahnhof, Marienplatz, Stiglmaierplatz, ...
type train_station = collect Hauptbahnhof, Karlsplatz, Marienplatz, ...
type station = subway_station | train_station;
type U1 = connect Sendlinger_Tor - Hauptbahnhof - Stiglmaierplatz - ...
type U3 = connect Sendlinger_Tor - Marienplatz - Odeonsplatz - ...
type S2 = connect Hauptbahnhof - Karlsplatz - Marienplatz - ...
type network = collect U1, U3, S2;
```

A network such as that given in Figure 7.1 could be modeled as illustrated. The type `subway_station` is defined as a collection of named subway stations. A collection is a finite sequence of entities which satisfy certain constraints, in this case denoting a subway station. The definition of `train_station` is analogous. The type `station` is defines by conjunction of types `subway_station` and `train_station`. A subway line (`U1`, `U3`, etc.) is defined as an ordered connection of subway stations. If there is a subway train servicing station A, B, and C (in this order), then these stations are connected with each other to form a line. Therefore, a line is directed, i.e. a service is usually comprised by two lines operating in different directions. This is especially useful for modeling different time tables and changing services for off-peak operation, etc. A network is in turn a collection of lines.

The above sketched approach provides a means for modeling topological data required in many applications like routing problems and map representations by linking the topological information to spatial data, for example sets of coordinates (polygons) which denote areas. This linking also facilitates the use of established calculi like RCC8 [RCC92]. Furthermore, this approach allows for symbolic queries, such as finding out which stations lie in a certain quarter of the city or which district office is in charge of a road segment. Linked data is also necessary whenever for example polygons cannot be *directly* represented as location granularities, which is only possible in special cases (e.g. when areas can be regularly divided into same-size cells.)

7.3 Concluding Remarks

This thesis has motivated the benefits and advantages of a programming language approach to time and calendars. The benefits have been discussed on practical applications as well as in comparison to rather theoretical approaches to time and calendars suggested in the literature. The motivated programming language approach has been realized in terms of the language CaTTS. CaTTS' modeling languages CaTTS-DL and CaTTS-CL have been defined. The former is a type definition language to specify time granularities and calendars as types as well as date formats for the values of such types. The latter is a constraint language to express a wide range of temporal and calendric constraints on time intervals which have a calendric type attached. Furthermore, two language processors have been defined for CaTTS: a multi-calendar constraint solver and a type system. The multi-calendar constraint solver is used to solve reasoning problems expressed in CaTTS-CL. The multi-calendar constraint solver is defined in the constraint system typed finite domains. The solver comes with a novel constraint, the *conversion constraint*. This constraint enables constraint solving over arbitrary calendar domains of calendric types defined in CaTTS-DL. A set of proof rules formalizes the constraint propagation algorithm of this solver. A prototype implementation of the solver is provided and shown to be efficient, i.e. linear in the number of constraints and in the number of variables regarding the size of the domains associated with the variables. Completeness of this implementation has also been proved. The type system is used to (statically) type check well-formedness of CaTTS-DL calendar specifications and correctness of CaTTS-CL programs. Calendric types are specified in terms of predicate subtypes and subtyping is defined by implicit coercion. The type system and the coercion have been prototypically implemented. Soundness and correctness of the implementation of the type system are sketched but not formally proved. The implementation of the coercion semantics for subtyping in CaTTS-CL is shown to be coherent.

Appendix A

CaTTS' Syntax

A.1 Reserved Words

The following table lists the reserved words used in CaTTS, in CaTTS-DL as well as in CaTTS-CL. Reserved words cannot be used as identifiers.

after	aggregate	alternate	backward	before	begin_of
by	cal	calendar	cal_fun	cat	catalog
contains	div	downto	duration	during	else
end	end_of	equals	extend	finishes	finished_by
for	format	forward	group	if	import
in	index	macro	max	meets	met_by
min	mod	named	none	on_or_after	on_or_before
otherwise	overlaps	overlapped_by	prog	program	reference
refinement	relative	select	shift	shorten	sig
starts	started_by	then	to	type	unqualified
upto	use_calendar	use_format	where	with	within

!= #< & && ( ) * + , − −> . .. : ; < <: <:: <=
= == > >= @ [ ] \ | || ~@ ^{n} *

A.2 Constants

A *time constant* is either a part or a date of any calendric type (time granularity) defined in CaTTS-DL. A part is a type constructor followed by any non-empty sequence of digits, possibly preceded by the minus symbol "−" within parentheses "( )". A date is any non-empty sequence of printable characters, digits, and/or spaces between quotes " "" following the definitions of some date format specified in a CaTTS-FDL catalog. An underlying alphabet of 256 characters (number 0 to 255) is assumed where the characters number 0 to 127 coincide with the ASCII character set[1].

A *duration constant* is an (unsigned) integer followed by a type constructor.

The *formatting characters* are a subset of the non-printable characters including at least space, tab, and newline.

In the following, Con denotes the class of all constants; k is used to range over Con.

A.3 Comments

A *comment* is any character sequence within *comment brackets,* i.e. (*   *). Comment brackets may not be nested in CaTTS programs[2]. An unmatched comment bracket is detected by the compiler.

A.4 Identifiers

The following table itemizes the classes of identifiers appearing in CaTTS; for each class the name for the elements used to range over the corresponding class is given.

k ∈ Con	*constants*	
i ∈ IndVar	*index variables*	
X ∈ ConVar	*(constraint) variables*	
m ∈ MacId	*macro identifiers*	long
t ∈ TyCon	*type constructors*	long
g ∈ GrCon	*group constructors*	long
c ∈ CalId	*calendar identifiers*	long
s ∈ CalSigId	*calendar signature identifiers*	
f ∈ CalFunId	*calendar function identifiers*	
p ∈ ProgId	*program identifiers*	
fo ∈ FormId	*format identifiers*	ext
ct ∈ CatId	*catalog identifiers*	ext

For each class of identifiers Y marked "long" there is a class longY of *long identifiers*; if y ranges over Y then *longy* ranges over longY. The syntax of the long identifiers is given by the following:

`longy ::=`		*long identifiers:*
	`y`	*identifiers*
	`c`$_1$`.  ...  .c`$_n$`.y`	*qualified identifiers*, $n \geq 1$

[1]Note that an extension to the UTF-8 or the Unicode alphabet would be easy to perform in CaTTS. A restriction to ASCII is merely used for CaTTS' prototype implementation.

[2]Note that CaTTS could be easily supplemented with nested comments, if necessary.

The long identifiers constitute a link between (type) definitions and (calendar) specifications.

For each class of identifiers Z marked "ext" there is a class extZ of *extended catalog identifiers*; if *z* ranges over Z then *extz* ranges over extZ. The syntax of the extended catalog identifiers is given by the following:

`extz ::=`		*extended catalog identifiers:*
	`z`	*identifiers*
	$\texttt{a}_1\texttt{.}\ \ \texttt{...}\ \ \texttt{.a}_n\texttt{.z}$	*qualified identifiers,* $n \geq 1$

Any (qualified) identifier must be alphanumeric, i.e. any sequence of letters, digits, and underscores (_). Reserved words are excluded in any case.

The term *identifier* is used throughout this text to refer to non-qualified identifiers only. The term *qualified identifier* is exclusively used for extended or long identifiers. In any other case, the term *(qualified) identifier* is used.

Note: The "." used to connect a calendar identifier (resp. a catalog identifier) with some other identifier from one of the groups of identifiers marked "long" (resp. "ext") is a CaTTS- operator applied on the calendar identifier (resp. on the catalog identifier) and the other identfifier referred to.

Note: Except for identfier specifications, CaTTS is a context-free language. CaTTS' parser determines the class an identfier belongs to by case on the syntactic supplements as follows:

- Constants appear in quotes "and ", indices are integer numbers, and variables are strings, all of them may be bound in a value binding.
- A type identifier and its associated binding is recognized by the keyword `type`.
- A group identifier and its associated binding is recognized by the keyword `group`.
- A macro identifier and its associated binding is recognized by the keyword `macro`.
- A calendar identifier and its associated binding is recognized by the keyword `calendar`.
- A calendar signature identifier and its associated binding is recognized by the keyword `calendar type`.
- A calendar function identifier and its associated binding is recognized by the keyword `cal_fun`.
- A format idenfifier and its associated binding is recognized by the keyword `format`.
- A catalog idenfier and its associated binding is recognized by the keyword `catalog`.

- A (CaTTS-CL) program identifier and its associated binding is recognized by the keyword `program`.

For each class of identifiers, a context is provided, selecting the identifiers and corresponding bindings during parsing. The use of the different classes of identifiers in CaTTS-DL as well as in CaTTS-CL is imposed by the syntax of the languages.

A.5 Grammar

CaTTS' grammar, including the syntactic forms for both language formalisms CaTTS-DL and CaTTS-CL is given, in an EBNF-like notation. The grammar has an EBNF-like notation because both CaTTS-DL and CATTS-CL declarations are not context-free, i.e. they are context-sensitive. Declarations are therefore marked seperately during parsing (following the different classes of idenfiers) CaTTS-DL calendar specifications and CaTTS-CL programs (cf. Section A.4). In the used EBNF-like notations, optenals $\langle e \rangle$ denote 0 or 1 occurences of the expression e and repetitions $\{e\}$ denote $n \geq 0$ occurences of expression e.

Table A.1 contains the syntax of the (core) language CaTTS-DL. The syntax of CaTTS-TDL declarations (including calendars, calendar functions, and calendar signature specifications) is given in Table A.2. Table A.3 provides with the syntax of CaTTS-FDL declarations including catalog specifications. CaTTS-CL's (core) syntax is given in Table A.4. Table A.5 gives the syntax of CaTTS-CL declarations (including program specifications).

Note: Both languages, CaTTS-DL and CaTTS-CL include types for booleans, integers, characters, strings, and lists with common values and operations. Those types are supported with the language implementing CaTTS, in particular, those types are supported with Haskell types for booleans, integers, characters, strings, and lists CaTTS' reference implementation is provided within.

A.6 Syntactic and Closure Restrictions

- No identifier may be bound more than once.
- No declarations may describe the same identifier more than once.
- Calendar functions and calendar signatures may only contain local references; except for the calendar function (resp. calendar signature) identifier itself.
- Calendar functions and signatures are local; they are no first order structures of CaTTS (to preserve name consistency while compiling calendar modules).
- During parsing of a (finite set of aligned) CaTTS-DL calendar specification(s), the (pre-defined or user-defined) reference type `ref` is parsed to the type $\mathrm{r}_{\mathcal{C}}$.

τ ::=		*type expressions:*
	ref	*(user-def. or predef.) reference type*
	refinement n @ te	*refinement, $n \in \mathbb{N}$*
	aggregate te {,te} @ te	*(abs. anchored) aggregation*
	aggregate te {,te} ˜@ z	*(rel. anchored) aggregation, $z \in \mathbb{Z}$*
	select te where te	*selection*
	$\tau \& \tau$	*conjunction*
	$\tau \mid \tau$	*disjunction*
	$\tau \setminus \tau$	*exception*
	$\tau \# < \tau$	*restriction*
K ::=		*kinding expression:*
	τ	*point*
	τ^*	*interval*
	τ^n	*duration*
te ::=		*CaTTS-DL expressions:*
	i	*index*
	X	*variable*
	k	*constant*
	τ(i)	*part, $i \in \mathbb{Z}$, index*
	n τ	*duration, $n \in \mathbb{N}$*
	te $\mathcal{R}$ te	*interval relation*
	te $\odot$ te	*comparison*
	duration te	*duration of*
	index te	*index of*
	begin_of te	*begin of*
	end_of te	*end of*
	shift te forward te	*forward shift*
	shift te backward te	*backward shift*
	extend te by te	*interval extension*
	shorten te by te	*interval shortening*
	relative te in τ $\langle \odot i \rangle$	*relative in*
	relative te to τ $\langle \odot i \rangle$	*relative to*
	min(te)	*minimum*
	max(te)	*maximum*
	te && te	*conjunction*
	te $\parallel$ te	*disjunction*
	alternate te $\mid$ te $\rightarrow$ te {$\mid$ te $\rightarrow$ te}	*alternate*
	if te then te else te	*conditional*
	te *arithm* te	*arithmetics*
$\mathcal{R} \in$	{equals, before, after, starts, started_by, finishes, finished_by, during, contains, meets, met_by, overlaps, overlapped_by, within, on_or_before, on_or_after}	
$\odot \in$	{==, <=, <, >, >=, ! =}	
arithm $\in$	{+, −, ∗, mod, div}	

Table A.1: The syntax of CaTTS-DL including CaTTS-TDL and CaTTS-FDL.

dcl ::=		*declarations:*
	type tybind	*types*
	group groupbind	*groups*
	macro macbind	*macros*
	valbind	*values*
	dcl;dcl	*sequentials*
tybind ::=		*type bindings:*
	t = τ	
groupbind ::=		*group bindings:*
	g = with τ {type t where te {te} for t_1=t_2} end	
macbind ::=		*macro bindings:*
	m = te	
valbind ::=		*value bindings:*
	X $\langle$: $\tau\rangle$ $\langle$= te$\rangle$	
	i = z	$z \in \mathbb{Z}$
caldcl ::=		*calendar declarations:*
	dcl	*declarations*
	calendar calbind	*calendars*
		empty
	caldcl;caldcl	*sequentials*
calbind ::=		*calendar binding:*
	c$\langle$ (:sige)$\rangle$ = cale	
cale ::=		*calendar expressions:*
	cal caldcl end	*generative*
	longc	*identifiers*
	f(cale)	*function applications*
sigdcl ::=		*signature declarations:*
	calendar type sigbind	*generative*
		empty
	sigdcl;sigdcl	*sequentials*
sigbind ::=		*signature bindings:*
	s = sige	
sige ::=		*signature expressions:*
	sig spec end	*generative*
	s	*identifiers*
spec ::=		*specifications:*
	type t <: τ	*aggregations*
	type t c : τ	*inclusions*
	group g c : τ	*groups*
	calendar c:sige	*calendars*
	spec;spec	*sequentials*
fundcl ::=		*function declarations:*
	cal_fun funbind	*generative*
		empty
	fundcl;fundcl	*sequential*
funbind ::=		*function bindings:*
	f(c:s):s' = cale	

Table A.2: The syntax of CaTTS-TDL declarations.

catdcl ::=		*catalog declarations:*
	catalog catbind	*generative*
	format fo:τ = fodesc where te	*formats*
		empty
	catdcl;catdcl	*sequentials*
catbind ::=		*catalog bindings:*
	ct$\langle$:s$\rangle$= cate	
cate ::=		*catalog expressions:*
	cat catdcl end	*generative*
	extct	*extended identifiers*
fodesc ::=		*format descriptions:*
	{str|id}	*str $\in$ String, identifier*

Table A.3: The syntax of CaTTS-FDL declarations.

- A (finite set of aligned) calendar specification(s) may have at most one type specified by refinement of the reference type `ref`; this type is translated (during parsing) to the type $r_{\mathcal{C}}$, and `ref` to an aggregation subtype of $r_{\mathcal{C}}$.
- Each calendar (or finite set of aligned calendars) $\mathcal{C}$ must have exactly 1 reference type $r_{\mathcal{C}}$.
- At most one CaTTS-DL calendar specification can be used unqualified in a CaTTS-CL program.
- At most one CaTTS-DL format specification can be used unqualified in a CaTTS-CL program.
- A `group` declaration is replaced by a finite set of type declarations during parsing.
- An expression like `select day(i) where relative i in week == 1` is parsed to `select X:day where relative (index X) in week== 1`.
- Type declarations may not be cyclic; for example if a type `year` is defined from a type `month`, then the definition of `month` may not refer to years.
- Arithmetic operations are only allowed on indices in the `alternate` constructed in aggregation subtype declarations and on indices in `macro` declarations.

A.7 Note: CaTTS' Reference Implementation

CaTTS' prototype is implemented in Haskell, in particular, the parser of CaTTS-CL and CaTTS-DL, conversion function generation from (user-defined) type predicates, type checking, and the transformation into the lower-level language CL_{catts} of CaTTS' constraint

ce ::=		*CaTTS-CL expressions:*
	k	*constant*
	X	*variable*
	i	*index*
	n τ	*duration,* $n \in \mathbb{N}$
	[ce..ce]	*endpoint interval*
	ce upto ce	*duration interval*
	ce downto ce	*duration interval*
	X is 1 τ	*event*
	X is τ	*task*
	X is n τ	*task with duration* $n \in \mathbb{N}$
	ce $\mathcal{R}$ ce	*interval constraint*
	ce $\odot$ ce	*metric constraint*
	duration ce	*duration of*
	index ce	*index of*
	begin_of ce	*begin of*
	end_of ce	*end of*
	shift ce forward ce	*forward shift*
	shift ce backward ce	*backward shift*
	extend ce by ce	*interval extension*
	shorten ce by ce	*interval shortening*
	relative ce in τ $\langle \odot$ i $\rangle$	*relative in*
	relative ce to τ $\langle \odot$ i $\rangle$	*relative to*
	min(ce)	*minimum*
	max(ce)	*maximum*
	ce && ce	*conjunction*

$\mathcal{R} \in$	{equals, before, after, starts, started_by, finishes, finished_by, during, contains, meets, met_by, overlaps, overlapped_by, within, on_or_before, on_or_after}
$\odot \in$	{==, <=, <, >, >=, ! =}

Table A.4: The syntax of CaTTS-CL.

progdcl ::=		*program declarations:*
	ce	*constraint*
	use_calendar⟨unqualified⟩longc$_1$ ... longc$_n$;	*use calendars, $n \geq 1$*
	use_format⟨unqualified⟩cate$_1$ ... cate$_n$;	*use formats, $n \geq 1$*
	import lib$_1$ ... lib$_n$;	*imports, $n \geq 1$*
	program progbind	*generative*
		empty
	progdcl;progdcl	*sequentials*
progbind ::=		*program bindings:*
	p proge	
proge ::=		*program expressions:*
	prog progdcl end	*declarations*
	p	*identifiers*

Table A.5: The syntax of CaTTS-CL declarations.

solver. The constraint solver is implemented in CHR (Constraint Handling Rules) embedded into Sicstus Prolog.

For proof of concept, a small subset of the language CaTTS has been implemented. The implementation excludes the following (advanced) features of CaTTS

- macros,
- calendar signature matching,
- imports of externally specified functions in CaTTS-DL calendar specifications, and
- data format specifications in CaTTS-FDL.

Only a rudimentary translation from CL$_{catts}$ back into CaTTS-CL after constraint propagation is supported.

Conversion function generation is implemented only for specific predicates (cf. Chapter 5). Furthermore, subtyping with duration types is not supported in the current prototype implementation.

CaTTS' prototype implementation is sufficient for proof of concept. In particular, CaTTS' novel features, conversion function generation, type checking and subtyping with calendric types, and constraint reasoning with different (calendar) domains in terms of CaTTS' novel conversion constraint are realized.

The prototype will be published as soon as the remaining features are realized, as well. Note that the main focus of the thesis is a formal specification of a type language for calendars with multi-calendar reasoning approaches rather than its implementation.

Appendix B

A CHR Implementation of CaTTS' Constraint Propagation Algorithm

CaTTS' prototype constraint propagation algorithm is implemented in CHR (Constraint Handling Rules) embedded into SICSTUS Prolog [FA97]. The implementation is a straightforward transcription of the declarative proof rules given in Tables 5.6, p.149, 5.4, p.147, 5.5, p.148, 5.8, p.149, 5.7, p.149, and 5.9, p.152 in Chapter 5.

The implementation departs from the declarative proof rules as it is in the following explained on the proof rule for the time constraint *before*:

$$\frac{\langle X\ \mathit{before}\, Y; X :: n_X..m_X + d_X^-..d_X^+, Y :: n_Y..m_Y + d_Y^-..d_Y^+\rangle}{\begin{array}{c}\langle X\ \mathit{before}\, Y;\\ X :: n_X..min(m_X, (min(m_{E_X}, m_Y - 1) - d_X^- + 1)) + d_X^-..min(d_X^+, (min(m_{E_X}, m_Y - 1) - n_X + 1))\\ Y :: max(n_Y, n_{E_X} + 1)..m_Y + d_Y^-..d_Y^+\rangle\end{array}}$$

This proof rule incorporates

- the reduction rule for the calendar domain associated with the variable X,
- the reduction rule for the calendar domain associated with the variable Y, and
- the application of the bounds consistency rule to the domain of X and to the domain of Y by taking the minimal and maximal values (for starting times and durations), respectively of the interval domain bounds from the old and new domains and replace this reduced domain with the old and new ones.

Those steps are performed successively, using separated propagation rules by the algorithm.

1. Application of *before* on X, which introduces a new calendar domain for X (using the rule `before_x`).

2. Application of the bounds consistency rule on the old and new bounds of the interval domains for starting times and durations of X, yielding a reduced calendar domain for X (using the rules `intersection` and `empty`).
3. Application of *before* on Y, which introduces a new, reduced calendar domain for Y (using the rule `before_y`).
4. Application of the bounds consistency rule on the old and new bounds of the interval domains for starting times and durations of Y, yielding a reduced calendar domain for Y (using the rules `intersection` and `empty`).

The rules are implemented such that they can only be applied, if some bound can be reduced by using so-called *guards* in CHR [FA97].

B.1 Constraints and Functions Available for the Constraint Solver

- Prolog built-in arithmetic constraints:

```
is, <,=<,>,>=, \=
```

- Prolog built-in arithmetic functions:

```
min,max,+,-
```

- Prolog built-in predicates to manipulate lists
- conversion functions generated from type predicates in CaTTS-DL calendar specifications where `X_s:TypeS`, `X_t:TypeT`, and `TypeS` $\leq$ `TypeT`:

```
%for predecessor, converts to supertype:
convert_to_join_minus(X_s,TypeS,TypeT,X_t).

%for successor, converts to supertype:
convert_to_join_plus(X_s,TypeS,TypeT,X_t).

%conver to subtype:
%this predicate might return a list of values from which the minimum
%or the maximum is chosen in the conversion constraint
convert_from_join(X_t,TypeS,TypeT,X_s).

 %for predecessor, converts to subtype:
convert_from_join_minus(X_t,TypeS,TypeT,X_s).

%for successor, converts to subtype:
convert_from_join_plus(X_t,TypeS,TypeT,X_s).
```

- functions to related indices relative in/to indices in other types; generated from type definitions in CaTTS-DL calendar specifications

```
%for relative index X in TypeT = Index where IListX represents index X
%this function applies the conversion functions
s_in_t(Index,TypeT,IListX,TypeX,NewIListX).

%for relative index X to TypeT = Index where IListX represents index X
%this function applies the conversion functions
s_to_t(Index,TypeT,IListX,TypeX,NewIListX).
```

B.2 Activity Constraints

- activity is a 4-ary constraint

```
constraints activity/4.
activity(X,Kind,Type,Representation)
```

where `X` the variable, `Kind` either `event` or `task`, `Type` the CaTTS-DL type of the activity, and `Representation` the representation of the event or task as follows:

  - for events: `start(LS..HS)`
  - for tasks: `i(start(N..M),dur(LD..HD))`

- auxiliary predicates for activity constraints to get and set min/max interval domain bounds for starting times, durations, and ending times as well as the corresponding index list of an event

```
%for starting times
getLStart(X,Kind,Representation,NewRep).
setLStart(X,Kind,TypeX,Representation,NewRep).
getHStart(X,Kind,Representation,NewRep).
setHStart(X,Kind,TypeX,Representation,NewRep).

%for ending times; imply reset of starting times and durations
getLEnd(X,Kind,Representation,NewRep).
setLEnd(X,Kind,TypeX,Representation,NewRep).
getHEnd(X,Kind,Representation,NewRep).
setHEnd(X,Kind,TypeX,Representation,NewRep).

%for durations
getLDur(X,Kind,Representation,NewRep).
setLDur(X,Kind,TypeX,Representation,NewRep).
getHDur(X,Kind,Representation,NewRep).
setHDur(X,Kind,TypeX,Representation,NewRep).
```

B.3 Bounds Consistency

```
empty @ activity(X,event,TypeX,start(LSX..HSX)) <=> HSX < LSX | fail.
empty @ activity(X,task,TypeX,i(start(LSX..HSX),dur(LDX..HDX))) <=>
        HSX < LSX | fail.
empty @ activity(X,task,TypeX,i(start(LSX..HSX),dur(LDX..HDX))) <=>
        LDX > HDX | fail.

intersection @ activity(X,event,TypeX,start(LS1..HS1)),
               activity(X,event,TypeX,start(LS2..HS2)) <=>
               LSX is max(LS1,LS2), HSX is min(HS1,HS2),
               activity(X,event,TypeX,start(LSX..HSX)).
intersection @ activity(X,task,TypeX,i(start(LS1..HS1),dur(LD1..HD1))),
               activity(X,task,TypeX,i(start(LS2..HS2),dur(LD2..HD2))) <=>
               LSX is max(LS1,LS2), HSX is min(HS1,HS2),
               LDX is max(LD1,LD2), HDX is min(HD1,HD2),
               activity(X,task,TypeX,i(start(LSX..HSX),dur(LDX..HDX))).
```

Note: Those rules affect activity constraints only; all further CaTTS-constraints remain unaffected.

B.4 Time Constraints

- the transformation rules for time constraints (cf. Table 5.6, p.149 and Table 5.8, p.149) are straightforward implemented by translating the proof rules into CHR simplification rules, e.g.

  ```
  X before X <=> fail.
  ```

- the domain reduction rules for time constraints are straightforward implemented by translating the proof rules into CHR propagation rules

  – interval time constraint $\mathcal{R}$ (cf. Table 5.4, p.147), e.g. `before` and its inverse `after`

    ```
    before_x @ X before Y, activity(X,KindX,TypeX,ReprX),
               activity(Y,KindY,TypeY,ReprY) ==>
               getHEnd(X,KindX,ReprX,HEX),
               getHStart(Y,KindY,ReprY,HSY),
               NewHEX is HSY-1, NewHEX < HEX |
               setHEnd(X,KindX,TypeX,ReprX,NewHEX).

    before_y @ X before Y, activity(X,KindX,TypeX,ReprX),
               activity(Y,KindY,TypeY,ReprY) ==>
               getLEnd(X,KindX,ReprX,LEX),
               getLStart(Y,KindY,ReprY,LSY),
               NewLSY is LEX+1, NewLSY > LSY |
               setLStart(Y,KindY,TypeY,ReprY,NewLSY).

    X after Y :- Y before X.
    ```

- metric time constraints (cf. Table 5.5, p.148), e.g. `shift_forward` and `relative_in`

```
shift_f_y @ shift_forward(X,LD..HD,Y),
            activity(X,KindX,TypeX,ReprX) ==>
            getLStart(X,KindX,ReprX,LSX),
            getHStart(X,KindX,ReprX,HSX),
            getLDur(X,KindX,ReprX,LDX),
            getHDur(X,KindX,ReprX,HDX),
            LSY is LSX+LD-1, HSY is HSX+HD-1,
            activity(Y,task,TypeX,i(start(LSY..HSY),dur(LDX..HDX))).

shift_f_x @ shift_forward(X,LD..HD,Y),
            activity(Y,KindY,TypeY,ReprY) ==>
            getLStart(Y,KindY,ReprY,LSY),
            getHStart(Y,KindY,ReprY,HSY),
            getLDur(Y,KindY,ReprY,LDY),
            getHDur(Y,KindY,ReprY,HDY),
            LSX is LSY-LD+1, HSX is HSY-HD+1,
            activity(X,task,TypeY,i(start(LSX..HSX),dur(LDY..HDY))).

relative_in @ relative_in(X,TypeT,Index),
              activity(X,event,TypeX,ReprX) ==>
              getLStart(X,event,ReprX,LSX),
              getIndex(X,event,ReprX,IListX),
              s_in_t(Index,TypeT,IListX,TypeX,[First|NewIListX]),
              LSX < First | setLStart(X,event,TypeX,ReprX,First).

relative_in @ relative_in(X,TypeT,Index),
              activity(X,event,TypeX,ReprX) ==>
              getHStart(X,event,ReprX,HSX),
              getIndex(X,event,ReprX,IListX),
              s_in_t(Index,TypeT,IListX,TypeX,NewIListX),
              last(NewIListX,Last),
              HSX > Last | setHStart(X,event,TypeX,ReprX,Last).
```

- metric time constraints on activity durations $\odot$ (cf. Table 5.7, p.149), e.g. $\leq$

```
le_x @ X le Y, activity(X,KindX,TypeX,ReprX),
       activity(Y,KindY,TypeY,ReprY) ==>
       getHDur(X,KindX,ReprX,HDX),
       getHDur(Y,KinsY,ReprY,HDY),
       HDX > HDY | setHDur(X,KindX,TypeX,ReprX,HDY).

le_y @ X le Y, activity(X,KindX,TypeX,ReprX),
       activity(Y,KindY,TypeY,ReprY) ==>
       getLDur(X,KindX,ReprX,LDX),
       getHDur(Y,KindY,ReprY,LDY),
       LDX > LDY | setLDur(Y,KindY,TypeY,ReprY,LDY).
```

Note: The types `TypeX` and `TypeY` must be equal for an application of any time constraint. This requirement is not tested here, because it is already ensured through type checking and subsequent transformation of a CaTTS-CL program into an equivalent CL$_{catts}$ program.

B.5 Conversion Constraint

This rule implements the proof rule given in Table 5.9, p.152.

```
convert_x @ X ~ Y, activity(X,KindX,TypeX,ReprX),
            activity(Y,KindY,TypeY,ReprY) ==>
            getLStart(Y,KindY,ReprY,LSY),
            getLStart(X,KindX,ReprX,LSX),
            convert_from_join_plus(LSY,TypeY,TypeX,LSY_in_X),
            LSX < LSY_in_X |
            setLStart(X,KindX,TypeX,ReprX,LSY_in_X).

convert_x @ X ~ Y, activity(X,KindX,TypeX,ReprX),
            activity(Y,KindY,TypeY,ReprY) ==>
            getHStart(Y,KindY,ReprY,HSY),
            getHStart(X,KindX,ReprX,HSX),
            convert_from_join(HSY,TypeY,TypeX,HSY_in_X),
            min(HSY_in_X,MinHSY_in_X),
            HSX > MinHSY_in_X |
            setHStart(X,KindX,TypeX,ReprX,HSY_in_X).

convert_x @ X ~ Y, activity(X,KindX,TypeX,ReprX),
            activity(Y,KindY,TypeY,ReprY) ==>
            getHStart(Y,KindY,ReprY,HSY),
            getLEnd(Y,KindY,ReprY,LEY),
            getLDur(X,KindX,ReprX,LDX),
            convert_from_join(HSY,TypeY,TypeX,HSY_in_X),
            min(HSY_in_X,MinHSY_in_X),
            convert_from_join(LEY,TypeY,TypeX,LEY_in_X),
            max(LEY_in_X,MaxLEY_in_X),
            LDY_in_X is MaxLEY_in_X - MinHSY_in_X + 1,
            LDX < LDY_in_X |
            setLDur(X,KindX,TypeX,ReprX,LDY_in_X).

convert_x @ X ~ Y, activity(X,KindX,TypeX,ReprX),
            activity(Y,KindY,TypeY,ReprY) ==>
            getLStart(Y,KindY,ReprY,LSY),
            getHEnd(Y,KindY,ReprY,HEY),
            getHDur(X,KIndX,ReprX,HDX),
            convert_from_join_plus(LSY,TypeY,TypeX,LSY_in_X),
            convert_from_join_minus(HEY,TypeY,TypeX,HEY_in_X),
            HDY_in_X is HEY_in_X - LSY_in_X + 1,
            HDX > HDY_in_X |
            setHDur(X,KindX,TypeX,ReprX,HDY_in_X).

convert_y @ X ~ Y, activity(X,KindX,TypeX,ReprX),
            activity(Y,KindY,TypeY,ReprY) ==>
            getLStart(X,KindX,ReprX,LSX),
            getLStart(Y,KindY,ReprY,LSY),
            convert_to_join_minus(LSX,TypeX,TypeY,LSX_in_Y),
            LSY < LSX_in_Y |
            setLStart(Y,KindY,TypeY,ReprY,LSX_in_Y).
```

```
convert_y @ X ~ Y, activity(X,KindX,TypeX,ReprX),
            activity(Y,KindY,TypeY,ReprY) ==>
            getHStart(X,KindX,ReprX,HSX),
            getHStart(Y,KindY,ReprY,HSY),
            convert_to_join_minus(HSX,TypeX,TypeY,HSX_in_Y),
            HSY > HSX_in_Y |
            setHStart(Y,KindY,TypeY,ReprY,HSX_in_Y).

convert_y @ X ~ Y, activity(X,KindX,TypeX,ReprX),
            activity(Y,KindY,TypeY,ReprY) ==>
            getHStart(X,KindX,ReprX,HSX),
            getLEnd(X,KindX,ReprX,LEX),
            getLDur(Y,KindY,ReprY,LDY),
            convert_to_join_minus(HSX,TypeX,TypeY,HSX_in_Y),
            convert_to_join_plus(LEX,TypeX,TypeY,LEX_in_Y),
            LDX_in_Y is LEX_in_Y - HSX_in_Y + 1,
            LDY < LDX_in_Y |
            setLDur(Y,KindY,TypeY,ReprY,LDX_in_Y).

convert_y @ X ~ Y, activity(X,KindX,TypeX,ReprX),
            activity(Y,KindY,TypeY,ReprY) ==>
            getLStart(X,KindX,ReprX,LSX),
            getHEnd(X,KindX,ReprX,HEX),
            getHDur(Y,KIndY,ReprY,HDY),
            convert_to_join_minus(LSX,TypeX,TypeY,LSX_in_Y),
            convert_to_join_plus(HEX,TypeX,TypeY,HEX_in_Y),
            HDX_in_Y is HEX_in_Y - LSX_in_Y +1,
            HDY > HDX_in_Y |
            setHDur(Y,KindY,TypeY,ReprY,HDX_in_Y).
```

To compute one or all solutions, this constraint propagation algorithm is combined with a standard search algorithm. In particular, Prolog's built-in backtracking is used. Labeling is used to bind each variable to its possible values one after the other.

B.6 Termination

In what follows, we show that the constraint propagation algorithm as previously introduced terminates. That means, in particular, that the algorithm fails if the considered CSP is inconsistent and if the algorithm returns an equivalent, but simplified CSP, then the simplified CSP is bounds consistent according to Definition 5.5.

We first state that from the set of time constraints and bounds consistent calendar domains, it is straightforward to generate all solutions to the bounds consistent problem. That means in particular that the previously defined solver is complete.

Lemma B.1 *If $\mathcal{P} = \langle \mathcal{C}; \mathcal{D}_\mathcal{E} \rangle$ where $\mathcal{D}_\mathcal{E} := X_1 \in D_1 \wedge \ldots \wedge X_n \in D_n$ is bounds consistent, then a solution $(d_1, \ldots, d_n) \in D_1 \times \ldots \times D_n$ to $\langle \mathcal{C}; \mathcal{D}_\mathcal{E} \rangle$ for every constraint $C \in \mathcal{C}$ on the variables $X_{i_1}, \ldots, X_{i_m}$, $(d_{i_1}, \ldots, d_{i_m}) \in \mathcal{C}$ exists.*

Proof B.1 *First, it is sufficient to note that the finite domain constraint $a*X \leq b*Y+c$, $a, b, c \geq 0$ is complete [FA97], i.e. if the constraint is bounds consistent, then it has a solution. Since every CL_{catts} time constraint is defined by a finite set of such finite domain constraints (according to the proof rules given in Table 5.6, Table 5.4, Table 5.5, Table 5.8, and Table 5.7), a bounds consistent time constraint has a solution in$\mathcal{P}$.*

Remains to proof that the conversion constraint $X^\alpha \simeq Y^\beta$, $\alpha \leq \beta$ (cf. Table 5.9) has a solution if it is bounds consistent. Let $X^\alpha :: n_X..m_X + d^-_X..d^+_X$ and $Y^\beta :: n_Y..m_Y + d^-_Y..d^+_Y$ after application of the proof rule for the conversion constraint and let $min(X^\alpha) := [n_X..n_X + d^-_X - 1]$ the minimal value of X^α and $min(Y^\beta) := [n_Y..n_Y + d^-_Y - 1]$ the minimal value of Y^β. By application of the proof rule for the conversion constraint, we show that this is a solution for $X^\alpha \simeq Y^\beta$.

Let $n_X := max(old\_n_X, c^+_{\beta\rightarrow\alpha}(old\_n_Y))$. By definition of the conversion function $c^+_{\beta\rightarrow\alpha}$, $c^+_{\beta\rightarrow\alpha}(old\_n_Y)$ is a starting time of an interval in type α. Thus, by applying the bounds consistency rule, n_X is a starting time of an interval in type α that is also started by an interval in type β. Let $n_Y + d^-_Y - 1 := max(old\_n_X, c^+_{\beta\rightarrow\alpha}(old\_n_Y)) + max(old\_d^-_X, (c_{\beta\rightarrow\alpha}(old\_n_{E_Y}) - c_{\beta\rightarrow\alpha}(old\_m_Y) + 1))$. As previously shown, $max(old\_n_X, c^+_{\beta\rightarrow\alpha}(old\_n_Y))$ starts an interval in type α that is also started by an interval in type β. With definition of the conversion function $c_{\beta\rightarrow\alpha}$, the duration of $[n_X..n_X + d^-_X - 1]$ corresponds to a duration of an equivalent time interval in type β. Since n_Y is a starting time of X that is also started in the domain of Y and since the duration of X is also represented in the duration of the domain of Y, the ending point of $[n_X..n_X + d^-_X - 1]$ in the domain of X is also represented by a corresponding ending time in the domain of Y. Thus, $[n_X..n_X + d^-_X - 1]$ has a corresponding solution in the domain of Y in type β.

Let $n_Y := max(old\_n_Y, c^-_{\alpha\rightarrow\beta}(old\_n_X))$. By definition of the conversion function $c^-_{\alpha\rightarrow\beta}$, $c^-_{\alpha\rightarrow\beta}(old\_n_X)$ is a starting time of an interval in type β. Thus, by applying the bounds consistency rule, n_Y is a starting time of an interval in type β that is also started by a starting time from the domain of X in type α. Let $n_Y + d^-_Y - 1 := max(old\_n_Y, c^-_{\alpha\rightarrow\beta}(old\_n_X)) + max(old\_d^-_Y, (c^+_{\alpha\rightarrow\beta}(old\_n_{E_X}) - c^-_{\alpha\rightarrow\beta}(old\_m_X) + 1))$. As previously shown, $max(old\_n_Y,$ $c^-_{\alpha\rightarrow\beta}(old\_n_X))$ is a starting time of Y in type β that has also a corresponding starting time in the domain of X in type α. With definition of the conversion functions $c^+_{\alpha\rightarrow\beta}$ and $c^-_{\alpha\rightarrow\beta}$, the duration of $[n_Y..n_Y + d^-_Y - 1]$ corresponds to a duration represented by the domain of X in type α. Since n_Y is a starting time of an interval represented by the domain of Y that has a corresponding starting time in the domain of X and since the duration of $[n_Y..n_Y + d^-_Y - 1]$ has a corresponding duration in the domain of X, the ending point of $[n_Y..n_Y + d^-_Y - 1]$ in the domain of Y is also represented by an ending point in the domain of X. Thus, the value $[n_Y..n_Y + d^-_Y - 1]$ has a corresponding value in the domain of X in type α.

With this result, $min(X^\alpha) := [n_X..n_X + d^-_X - 1]$ and $min(Y^\beta) := [n_Y..n_Y + d^-_Y - 1]$ is a solution to $X^\alpha \simeq Y^\beta$.

Since every CL_{catts} constraint has a solution, in particular the minimal value from its domain if $\mathcal{P}$ is bounds consistent, $\mathcal{P}$ has a solution.

To find a solution to a problem formulated in CL$_{catts}$, it is thus sufficient to transform it into an equivalent set which is bounds consistent. The previously given algorithm does it, if it is possible, and otherwise it halts with failure.

Proposition B.1 (Termination.) *Let $\mathcal{P}$ a CSP formulated in CL$_{catts}$. Let solve($\mathcal{P}$) be the algorithm implementing the proof rules given in in Table 5.6, Table 5.4, Table 5.5, Table 5.8, Table 5.7 and Table 5.9 as specified above. If $\mathcal{P}$ has a solution, then solve($\mathcal{P}$) successfully terminates and produces an equivalent bounds consistent set of solutions $\mathcal{P}'$, and otherwise it terminates with failure.*

Proof B.2

Claim 1: *The algorithm always terminates.*
This means in particular that the algorithms must always return something, i.e. that it cannot diverge. This we can do by observing that the number of constraints in the constraint store never increases. Since the number of rules is finite, as well, only a finite number of rules can be tried for a finite number of constraints. Thus, an infinite sequence of rule tries is not possible.

Claim 2: *Each application of a CL$_{catts}$ constraint C replaces the set of constraints participating in C by an equivalent one.*
This follows directly from the definition of the proof rules of the algorithm which are by definition equivalence preserving.

Claim 3: *If the algorithm successfully terminates, then the final problem $\mathcal{P}'$ is bounds consistent.*

- *the "bounds consistency rules" (i.e.* `empty` *and* `intersection` *removes one interval domain constraint*
- *assume that the remaining rules deal with non-empty interval domains only*
- *in each rule, at least one interval domain is strictly smaller in the conclusion than the corresponding interval domain in the premise, while the other interval domains remain unaffected*
- *a rule can only be applied, if at least one interval domain can be reduced; thus, if solve($\mathcal{P}$) returns $\mathcal{P}'$, no further rule can be applied to $\mathcal{P}'$. Since no further rule can be applied to $\mathcal{P}'$, each constraint in $\mathcal{P}'$ is bounds consistent and so is $\mathcal{P}'$.*

Claim 4: *If the algorithm terminates with failure, then the problem at the moment of failure does not have a solution.*
The algorithm terminates with failure, if either the rule `empty` *of the bounds consistency rule is applied or one of the rules given in Table 5.6 is applied. Each of those rules imply that the domain of at least one variable in $\mathcal{P}$ is empty. By definition, a problem formulated in CL$_{catts}$ has no solution, if one of its participating domains is empty.*

Lemma B.1 and the claims proved in Proposition B.1 imply directly that the constraint propagation algorithm of CL_{catts} is complete.

Appendix C

A Haskell Implementation of Predicate Subtyping in CaTTS

C.1 Auxiliary Data Structures and Functions

The following auxiliary data structures are implemented with CaTTS' type checking algorithms:

- Haskell built-in types like
  `Int`, `Bool`
- Haskell built-in functions over such types
- a data structure for the context that collects variable bindings accessed by the type checking algorithms in a list (where variables have an additional integer argument to determine their position in the context list)
  `Context`

The following auxiliary functions are implemented with CaTTS' type checking algorithms:

- functions to manipulate the context

```
-- adding a Variable binding to the context
addToContext :: Context -> String -> Kind -> Context

-- taking a fresh variable name
pickFreshName :: Context -> String -> String

-- getting the kind of an expression from the context
getKindFromContext :: Context -> Int -> Kind
```

- functions to refer to types and kinds of types

```
    -- test the kind of a type
    hasKindPoint :: Kind -> Bool
    hasKindDuration :: Kind -> Bool
    hasKindInterval :: Kind -> Bool

    -- get the type from a Kind
    getTypeFromKind :: Kind -> Ty
```

C.2 Well-Formedness of CaTTS-DL Calendar Specifications

The algorithm that tests well-formedness of CaTTS-DL calendar specifications is implemented in Haskell. The implementation is a straightforward transcription of the typing relation for CaTTS-DL as given in Table 6.4, p.192 into a recursive algorithm.

C.2.1 Syntax

The data type definitions for CaTTS-DL types and CaTTS-DL object language expressions follow the (abstract) Syntax of CaTTS-DL (cf. Table 6.3, p.189).

```
data Ty                                   -- type expression --
        = RCal
        | Aggregate Ty TEx
        | Select Ty TEx
        | And Ty Ty
        | Or Ty Ty
        | Except Ty Ty
        | Restrict Ty Ty
data Kind                                 -- kinding expression --
        = KPoint Ty
        | KInterval Ty
        | KDuration Ty
        | KInt
        | KBool
data TEx                                  -- CaTTS-DL expression --
        = Abs [TEx] TEx
        | Rel [TEx] TEx
        | Where TEx Ty TEx
        | Part Int Ty
        | Var String Int
        | Duration Int Ty
        | Index TEx
        | Relative TEx Ty MRel Int
        | Conj TEx TEx
        | Disj TEx TEx
        | Alter TEx Ty [(TEx,TEx)]
data MRel = Eqs | Neq | Le | Leq | Ge | Geq
```

C.2.2 Well-Formedness

The algorithm that tests well-formedness of a CaTTS-DL calendar specification departs from the declarative rules given in Table 6.4, p.192 for the following reason. Reconsider the rule (F-And)

$$\frac{\Gamma \vdash \tau_1 \subseteq \tau \quad \Gamma \vdash \tau_2 \subseteq \tau}{\Gamma \vdash (\tau_1 \& \tau_2)_{type}} \quad (\text{F-And})$$

The reason why (F-And) is problematic is that the premise of the rule mentions a type τ which must include both types τ_1 and τ_2 and which is not mentioned in the conclusion of the rule. Thus, reading the rule from bottom to top as it is done by the algorithm, the algorithm should *guess* a type τ and then attempts to show that $\tau_1 \subseteq \tau$ and $\tau_2 \subseteq \tau$. Since inclusion defines a kind of subtype relation, the type τ might be *any* type that includes τ_1 and τ_2. Making the decision algorithmic, and thus, syntax-directed, the least type that is a supertype of τ_1 and τ_2 is computed and the inclusion conditions are tested. This type that is the least common supertype of two types σ and τ is called the *cattsJoin*[1] of the types, written $\sigma \vee \tau$. The computation of such joins is subsequently introduced with CaTTS' subtype algorithm. The rule (F-And) can be now rewritten by

$$\frac{\Gamma \vdash \tau_1 \subseteq \tau \quad \Gamma \vdash \tau_2 \subseteq \tau \quad \tau = \tau_1 \vee \tau_2}{\Gamma \vdash (\tau_1 \& \tau_2)_{type}} \quad (\text{F-And})$$

The same has to be done with the rules (F-Or), (F-Ex), and (F-Res) from Table 6.4, p.192. Now we are able to implement CaTTS' algorithm to test well-formedness of calendar specifications by case on the different syntactic forms of the CaTTS-DL type expressions:

```
wellFormed :: Context -> Ty -> Bool
wellFormed ctx (Aggregate t (Abs [] te)) =          -- (F-Aggr)
  if kindOf ctx te == KPoint t then True
  else error "invalid CaTTS-DL type specification"
wellFormed ctx (Aggregate t (Abs (te1:tes) te)) =   -- (F-Aggr)
  if kindOf ctx te1 == KDuration t
     then wellFormed ctx (Aggregate t (Abs tes te))
  else error "invalid CaTTS-DL type specification"
wellFormed ctx (Aggregate t (Rel [] z)) =           -- (F-Aggr~)
  if kindOf ctx z ==  KInt then True
  else error "invalid CaTTS-DL type specification"
wellFormed ctx (Aggregate t (Rel (te1:tes) z)) =    -- (F-Aggr~)
  if kindOf ctx te1 == KDuration t
     then wellFormed ctx (Aggregate t (Rel tes z))
  else error "invalid CaTTS-DL type specification"
wellFormed ctx (Select t (Where x tx te) =          -- (F-Sel)
  let ctx' = addToContext ctx x tx in
```

[1]Recall that joins in CaTTS might be slightly weaker than ordinary lattice joins (cf. Proposition 3.1).

```
     if kindOf ctx' x == KPoint t && kindOf ctx' te == KBool then True
     else error "invalid CaTTS-DL type specification"
wellFormed ctx (And t1 t2) =                       -- (F-And)
  let t = cattsJoin t1 t2 in inclusion t1 t && inclusion t2 t
wellFormed ctx (Or t1 t2) =                        -- (F-Or)
  let t = cattsJoin t1 t2 in inclusion t1 t && inclusion t2 t
wellFormed ctx (Except t1 t2) =                    -- (F-Ex)
  let t = cattsJoin t1 t2 in inclusion t1 t && inclusion t2 t
wellFormed ctx (Restrict t1 t2) =                  -- (F-Res)
  let t = cattsJoin t1 t2 in aggregation t1 t && inclusion t2 t
wellFormed _ _ = False
```

The function that infers the types (or to be more precise, the kinds) of the CaTTS-DL expressions used to define CaTTS-DL types in a calendar specification is defined by case on the syntactic forms of the expressions, simply reading the rules specified in Table 6.5, p.192 “from bottom to top”:

```
kindOf :: Context -> TEx -> Kind
kindOf ctx (Part i t) = KPoint t                 -- (T-Part)
kindOf ctx (Var _ i) = getKindFromContext ctx i -- (T-Var)
kindOf ctx (Duration i t) = KDuration t         -- (T-Dur)
kindOf ctx (Index te) =                          -- (T-Index)
  if hasKindPoint (kindOf ctx te) then KInt
  else error "invalid CaTTS-DL expression"
kindOf ctx (Relative te t _ i) =                 -- (T-Rel)
  if kindOf ctx te == KInt && wellFormed ctx t then KBool
  else error "invalid CaTTS-DL expression"
kindOf ctx (Alter x t ((te1,te2):tes))           -- (T-Alter)
  | hasKindDuration t1 && kindOf ctx' te1 == KBool = kindOf' ctx' tes
  | otherwise error "invalid CaTTS-DL expression"
    where
    ctx' = addToContext x t
    t1 = KindOf ctx' te2
    kindOf' ctx' [] = t1
    kindOf' ctx' tes =
      if kindOf ctx' (fst (head tes)) == KBool &&
         kindOf ctx' (snd (head tes)) == t1
      then kindOf' ctx tail tes
      else error "invalid CaTTS-DL expression"
kindOf ctx (Conj te1 te2) =                      -- (T-Conj)
  if kindOf ctx te1 == KBool && kindOf ctx te2 == KBool then KBool
  else error "invalid CaTTS-DL expression"
kindOf ctx (Disj te1 te2) =                      -- (T-Disj)
  if kindOf ctx te1 == KBool && kindOf ctx te2 == KBool then KBool
  else error "invalid CaTTS-DL expression"
kindOf _ _ = error "invalid CaTTS-DL expression"
```

The algorithm that tests aggregation (resp. inclusion) between two calendric type definitions is given in the following. It implements the rules given in Table 6.6, p.193 (resp.

those given in Table 6.7, p.194). Since the rules (AS-Refl) and (AS-Trans) (resp. (IS-Refl) and (IS-Trans)) are not syntax-directed, a naive "bottom to top" implementation of these rules would never know whether to try one of these rules or not. Therefore, reflexivity and transitivity are incorporated into each of the syntactic rules which can be then read from bottom to top by the algorithm.

```
aggregation :: Ty -> Ty -> Bool
aggregation RCal RCal = True
aggregation (Aggregate s1 _) t                          -- (AS-Aggr)
     | s1 == t = True
     | otherwise = aggregation s1 t
aggregation (Restrict s1 s2) t                          -- (AS-Res)
     | s2 == t = True
     | otherwise = aggregation s2 t
aggregation _ _ = False

inclusion :: Ty -> Ty -> Bool
inclusion RCal RCal = True
inclusion (Select s1 _) t                               -- (IS-Sel)
     | s1 == t = True
     | otherwise = inclusion s1 t
inclusion s (Or t1 t2)                                  -- (IS-DJ1)
     | inclusion s t1 = True
     | otherwise = subtype s t2
inclusion (Or s1 s2) t                                  -- (IS-DJ2)
     | inclusion s1 t = inclusion s2 t
     | otherwise = False
inclusion (And s1 s2) t                                 -- (IS-CJ1)
     | inclusion s1 t = True
     | otherwise = inclusion s2 t
inclusion s (And t1 t2)                                 -- (IS-CJ2)
     | inclusion s t1 = subtype s t2
     | otherwise = False
inclusion (Except s1 s2) t                              -- (IS-Ex)
     | inclusion s1 t = True
     | otherwise = False
inclusion _ _ = False
```

C.3 Typing and Subtyping in CaTTS-CL

C.3.1 Syntax

The syntax extends the Haskell syntax for CaTTS-DL type expressions and object language expressions (cf. Appendix C.2) with data type definitions for CaTTS-CL object language expressions follow the syntax of CaTTS-CL as given in Table 6.8, p.198.

```
data CEx                          -- CaTTS-CL expression --
          = Part Int Ty
          | Var String Int Dom
          | CETrue
          | CEFalse
          | Duration Int Ty
          | EnpI CEx CEx
          | DurI TO CEx CEx
          | IsEvent String Ty CEx
          | IsTask String Ty CEx
          | IntervalC Rel CEx CEx
          | MetricC MRel CEx CEx
          | Dur CEx
          | End BE CEx
          | Index CEx
          | Shift FB CEx CEx
          | ExSh ES CEx CEx
          | Relative CEx Ty MRel Int
          | Conj CEx CEx
data TO = Upto | Downto
data Rel = Equals | Before | After | Starts | StartedBy | Finishes
           | FinishedBy | Meets | MetBy | Overlaps | OverlappedBy
           | Contains | During | OnOrBefore | OnOrAfter | Within
data MRel = Eqs | Neq | Le | Leq | Ge | Geq
data BE = Begin | End
data FB = Forward | Backward
data ES = Extend | Shorten

data LEx                             -- CL_catts expression --
         = CL CEx
         | ConvC CEx CEx
```

C.3.2 Subtyping

CaTTS-CL's subtype checker is given in the following. It implements the rules given in Table 6.9. Since the rules (S-Refl) and (S-Trans) are not syntax-directed, a naive "bottom to top" implementation of these rules would never know whether to try one of these rules or not. Therefore, reflexivity and transitivity are incorporated into each of the rules given in Table 6.9 whenever transitivity is used to past together different subtyping rules. The syntax-directed rules have the following forms:

$$\frac{\sigma^* \leq \tau^*}{\sigma \leq \tau^*} \quad (\text{S-ICoer}) \qquad \sigma \leq r_{\mathfrak{C}} \quad (\text{S-Ref})$$

$$\frac{\sigma \leq \tau}{(\mathit{select}\ \sigma\ X : \sigma\ \mathit{where}\ te) \leq \tau} \quad (\text{S-Sel}) \qquad \frac{\tau_1 \leq \tau}{\tau_1 \setminus \sigma_1 \leq \tau} \quad (\text{S-Ex})$$

$$\frac{\exists i \in \{1,2\}, \sigma \leq \tau_i}{\sigma \leq \tau_1 \mid \tau_2} \quad (\text{S-Dj}_1) \qquad \frac{i \in \{1,2\},\ \sigma_i \leq \tau}{\sigma_1 \mid \sigma_2 \leq \tau} \quad (\text{S-Dj}_2)$$

$$\frac{\forall i \in \{1,2\},\ \tau_i \leq \tau}{\tau_1 \& \tau_2 \leq \tau} \quad (\text{S-Cj}_1) \qquad \frac{\sigma \leq \tau_i}{\sigma \leq \tau_1 \& \tau_2} \quad (\text{S-Cj}_2)$$

$$\frac{\sigma \leq \tau}{(\mathit{aggregate}\ \sigma\ te_i^{i \in 1..n}\ \tilde{}@te) \leq \tau} \quad (\text{S-Aggr}) \qquad \frac{\sigma \leq \tau}{(\mathit{aggregate}\ \sigma\ te_i^{i \in 1..n}\ \tilde{}@z) \leq \tau} \quad (\text{S-Aggr}\tilde{})$$

$$\frac{\tau_2 \leq \tau}{\tau_1 \# < \tau_2 \leq \tau} \quad (\text{S-Res}) \qquad \frac{\sigma \leq \tau}{\sigma^* \leq \tau^*} \quad (\text{S-Int})$$

Lemma C.1

1. *If $\forall \alpha :: K.\, \alpha \leq \alpha$ is derivable using* (S-Refl) *and the declarative rules from Table 6.9, p.200, then it is derivable without* (S-Refl) *using the rules given above.*

2. *If $\forall \alpha, \beta :: K.\, \alpha \leq \beta$ is derivable using* (S-Trans) *and the declarative rules from Table 6.9, p.200, then it is derivable without* (S-Trans) *using the rules given above.*

Proof C.1 (sketched)
Part (1): Follows directly from the structure of α.

Part (2): With (1), if there is any derivation of $\alpha \leq \beta$, then there is a reflexivity-free one. By induction on the structure of the final rule in the derivation of $\alpha \leq \beta$.
Induction hypothesis: all the sub-derivations of the final rule can be replaced by derivations not involving transitivity.
Case (1): The final rule in the derivation is anything other than (S-Trans)*, then the result follows directly by induction hypothesis; since the final rule is transitivity-free either, the whole derivation is now transitivity free.*
Case(2): Suppose that the final rule is (S-Trans)*, i.e. there are sub-derivations $\alpha \leq \gamma$ and $\gamma \leq \beta$ for some γ. By case on the final rules in both of these sub-derivations:*

(S-Sel)/(S-Ref)	$\beta = r_{\mathcal{C}}$
(S-Ex)/(S-Ref)	$\beta = r_{\mathcal{C}}$
(S-Dj$_1$)/(S-Ref)	$\beta = r_{\mathcal{C}}$
(S-Dj$_2$)/(S-Ref)	$\beta = r_{\mathcal{C}}$
(S-Cj$_1$)/(S-Ref)	$\beta = r_{\mathcal{C}}$
(S-Cj$_2$)/(S-Ref)	$\beta = r_{\mathcal{C}}$
(S-Aggr)/(S-Ref)	$\beta = r_{\mathcal{C}}$
(S-Aggr˜)/(S-Ref)	$\beta = r_{\mathcal{C}}$
(S-Res)/(S-Ref)	$\beta = r_{\mathcal{C}}$

The result is immediate since $\sigma \leq r_{\mathcal{C}}$ can be derived as long as σ is any basic (point) type. The other combinations with (S-Ref) *are not possible, since they place incompatible constraints on the form of γ.*

(S-Ref)/*Any* $\gamma = r_{\mathcal{C}}$

The derivation $\alpha \leq \gamma$ ends with $r_{\mathcal{C}}$; $\gamma \leq \beta$ is transitivity-free by induction hypothesis. By structure the final rule must be (S-Ref)*, since* (S-Refl) *has been already eliminated.*

(S-ICoer)/(S-Int) $\alpha = \sigma,\ \gamma = \sigma^*,\ \beta = \tau^*$
$\sigma \leq \sigma^*,\ \sigma^* \leq \tau^*$

Using (S-Trans)*, the derivation $\sigma \leq \tau^*$ can be constructed from the given subderivations which can be by induction hypothesis replaced by transitivity-free derivations. Examining the structure of this derivation, σ has a (basic) point type and τ^* an interval type, i.e. the final rule must be* (S-Coer)*; thus obtaining a transitivity-free derivation of $\sigma \leq \tau^*$.*

All other cases (for possible combinations of subtyping rules, i.e. rules that place possible constraints on γ) are proceeded similar.

From this lemma, it directly follows that the syntax-directed subtyping rules are *sound* and *complete*: every subtyping statement that can be derived from the syntax-directed subtyping rules can be derived from the declarative rules given in Table 6.9, p.200. And every subtyping statement that can be derived from the declarative rules can also be derived from the syntax-directed subtyping rules.

The syntax-directed subtyping rules can be now read from "bottom to top", yielding the following subtyping algorithm.

```
subtype :: Kind -> Kind -> Bool
subtype (KPoint _)  (KPoint RCal) = True                -- (S-Ref)
subtype (KPoint s) (KInterval t)                        -- (S-ICoer)
     | s == t   = True
     | otherwise = subtype (KInterval s) (KInterval t)
subtype (KPoint (Select s1 _)) (KPoint t)               -- (S-Sel)
     | s1 == t = True
     | otherwise = subtype (KPoint s1) (KPoint t)
subtype (KPoint s) (Or (KPoint t1) (KPoint t2))         -- (S-DJ1)
     | subtype (KPoint s) (KPoint t1) = True
     | otherwise = subtype (KPoint s) (KPoint t2)
subtype (Or (KPoint s1) (KPoint s2)) (KPoint t)         -- (S-DJ2)
```

```
    | subtype (KPoint s1) (KPoint t) = subtype (KPoint s2) (KPoint t)
    | otherwise = False
subtype (And (KPoint s1) (KPoint s2)) (KPoint t)       -- (S-CJ1)
    | inclSubtype (KPoint s1) (KPoint t) = True
    | otherwise = subtype (KPoint s2) (KPoint t)
subtype (KPoint s) (And (KPoint t1) (KPoint t2))       -- (S-CJ2)
    | subtype (KPoint s) (KPoint t1) = subtype (KPoint s) (KPoint t2)
    | otherwise = False
subtype (Except (KPoint s1) (KPoint s2)) (KPoint t)    -- (S-Ex)
    | subtype (KPoint s1) (KPoint t) = True
    | otherwise = False
subtype (Aggregate (KPoint s1) _) (KPoint t)           -- (S-Aggr)
    | s1 == t = True
    | otherwise = subtype (KPoint s1) (KPoint t)
subtype (Restrict (KPoint s1) (KPoint s2)) (KPoint t) -- (S-Res)
    | s2 == t = True
    | otherwise = subtype (KPoint s2) (KPoint t)
subtype (KInterval s) (KInterval t) =                  -- (S-Int)
    subtype (KPoint s) (KPoint t)
subtype _ _ = False
```

C.3.3 Typing

The definitions of the declarative typing relation for CaTTS-CL in Table 6.10, p.203 is not immediately suitable for implementation. The relation is not syntax-directed, i.e. it cannot just be read reversely (from bottom to top) to yield a type checking algorithm. The reason for this is the typing rule of subsumption (T-Sub). The reason why this rule is unsuitable for an implementation of a type checker is that the expression in the conclusion *ce* is a bare metavariable:

$$\frac{\forall\alpha,\beta :: K \quad \Gamma \vdash ce : \alpha \quad \alpha \leq \beta}{\Gamma \vdash ce : \beta} \quad (\text{T-Sub})$$

That is, this rule might be applied in each step of a derivation and the type checker would never know when to use it. Every other typing rule (in Table 6.10) specifies an expression of a specific syntactic form.

To yield a syntax-directed typing relation that can be implemented straightforwardly from bottom to top, we must consider each of the typing rules in Table 6.10 and figure out where (T-Sub) could be used. These rules then need some further conditions to be tested in the premises resulting from (T-Sub). The rules (T-EnpI) and (T-Interval) need to be modified:

$$\frac{\Gamma \vdash ce_1 : \tau_1 \quad \Gamma \vdash ce_2 : \tau_1}{\Gamma \vdash [ce_1..ce_2] : \tau_1^*} \quad (\text{T-EndpI}) \qquad \frac{\Gamma \vdash ce_1 : \tau_1^* \quad \Gamma \vdash ce_2 : \tau_1^*}{\Gamma \vdash ce_1 \,\mathcal{R}\, ce_2 : \mathbb{B}} \quad (\text{T-Interval})$$

Examining derivations for typing proofs of some typing statement $\forall\alpha :: K.\, \Gamma \vdash ce : \alpha$, the use of subsumption can always be moved to the end of the resulting derivation tree. Therefore, the subsumption rule can be dropped and the rules (T-EndpI) and (T-Interval) are replaced with slightly more powerful rules of the form

$$\frac{\Gamma \vdash ce_1 : \tau_1 \;\; \Gamma \vdash ce_2 : \tau_2 \; \tau_1 \leq \tau, \tau_2 \leq \tau}{\Gamma \vdash [ce_1..ce_2] : \tau^*} \qquad \text{(T'-EndpI)}$$

$$\frac{\forall \alpha :: \tau_1 \mid \alpha :: \tau_1^* \; and \; \forall \beta :: \tau_2 \mid \beta :: \tau_2^* \;\; \Gamma \vdash ce_1 : \alpha \;\; \Gamma \vdash ce_2 : \beta \;\; \alpha, \beta \leq \iota}{\Gamma \vdash ce_1 \; \mathcal{R} \; ce_2 : \mathbb{B}} \qquad \text{(T'-Interval)}$$

incorporating a single instance of the subsumption rule in the premises.

Unfortunately, those two rules are still not syntax-directed, since the types τ and ι do not appear in the conclusion of the rules. Thus, the type checker has to guess some type τ (resp. ι) and then attempts to show that for those types the subtyping statement holds. For example, if in rule (T-Interval) ce_1 : *working_week* and ce_2 : *month*, then ι might be *day**, or *hour**, or even $r_{\mathbb{C}}^*$; in fact, *any* type that is a supertype of both *working_week* and *month*. To make a clear decision in any case, CaTTS' type checker is always looking for some "minimal" type that is the supertype of the types of the both expressions. This minimal type is in fact the *join* (according to Proposition 3.1) of the two types τ_1 and τ_2. That such a join always exists and that it is unique for any pair of calendric types defined in some (finite set of aligned) CaTTS-DL calendar specification(s) is shown in Proposition 3.1. For types α_1 and α_2, the join τ is denoted by $\alpha_1 \vee \alpha_2$. We can now replace the rules for (T-Interval) and (T-EndpI) by the following rules:

$$\frac{\Gamma \vdash ce_1 : \tau_1 \;\; \Gamma \vdash ce_2 : \tau_2 \; \tau_1 \vee \tau_2 = \tau}{\Gamma \vdash [ce_1..ce_2] : \tau^*} \qquad \text{(AT-EndpI)}$$

$$\frac{\forall \alpha :: \tau_1 \mid \alpha :: \tau_1^* \; and \; \forall \beta :: \tau_2 \mid \beta :: \tau_2^* \;\; \Gamma \vdash ce_1 : \alpha \;\; \Gamma \vdash ce_2 : \beta \;\; \alpha \vee \beta = \gamma}{\Gamma \vdash ce_1 \; \mathcal{R} \; ce_2 : \mathbb{B}} \qquad \text{(AT-Interval)}$$

The syntax-directed subtyping rules are then the subtyping rules given in Table 6.10, p.203 without the rule (T-Sub) where the rules (T-Interval) and (T-EndpI) are replaced by the rules (AT-Interval) and (AT-EndpI).

The discussion above provides with an informal proof that the syntax-directed typing relation corresponds to the original declarative rules given in Table 6.10, p.203. In particular, it states that the syntax-directed typing relation is both *sound* and *complete* with respect to the original declarative rules from Table 6.10. The formal proof for soundness of the type checker goes by straightforward induction on the structure of the final rule of the syntax-directed typing derivation. The formal proof for completeness (also called minimal typing) goes by induction on the declarative typing relation. It is also proceed by cases on the final rule in the derivation, applying Lemma C.1.

Note: The subtyping rule for durations is not implemented with the syntax-directed typing relation as already discussed in Section 7.2.1.3.

Now, we can implement the type checker straightforwardly in Haskell by reading the syntax-directed typing rules from “bottom to top”, giving a pattern matching rule for each inference rule.

```
typeOf :: Context -> CEx -> Kind
typeOf ctx (Part i t) = t                                   -- (T-Const)
typeOf ctx (Var _ i _) ctx = getKindFromContext ctx i      -- (T-Var)
typeOf ctx (Duration i t) = KDuration t                     -- (T-Dur)
typeOf ctx (Index ce)                                       -- (T-Index)
    | hasKindPoint (typeOf ctx ce) = KInt
    | otherwise = error "invalid CaTTS-CL expression"
typeOf ctx (Dur ce)                                         -- (T-Duration)
    | hasKindPoint t || hasKindInterval t = KDuration (getTypeFromKind t)
    | otherwise = error "invalid CaTTS-CL expression"
      where
      t = typeOf ctx ce
typeOf ctx (End _ ce)                                       -- (T-EndP)
    | hasKindPoint t || hasKindInterval t = KPoint (getTypeFromKind t)
    | otherwise = error "invalid CaTTS-CL expression"
      where
      t = typeOf ctx ce
typeOf ctx (EndpI ce1 ce2)                                  -- (AT-EndpI)
    | hasKindPoint t1 && hasKindPoint t2
      = KInterval (cattsJoin (getTypeFromKind t1) (getTypeFromKind t2))
    | otherwise = error "invalid CaTTS-CL expression"
      where
      t1 = typeOf ctx c1
      t2 = typeOf ctx c2
typeOf ctx (DurI _ ce1 ce2)                                 -- (T-DurI)
    | hasKindDuration t1 && hasKindPoint t2 &&
      getTypeFromKind t1 == getTypeFromKind t2   = KInterval t2
    | otherwise = error "invalid CaTTS-CL expression"
      where
      t1 = typeOf ctx ce1
      t2 = typeOf ctx ce2
typeOf ctx (Shift _ ce1 ce2)                                -- (T-Shift)
     | (hasKindPoint t1 || hasKindInterval t1) && hasKindDuration t2 &&
       getTypeFromKind t1 == getTypeFromKind t2   = t1
     | otherwise  = error "invalid CaTTS-CL expression"
      where
      t1 = typeOf ctx ce1
      t2 = typeOf ctx ce2
typeOf ctx (ExSh _ ce1 ce2)                                 -- (T-ExSh)
     | (hasKindPoint t1 || hasKindInterval t1) && hasKindDuration t2 &&
       getTypeFromKind t1 == getTypeFromKind t2   = t1
     | otherwise  = error "invalid CaTTS-CL expression"
      where
      t1 = typeOf ctx ce1
      t2 = typeOf ctx ce2
typeOf ctx CETrue = KBool                                   -- (T-True)
typeOf ctx CEFalse = KBool                                  -- (T-False)
typeOf ctx (Conj (IsEvent x t ce) ce2)                      -- (T-Event)
   let ctx' = addToContext ctx x (KPoint t)
   in typeOf ctx' ce2
typeOf ctx (Conj (IsTask x t ce)  ce2)                      -- (T-Task)
   let ctx' = addToContext ctx x (KInterval t)
   in typeOf ctx' ce2
typeOf ctx (IntervalC _ ce1 ce2)                            -- (AT-Interval)
   | (haskindPoint t1 || hasKindInterval t1) &&
     (haskindPoint t2 || hasKindInterval t2) &&
     subtype t1 (KInterval (cattsJoin (getTypeFromKind t1)
                 (getTypeFromKind t2))) &&
     subtype t2 (KInterval (cattsJoin (getTypeFromKind t1)
                 (getTypeFromKind t2))) = KBool
   | otherwise = error "invalid CaTTS-CL expression"
     where
     t1 = typeOf ctx ce1
     t2 = typeOf ctx ce2
typeOf ctx (MetricC _ ce1 ce2)                              -- (T-Metric)
   | hasKindDuration t1 && hasKindDuration t2 &&
     getTypeFromKind t1 == getTypeFromKind t2 = KBool
   | otherwise = error "invalid CaTTS-CL expression"
     where
     t1 = typeOf ctx ce1
     t2 = typeOf ctx ce2
typeOf ctx (Relative ce t _ i)                              -- (T-Rel)
   | typeOf ctx ce == KInt && wellFormed ctx t = KBool
   | otherwise = error  "invalid CaTTS-CL expression"
typeOf ctx (Conj ce1 ce2)                                   -- (T-Conj)
  | typeOf ctx ce1 == KBool && typeOf ctx ce2 == KBool = KBool
  | otherwise =  error  "invalid CaTTS-CL expression"
```

And the function `cattsJoin` is defined as follows on the syntactic forms of CaTTS-DL type constructors:

```
cattsJoin :: Ty -> Ty -> Ty
cattsJoin s t
    | subtype s t = t
    | subtype t s = s
    | otherwise = syntaxCheck s t

syntaxCheck :: Ty -> Ty -> Ty
syntaxCheck (Aggregate s1 _) (Aggregate t2 _) = cattsJoin s1 t2
syntaxCheck (Aggregate s1 _) (Restrict t1 t2) = cattsJoin s1 t2
syntaxCheck (Aggregate s1 _) (Select t1 _) = cattsJoin s1 t1
syntaxCheck (Aggregate s1 _) (And t1 t2)
   | subtype j1 j2 = j1
   | otherwise = j2
     where
     j1 = cattsJoin s1 t1
     j2 = cattsJoin s1 t2
syntaxCheck (Aggregate s1 _) (Or t1 t2) = cattsJoin s1 t1
syntaxCheck (Aggregate s1 _) (Except t1 t2) = cattsJoin s1 t1
syntaxCheck (Restrict s1 s2) (Restrict t1 t2) = cattsJoin s2 t2
syntaxCheck (Restrict s1 s2) (Aggregate t1 _) = cattsJoin s2 t1
syntaxCheck (Restrict s1 s2) (Select t1 _) = cattsJoin s1 t1
syntaxCheck (Restrict s1 s2) (And t1 t2)
    | subtype j1 j2 = j1
    | otherwise = j2
      where
      j1 = (cattsJoin s1 t1)
      j2 = (cattsJoin s1 t2)
syntaxCheck (Restrict s1 s2) (Or t1 t2) =  cattsJoin s1 t1
syntaxCheck (Restrict s1 s2) (Except t1 t2) = cattsJoin s1 t1
syntaxCheck (Select s1 _) (Select t2 _) = cattsJoin s1 t2
syntaxCheck (Select s1 _) (Aggregate t1 _) = cattsJoin s1 t1
syntaxCheck (Select s1 _) (Restrict t1 t2) = cattsJoin s1 t1
syntaxCheck (Select s1 _) (And t1 t2)
    | subtype j1 j2 = j1
    | otherwise = j2
      where
      j1 = (cattsJoin s1 t1)
      j2 = (cattsJoin s1 t2)
syntaxCheck (Select s1 _) (Or t1 t2) =  cattsJoin s1 t1
syntaxCheck (Select s1 _) (Except t1 t2) = cattsJoin s1 t1
syntaxCheck (And s1 s2) (Aggregate t1 _)
    | subtype j1 j2 = j1
    | otherwise = j2
      where
      j1 = (cattsJoin s1 t1)
      j2 = (cattsJoin s2 t1)
syntaxCheck (And s1 s2) (Restrict t1 t2)
    | subtype j1 j2 = j1
    | otherwise = j2
      where
      j1 = (cattsJoin s1 t1)
      j2 = (cattsJoin s2 t1)
syntaxCheck (And s1 s2) (Select t1 _)
    | subtype j1 j2 = j1
    | otherwise = j2
      where
      j1 = (cattsJoin s1 t1)
      j2 = (cattsJoin s2 t1)
syntaxCheck (And s1 s2) (Or t1 t2)
    | subtype j1 j2 = j1
    | otherwise = j2
      where
      j1 = (cattsJoin s1 t1)
      j2 = (cattsJoin s2 t1)
syntaxCheck (And s1 s2) (Except t1 t2)
    | subtype j1 j2 = j1
    | otherwise = j2
      where
      j1 = (cattsJoin s1 t1)
      j2 = (cattsJoin s2 t1)
syntaxCheck (And s1 s2) (And t1 t2)
   = cattsJoin j1 j2
      where
      j1 = cattsJoin s1 t1
      j2 = cattsJoin s2 t2
syntaxCheck (Or s1 s2) (Aggregate t1 _) = cattsJoin s1 t1
syntaxCheck (Or s1 s2) (Restrict t1 t2) = cattsJoin s1 t1
syntaxCheck (Or s1 s2) (Select t1 _) = cattsJoin s1 t1
syntaxCheck (Or s1 s2) (And t1 t2)
    | subtype j1 j2 = j1
    | otherwise = j2
      where
      j1 = (cattsJoin s1 t1)
      j2 = (cattsJoin s1 t2)
syntaxCheck (Or s1 s2) (Except t1 t2) = cattsJoin s1 t1
syntaxCheck (Or s1 s2) (Or t1 t2) = cattsJoin s1 t1
syntaxCheck (Except s1 s2) (Aggregate t1 _) = cattsJoin s1 t1
syntaxCheck (Except s1 s2) (Restrict t1 t2) = cattsJoin s1 t1
syntaxCheck (Except s1 s2) (Select t1 _) = cattsJoin s1 t1
syntaxCheck (Except s1 s2) (And t1 t2)
    | subtype j1 j2 = j1
    | otherwise = j2
      where
      j1 = (cattsJoin s1 t1)
      j2 = (cattsJoin s1 t2)
syntaxCheck (Except s1 s2) (Or t1 t2) = cattsJoin s1 t1
syntaxCheck (Except s1 s2)(Except t1 t2) = cattsJoin s1 t1
```

C.3.4 Coercion

The coercion is a straightforward implementation of the subtype function given above: for each subtyping rule a coercion according to Definition 6.3 is performed. As an example, the coercion for the rule (S-SEL) is given.

```
coercion :: Context -> Kind -> Kind -> CEx -> LEx
coercion ctx (KPoint (Select s1 _)) (KPoint t) ce
  | s1 == t = ConvC x' ce
  | subtype (KPoint s1) (KPoint t)
      = CL (Conj (COnvC x' ce) (coercion ctx' s1 t x'))
  | otherwise = error "invalid CaTTS-CL expression"
    where
    x' = pickFreshName ctx ce
    ctx' = addToContext ctx x' (Kpoint s1)
```

C.3.5 Transformation

The transformation is a straightforward implementation of the CaTTS-CL typing function given above: for each typing rule a transformation according to Definition 6.4 is preformed. As an example, the transformation for the rule (T-INTERVAL) is given.

```
transform :: Context -> CEx -> LEx
transform ctx (Interval _ ce1 ce2) =
 CL (Conj (Interval _ x y) (Conj (coercion ctx'' t1 j ce1)
                                  (coercion ctx'' t2 j ce2)))
     where
     t1 = typeOf ctx ce1
     t2 = typeOf ctx ce2
     j = cattsJoin t1 t2
     x = pickFreshName ctx ce1
     ctx' = addToContext ctx x j
     y = pickFreshName ctx' ce2
     ctx'' = addtToContext ctx' y j
```

Bibliography

[AAM05] J.J. Alferes, R. Amador, and W. May. A General Language for Evolution and Reactivity on the Semantic Web. In *Proceedings of Workshop on Principles and Practices in Semantic Web Reasoning*, LNCS 3703, pages 101–115. Springer-Verlag, 2005.

[ABB04] J. J. Alferes, F. Banti, and A. Brogi. Well Founded Semantics for Logic Program Updates. In *Proceedings of the 9th Ibero-American Conference on Artificial Intelligence, Puebla, Mexico*, LNCS 3315, pages 397–407. Springer-Verlag, 2004.

[ABH$^+$02] A. Ankolekar, M.H. Burstein, J.R. Hobbs, O. Lassila, D.L. Martin, S.A. McIlraith, S. Narayanan, M. Paolucci, T.R. Payne, K.P. Sycara, and H. Zeng. DAML-S: Web Service Description for the Semantic Web. In *Proceedings of the 1st International Semantic Web Conference*, LNCS 2342, pages 411–430. Springer-Verlag, 2002.

[ABS00] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web – From Relations to Semistructured Data and XML.* Morgan Kaufmann Publishers, 2000.

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects.* Springer-Verlag, 1996.

[AC01] D. Aspinall and A. Compagnoni. Subtyping Dependent Types. *Information and Computation*, 266(1–2):273–309, 2001.

[AK94] L. Al-Khatib. *Reasoning with Non-convex Intervals.* PhD Thesis, Florida Institute of Technology, 1994.

[All83] J.F. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[All91] J. Allen. Planning as Temporal Reasoning. In *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning*, pages 3–14. Morgan Kaufmann Publishers, 1991.

[And83] T. Anderson. Modeling Events and Processes at the Conceptual Level. In *Proceedings of the 2nd International Conference on Databases*, pages 151–168. Wiley Heyden Ltd., 1983.

[Apt03] K.R. Apt. *Principles of Constraint Programming.* Cambridge University Press, 2003.

[Asp94] D. Aspinall. Subtyping with Singleton Types. In *Proceedings of Computer Science Logic*, LNCS 933, pages 1–15. Springer-Verlag, 1994.

[AYU01] T. Amagasa, M. Yoshikawa, and S. Uemura. Realizing Temporal XML Repositories Using Temporal Relational Databases (Poster). In *Proceedings of the 3rd International Symposium on Cooperative Database Systems for Advanced Applications.* IEEE Computer Society, 2001.

[BAF98] L. Bertossi, M. Arenas, and C. Ferretti. SCDBR: An Automated Reasoner for Specifications of Database Updates. *Journal of Intelligent Information Systems*, 10(3):235–280, 1998.

[Bak91] A. Baker. Non-monotonic Reasoning in the Framework of the Situation Calculus. *Artificial Intelligence*, 49(1-3):5–23, 1991.

[BBL+04] S. Berger, F. Bry, B. Lorenz, H.J. Ohlbach, P. Patranjan, S. Schaffert, U. Schwertel, and S. Spranger. Reasoning on the Web: Language Prototypes and Perspectives. In *Proceedings of the European Workshop on the Integration of Knowledge, Semantics and Digital Media Technology*, 2004.

[BCDE00] G. Becher, F. Cléin-Debart, and P. Enjalbert. A Qualitative Model for Time Granularity. *Computational Intelligence*, 16(2):138–169, 2000.

[BCF+98] W. Burgard, A.B. Cremers, D. Fox, D. Hähnel, G. Lakemeyer, D. Schulz, W. Steiner, and S. Thrun. The Interactive Museum Tour-guide Robot. In *Proceedings of the* 15th *National Conference on Artificial Intelligence/10th Conference on Artificial intelligence/Innovative Applications of Artificial Intelligence*, pages 11–18. American Association for Artificial Intelligence, 1998.

[BCM03] F. Baader, D. Calvanese, and D. McGuinness. *The Description Logic Handbook: Theory, Implementation, and Applications.* Cambridge University Press, 2003.

[BCP01] A. Bonifati, S. Ceri, and S. Paraboschi. Active Rules for XML: A new Paradigm for e-Services. *VLDB Journal*, 10(1):39–47, 2001.

[BDD95] K.van Belleghem, M. Denecker, and D. DeSchreye. Combining Situation Calculus and Event Calculus. In *Proceedings of the International Conference on Logic Programming, MIT Press*, pages 83–97, 1995.

[be96] Shibuyakuritsu Shoto bijutsukan (ed.). *Mojie to emoji no keifu, Tōkiō.* 1996.

[Beu04] J. De Beule. Creating Temporal Categories for an Ontology of Time. VUB, ARTI-lab, 2004.

[BHRS05] F. Bry, J. Haußer, F.-A. Rieß, and S. Spranger. Cultural Calendars for Programming and Querying. In *Proceedings of the 1st Forum on the Promotion of European and Japanese Culture and Traditions in Cyber Society and Virtual Reality*, 2005.

[BJW00] C. Bettini, S. Jajodia, and S.X. Wang. *Time Granularities in Databases, Data Mining, and Temporal Reasoning.* Springer-Verlag, 2000.

[BLOS03] F. Bry, B. Lorenz, H.J. Ohlbach, and S. Spranger. On Reasoning on Time and Location on the Web. In *Proceedings of the* 1st *International Workshop on Principles and Practice of Semantic Web Reasoning*, LNCS 2901, pages 102–117. Springer-Verlag, 2003.

[BLS05] F. Bry, B. Lorenz, and S. Spranger. Calendars and Topologies as Types – A Programming Language Approach to Modeling Mobile Applications. In *Proceedings of 9th International Conference on Knowledge-Based Intelligent Information and Engineering Systems*, LNCS 3684, pages 352–358. Springer-Verlag, 2005.

[BM05] F. Bry and M. Marchiori. Ten Theses on Logic Languages for the Semantic Web. In *Proceedings of the W3C Workshop on Rule Languages for Interoperability, USA*, 2005.

[Boc90] A. Bochman. Concerted Instance-interval Temporal Semantics: Temporal Ontologies. *Notre Dame Journal of Formal Logic*, 31(3):403–414, 1990.

[BRS05] F. Bry, F.-A. Rieß, and S. Spranger. CaTTS: Calendar Types and Constraints for Web Applications. In *Proceedings of the 14th International World Wide Web Conference*, pages 702–711. ACM Press, 2005.

[BRST00] C. Boutilier, R. Reiter, M. Soutchanski, and S. Thrun. Decision-theoretic, High-level Agent Programming in the Situation Calculus. In *Proceedings of the 17th National Conference on Artificial Intelligence and 12th Conference on on Innovative Applications of Artificial Intelligence,*, pages 355–362. AAAI Press / The MIT Press, 2000.

[Bru72] B. Bruce. A Model for Temporal References and its Applications in a Question Answering Program. *Artificial Intelligence*, 4:1–25, 1972.

[BS03] F. Bry and S. Spranger. Temporal Constructs for a Web Language. In *Proceedings of the* 4th *Workshop on Interval Temporal Logics and Duration Calculi (ESSLLI)*, 2003.

[BS04] F. Bry and S. Spranger. Towards a Multi-calendar Temporal Type System for (Semantic) Web Query Languages. In *Proceedings 2nd International Workshop Principles and Practice in Semantic Web Reasoning*, LNCS 3208, pages 69–83. Springer-Verlag, 2004.

[BTCGS91] V. Breazu-Tannen, T. Coquand, C. Gunter, and A. Scedrov. Inheritance as Implicit Coercion. *Interval Computations*, 93(1):172–221, 1991.

[CAB$^+$84] R.L. Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, J.T. Sasaki, and S.F. Smith. *Implementing Mathematics with the NuPRL Proof Development System*. Prentice-Hall, Englewoof Cliffs, 1984.

[CAM02] G. Cobena, S. Abiteboul, and A. Marian. Detecting Changes in XML Documents. In *Proceedings of the 18th International Conference on Data Engineering*, pages 41–52. IEEE Computer Society, 2002.

[Car88] L. Cardelli. Typechecking Dependent Types and Subtypes. In *Foundations of Logic and Functional Programming, Workshop Proceedings*, LNCS 306, pages 45–57. Springer-Verlag, 1988.

[Car96] L. Cardelli. Type Systems. In *A.B. Tucker (ed.). Handbook of Computer Science and Engineering*. CRC Press, 1996.

[CCM95] I. Cervesato, L. Chittaro, and A. Montanari. A Modal Calculus of Partially Ordered Events in a Logic Programming Framework. In *Proceedings of the* 12th *International Conference on Logic Programming*, pages 299–313. MIT Press, 1995.

[CEMP93] E. Ciapessoni, E.Corsetti, A. Montanari, and P. San Pietro. Embedding Time Granularity in a Logical Specification Language for Synchronous Real-time Systems. *Science of Computer Programming*, 20(1–2):141–171, 1993.

[CFP04] C. Combi, M. Franceschet, and A. Peron. Representing and Reasoning about Temporal Granularities. *Journal of Logic and Computation*, 14(1):51–77, 2004.

[Chu36] A. Church. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics*, 58:354–363, 1936.

[Chu40] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[CL96] G. Chen and G. Longo. Subtyping Parametric and Dependent Types. In *Kamareddine, et al. (eds.), Type Theory and Term Rewriting*, 1996. invited lecture.

[CMP93] I. Cervesato, A. Montanari, and A. Provetti. On the Non-monotonic Behavior of Event Calculus for Deriving Maximal Time Intervals. *Interval Computations*, 3(2):83–119, 1993.

[Cod70] E.F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970.

[CR87] J. Clifford and A. Rao. A Simple General Structure for Temporal Domains. In *C. Rolland, and M. Leonard (eds.), Temporal Aspects of Information Systems*, pages 17–28. Elsevier Science Publishers, 1987.

[CTB03] S. Casteleyn, O.D: Troyer, and S. Brockmans. Design Time Support for Adaptive Behavior in Web Sites. In *Proceedings of the 2003 ACM Symposium on Applied Computing*, pages 1222–1228. ACM Press, 2003.

[CTZ00] S.Y. Chien, V. Tsotras, and C. Zaniolo. A Comparative Study of Version Management Schemes for XML Documents. Time Center, Technical Report, TR-51, 2000.

[CTZ01] S. Chien, V.J. Tsotras, and C. Zaniolo. Efficient Management of Multiversion Documents by Object Referencing. In *Proceedings of the* 27^{th} *International Conference on Very Large Data Bases*, pages 291–300. Morgan Kaufmann Publishers, 2001.

[CW85] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[Cyc97] Cycorp Inc. *Cycorp Ontology*, 1997. $http://www.cyc.com/products2.html$.

[DAR02] DARPA Agent Markup Language. *A DAML Ontology of Time*, 2002.

[dB80] N.G. de Bruijn. A Survey of the Project AUTOMATH. In *J.P. Seldin and J.R. Hindley (eds.), To H.B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalisms.* Academic Press, 1980.

[DBB$^+$88] U. Dayal, B.T. Blaustein, A.P. Buchmann, U.S. Chakravarthy, M. Hsu, R. Ledin, D.R. McCarthy, A. Rosenthal, S.K. Sarin, M.J. Carey, M. Livny, and R. Jauhari. The HIPAC Project: Combining Active Databases and Timing. *SIGMOD Record*, 17(1):51–70, 1988.

[DBCK96] F. Douglis, T. Ball, Y.-F. Chen, and E. Koutsofios. WebGUIDE: Querying and Navigating Changes in Web Repositories. *Computer Networks*, 28(7–11):1335–1344, 1996.

[DBCK98] F. Douglis, T. Ball, Y.-F. Chen, and E. Koutsofios. The AT&T Internet Difference Engine: Tracking and Viewing Changes on the Web. *World Wide Web*, 1(1):27–44, 1998.

[dBHW99] P. de Bra, G.J. Houben, and H. Wu. AHAM: A Dexter-based Reference Model for Adaptive Hypermedia. In *Proceedings of the* 10^{th} *ACM Conference on Hypertext and Hypermedia*, pages 147–156. ACM Press, 1999.

[DC91] W. Davis and J. Carnes. Clustering Temporal Intervals to Generate Reference Hierarchies. In *Proceedings of the* 2nd *International Conference on Principles of Knowledge Representation and Reasoning*, pages 111–117. Morgan Kaufman, 1991.

[Dem04] S. Demri. LTL over Integer Periodicity Constraints (Extended Abstract). In *Proceedings of 7th International Conference on Foundations of Software Science and Computation Structures*, LNCS 2987, pages 121–135. Springer-Verlag, 2004.

[DJ97] T. Drakengren and P. Jonsson. Eight Maximal Tractable Subclasses of Allen's Algebra with Metric Time. *Journal of Artificial Intelligence Research*, 7:25–45, 1997.

[DLW04] C.E. Dyreson, H. Lin, and Y. Wang. Managing Versions of Web Documents in a Transaction-time Web Server. In *Proceedings of the* 13th *International World Wide Web Conference*, pages 422–432. ACM Press, 2004.

[DMB92] M. Denecker, L. Missiaen, and M. Bruynooghe. Temporal Reasoning with Abductive Event Calculus. In *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 384–388. John Wiley and Sons, Chichester, 1992.

[DMP91] R. Dechter, I. Meiri, and J. Pearl. Temporal Constraint Networks. *Artificial Intelligence*, 49(1–3):61–95, 1991.

[Dow79] D. Dowty. *Word Meaning and Montague Grammar*. Kluwer Academic Publishers, 1979.

[DR01] N. Dershowitz and E.M. Reingold. *Calendrical Calculations: The Millennium Edition*. Cambridge University Press, 2001.

[DR04] N. Dershowitz and E.M. Reingold. Hebrew Dating. In *Proceedings of the* 24th *International Jewish Genealogy Conference* . Israel Genealogy Society, 2004.

[Dyr01] C. Dyreson. Towards a Temporal World Wide Web: A Transaction-Time Server. In *Proceedings of the 12th Australian Database Conference*, pages 169–175. ACM Press, 2001.

[Esh88] K. Eshghi. Abductive Planning with Event Calculus. In *Proceedings of the* 5th *International Conference on Logic Programming, MIT Press*, pages 562–579, 1988.

[Euz93] J. Euzenat. Représentation Granulaire du Temps. *Revue d'Intelligence Artificielle*, 7(3):329–361, 1993.

[Euz95] J. Euzenat. A Categorical Approach to Time Representation: First Studies on Qualitative Aspects. In *Proceedings of the IJCAI Workshop on Spatial and Temporal Reasoning*, pages 142–152, 1995.

[Euz01] J. Euzenat. Granularity in Relational Formalisms with Applications to Time and Space Representation. *Computational Intelligence*, 17(3):703–737, 2001.

[FA97] T. Frühwirth and S. Abdennadher. *Constraint-Programmierung.* Springer-Verlag, 1997.

[FGV05] M. Fisher, D. Gabbay, and L. Vila. *Handbook of Temporal Reasoning in Artificial Intelligence.* J. Hendler, H. Kitano, B. Nebel (eds.), Foundations of Artificial Intelligence: Volume I. Elsevier, 2005.

[FHN72] R. Fikes, P. Hart, and N.J. Nilsson. Learning and Executing Generalized Robot Plans. *Artificial Intelligence*, 3(4):251–288, 1972.

[FM94] J.L. Fiadeiro and T. Maibaum. Sometimes "Tomorrow"is "Sometime": Action Refinement in a Temporal Logic of Objects. In *Proceedings of the 1st International Conference on Temporal Logic*, LNCS 827, pages 48–66. Springer-Verlag, 1994.

[FM01] M. Franceschet and A. Montanari. A Combined Approach to Temporal Logics for Time Granularity. In *Workshop on Methods for Modalities*, 2001.

[FN71] R. Fikes and N. Nilsson. STRIPS: A new Approach to Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2(3-4):189–208, 1971.

[Fre92] C. Freska. Temporal Reasoning Based on Semi-Intervals. *Artificial Intelligence*, 54(1):199–227, 1992.

[FWP97] A. Fernandes, M. Williams, and N. Paton. A Logic-based Integration of Active and Deductive Databases. *New Generation Computing*, 15(2):205–244, 1997.

[Gal90] A. Galton. A Critical Examination of Allen's Theory of Action and Time. *Artificial Intelligence*, 42(2-3):159–188, 1990.

[Gan99] S. Gançarski. Database Versions to Represent Bitemporal Databases. In *Proceedings of the 10th Conference on Database and Expert Systems Applications*, LNCS 1677, pages 832–841. Springer-Verlag, 1999.

[GD92] S. Gatziu and K.R. Dittrich. SAMOS: An Active Object-Oriented Database System. *IEEE Data Engineering Bulletin*, 15(1-4):23–26, 1992.

[GF92] M. Genesereth and R. Fikes. *Knowledge Interchange Format, Version 3.0 Reference Manual.* KSL, Technical Report, KSL-92-86, 1992.

[GJ79] M. Garey and D. Johnson. *Computers and Intractability.* W. Freeman and Company, 1979.

[GJS93] N. Gehani, H.V. Jagadish, and Oded Shmueli. COMPOSE: A System for Composite Specification And Detection. In *Advanced Database Concepts*, LNCS 759, pages 3–15. Springer-Verlag, 1993.

[GLL97] G. De Giacomo, Y. Lespérance, and H.J. Lévesque. Reasoning about Concurrent Execution, Prioritized Interrupts, and Exogenous Actions in the Situation Calculus. In *Proceedings of the* 15th *International Joint Conference on Artificial Intelligence*, pages 1221–1226. Morgan Kaufmann Publishers, 1997.

[GLR91] M. Gelfond, V. Lifschitz, and A. Rabinov. What are the Limitations of the Situation Calculus? In *Automated Reasoning, Essays in Honor of Woody Bledsoe, (ed.)S. Boyer*, pages 167–181. Kluwer Academic Publishers, 1991.

[GLS96] L. Garrido-Luna and K.P. Sycar. Towards a Totally Distributed Meeting Scheduling System. In *Proceedings of the 20th Annual German Conference on Artificial Intelligence*, LNCS 1137, pages 85–97. Springer-Verlag, 1996.

[GM89] J. Greer and G. McCalla. A Computational Framework for Granularity and its Application to Educational Diagnosis. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pages 477–482. Morgan Kaufmann Publishers, 1989.

[GM94] C.A. Gunter and J.C. Mitchell. *Theoretical Aspects of Object-Oriented Programming.* MIT Press, 1994.

[GM00] F. Grandi and F. Mandreoli. The Valid Web: An XML/XSL Infrastructure for Temporal Management of Web Documents. In *Proceedings of the 1st International Conference on Advances in Information Systems*, LNCS 1909, pages 294–303. Springer-Verlag, 2000.

[GPP95] M. Grigni, D. Papadias, and C. Papadimitriou. Topological Inference. In *Proceedings of the* 14th *International Joint Conference on Artificial Intelligence*, pages 901–907. Morgan Kaufmann Publishers, 1995.

[Gru93] T.R. Gruber. Towards Principles for the Design of Ontologies Used for Knowledge Sharing. In *N. Guarino and R. Poli (eds.), Formal Ontology in Conceptual Analysis and Knowledge Representation.* Kluwer Academic Publishers, 1993.

[HA89] P.J. Hayes and J.F. Allen. Moments and Points in an Interval-based Temporal Logic. *Computational Intelligence*, 5(4):225–238, 1989.

[Ham72] C.L. Hamblin. Instants and Intervals. In *J.T. Fraser, F.C. Haber, and G.H. Müller (eds.), The Study of Time*, pages 324–328. Springer-Verlag, 1972.

[Har79] D. Harel. *First-order Dynamic Logic.* LNCS 68. Springer-Verlag, 1979.

[Har84] D. Harel. Dynamic Logic. In *D. Gabby et al. (eds.), Handbook of Philosophical Logic, vol. II, Extensions of Classical Logic, Publishing Company, Dordrecht (NL)*, 1984.

[Her94] D. Hernández. Qualitative Representation of Spatial Knowledge. LNAI 804. Springer-Verlag, 1994.

[Hin03] A. Hinze. Efficient Filtering of Composite Events. In *Proceedings of the* 20th *British National Database Conference*, LNCS 2712, pages 207–225. Springer-Verlag, 2003.

[Hir96] R. Hirsh. Relational Algebras of Intervals. *Artificial Intelligence*, 83(2):267–295, 1996.

[HM01] V. Haarslev and R. Möller. RACER System Description. In *Proceedings of the 1st International Joint Conference on Automated Reasoning*, LNCS 2083, pages 701–706. Springer-Verlag, 2001.

[Hob85] J. Hobbs. Granularity. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 432–435. Morgan Kaufmann Publishers, 1985.

[Hof97] M. Hofmann. Syntax and Semantics of Dependent Types. In *P. Dybjer and A. Pitts (eds.) Semantics and Logic of Computation.* Cambridge University Press, 1997.

[Hor98] I. Horrocks. Using an Expressive Description Logic: FaCT or Fiction? In *Proceedings of the 6th International Conference on Principles of Knowledge Representation and Reasoning*, pages 636–649. Morgan Kaufmann Publishers, 1998.

[HR04] M.A. Harris and E.M. Reingold. Line Drawing and Leap Years. *ACM Computing Surveys*, 36:60–80, 2004.

[HST99] I. Horrocks, U. Sattler, and S. Tobies. Practical Reasoning for Expressive Description Logics. In *Proceedings of the 6th International Conference on Logic Programming and Automated Reasoning*, LNCS 1705. Springer-Verlag, 1999.

[IJ04] R. Ingria and J.Pustejovsky. TimeML Specification 1.0. $http://time2002.org$, 2004.

[ISO00] *ISO-8601: Data Elements and Interchange Formats – Information Interchange – Representation of dates and times.* International Organization for Standardization, 2000.

[Je98] C. Jensen and C. Dyreson (eds.). *The consensus glossary of temporal database concepts - February 1998 version*, 1998.

[KG77] K. Kahn and G.A. Gorry. Mechanizing Temporal Knowledge. *Artificial Intelligence*, 9:87–108, 1977.

[KJJ03] A. Krokhin, P. Jeavons, and P. Jonsson. The Tractable Subalgebras of Allen's Interval Algebra. *Journal of the ACM*, 50(5):591–640, 2003.

[KL91] H.A. Kautz and P. Ladkin. Integrating Metric and Temporal Qualitative Constraints. In *Proceedings of the* 9th *National Conference on Artificial Intelligence*, pages 241–246. MIT Press, 1991.

[KM93] R. Kozierok and P. Maes. A Learning Interface Agent for Scheduling Meetings. In *Proceedings of the ACM-SIGCHI International Workshop on Intelligent User Interfaces*, pages 81–88. ACM Press, 1993.

[KM99] L. Khatib and R. Morris. Generating Scenarios for Periodic Events with Binary Constraints. In *Proceedings of the* 6th *International Workshop on Temporal Representation and Reasoning*, pages 67–72. IEEE Computer Society, 1999.

[Kno] Knowledge Systems Laboratories, Stanford. *Ontolingua Server.* $http : //www.ksl - svc.stanford.edu.$

[Kou92] M. Koubarakis. Dense Time and Temporal Constraints with $\neq$. In *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning*, pages 24–35. Morgan Kaufmann Publishers, 1992.

[Kow92] R. Kowalski. Database Updates in the Event Calculus. *Journal of Logic Programming*, 12(1–2):121–146, 1992.

[KS86] R. Kowalski and M. Sergot. A Logic-based Calculus of Events. *New Generation Computing*, 4(1):67–95, 1986.

[KS94] R. Kowalski and F. Sadri. The Situation Calculus and the Event Calculus Compared. In *Proceedings of the 1994 International Symposium on Logic Programming*, pages 539–553. MIT Press, 1994.

[KW02] N. Koch and M. Wirsing. The Munich Reference Model for Adaptive Hypermedia Applications. In *Proceedings of the* 2nd *International Conference on Adaptive Hypermedia and Adaptive Web-Based Systems*, LNCS 2347, pages 213–222. Springer-Verlag, 2002.

[L9́6] H. Lévesque. What is Planning in the Presence of Sensing? In *Proceedings of the* 13th *National Conference on Artificial Intelligence*, pages 1139–1146. AAAI Press, 1996.

[Lad87] P.B. Ladkin. *The logic of Time Representation.* PhD Thesis, University of California, 1987.

[Lan64] P.J. Landin. The Mechanical Evaluation of Expressions. *Computer Journal*, 6:308–320, 1964.

[Lan65] P.J. Landin. A Correspondence Between ALGOL 60 and Church's Lambda-Notation: Parts I and II. *Communications of the ACM*, 8(2,3):89–101,158–165, 1965.

[Lan66] P.J. Landin. The next 700 Programming Languages. *Communications of the ACM*, 9(3):157–166, 1966.

[Lif91] V. Lifschitz. Towards a Metatheory of Action. In *Proceedings of the 2nd International Conference on Principles Knowledge Representation and Reasoning*, pages 376–386. Morgan Kaufmann Publishers, 1991.

[Lig98] G. Ligozat. Generalized Intervals: A Guided Tour. In *Proceedings of Workshop on Spatial and Temporal Reasoning*, 1998.

[LLR99] H. Lévesque, Y. Lespérance, and R. Reiter. A Situation Calculus Approach to Modeling and Programming Agents. In *In A. Rao and M.Wooldridge (eds.), Foundations and Theories of Rational Agency*, pages 275–299. Kluwer Academic Publishers, 1999.

[LM01] U. Dal Lago and A. Montanari. Calendars, Time Granularities, and Automata. In *Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases*, LNCS 2121, pages 279–298. Springer-Verlag, 2001.

[LMF86] B. Leban, D. McDonald, and D. Foster. A Representation for Collections of Temporal Intervals. In *Proceedings of the 5th National Conference on Artificial Intelligence*, pages 367–371. Morgan Kaufmann Publishers, 1986.

[LP92] Z. Luo and R. Pollack. *The LEGO Proof Development System: A User's Manual*, 1992. Technical Report, University of Edinburgh.

[LPR98] H. Lévesque, F. Pirri, and R. Reiter. Foundations for the Situation Calculus. *Linköping Electronic Articles in Computer and Information Science*, 3(18):159–178, 1998.

[LS95] F. Lin and Y. Shoham. Provably Correct Theories of Actions. *Journal of the ACM*, 42(2):293–320, 1995.

[Luo94] Z. Luo. Computation and Reasoning: A Type Theory for Computer Science. In *Number 11 in International Series of Monographs on Computer Science.* Oxford University Press, 1994.

[MAB04] W. May, J.J. Alferes, and F. Bry. Towards Generic Query, Update, and Event Languages for the Semantic Web. In *Proceedings of Workshop on Principles and Practices in Semantic Web Reasoning*, LNCS 3208, pages 19–33. Springer-Verlag, 2004.

[McC62] J. McCarthy. *LISP 1.5 Programmer's Manual*. MIT Press, 1962.

[McC02] J. McCarthy. Actions and other Events in Situation Calculus. In *Proceedings of the 8th International Conference on Principles and Knowledge Representation and Reasoning*, pages 615–628. Morgan Kaufmann Publishers, 2002.

[McD82] D. V. McDermott. A Temporal Logic for Reasoning about Processes and Plans. *Cognitive Science*, 6:101–155, 1982.

[MCF$^+$94] T. Mitchell, R. Caruana, D. Freitag, J. McDermott, and D. Zabowski. Experience with a Learning Personal Assistant. *Communications of the ACM*, 37(7):81–94, 1994.

[Mei96] I. Meiri. Combining Qualitative and Quantitative Constraints in Temporal Reasoning. *Artificial Intelligence*, 87(1–2):343–385, 1996.

[Mey99] J. Meyer. Dynamic Logic Reasoning about Actions and Agents. In *Workshop on Logic-Based Artifical Intelligence*, 1999.

[MH87] J. McCarthy and P. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In *Readings in Non-monotonic Reasoning*, pages 26–45. Morgan Kaufmann Publishers, 1987.

[Mis91] L. Missiaen. *Localized Abductive Planning with the Event Calculus*. PhD Thesis, Department of Computer Science, K.U. Leuven, 1991.

[Mit96] J.C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[MMCR92] A. Montanari, E. Maim, E. Ciapessoni, and E. Ratto. Dealing with Time and Granularity in the Event Calculus. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 702–712. IOS Press, 1992.

[Mon74] U. Montanari. Networks of Constraints: Fundamental Properties and Applications to Picture Processing. *Information Sciences – Informatics and Computer Science: An International Journal*, 7(2):95–132, 1974.

[Mon96] A. Montanari. *Metric and Layered Temporal Logics for Time Granularity*. ILLC Dissertation Series 1996-02, University of Amsterdam, 1996.

[Moo85] R. Moore. A Logic of Knowledge and Action. In *Hobbs, J.R. and Moore, R.C. (eds.) Formal Theories of the Common-Sense World*, pages 319–358. Ablex, 1985.

[MP96] A. Montanari and A. Policriti. Decidability Results for Metric and Layered Temporal Logics. *Notre Dame Journal of Formal Logic*, 37:260–282, 1996.

[MPP99] A. Montanari, A. Peron, and A . Policriti. Theories of Omega-Layered Temporal Structures: Expressiveness and Decidability. *Logic Journal of the IGPL*, 7(1):79–102, 1999.

[MPP02] P. Mateus, A. Pacheco, and J. Pinto. Observations and the Probabilistic Situation Calculus. In *Proceedings 8th International Conference on Principles of Knowledge Representation and Reasoning*, pages 327–338. Morgan Kaufmann Publishers, 2002.

[MS98] K. Marriott and P.J. Stuckey. *Programming with Constraints. An Introduction.* MIT Press, 1998.

[MS99] R. Miller and M. Shanahan. The Event Calculus in Classical Logic — Alternative Axiomatizations. *Linköping Electronic Articles in Computer and Information Science*, 4(16), 1999.

[MSZ01] S. McIlraith, T. Son, and H. Zeng. Semantic Web Services. *IEEE Intelligent Systems*, 16(2):46–53, 2001.

[MTH90] R. Milner, M. Tofte, and R.W. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[New36] I. Newton. *The Principia: Mathematical Principles of Natural Philosophy.* University of California Press, 1936.

[NLM03] K. Nørvåg, M. Limstrand, and L. Myklebust. TeXOR: Temporal XML Database on an Object-Relational Database System. In *Proceedings of the 5th International Andrei Ershov Memorial Conference on Perspectives of Systems Informatics*, LNCS 2890, pages 520–530. Springer-Verlag, 2003.

[Nør02] K. Nørvåg. Temporal Query Operators in XML Databases. In *Proceedings of the 2002 ACM Symposium on Applied Computing*, pages 402–406. ACM Press, 2002.

[NS80] W.H. Newton-Smith. *The Structure of Time.* Routledge & Heagan Paul, 1980.

[NS92] M. Niezette and J.-M. Stevenne. An Efficient Symbolic Representation of Periodic Time. In *Proceedings of the 1st Conference on Information and Knowledge Management*, LNCS 752, pages 161–168. Springer-Verlag, 1992.

[NWJ02] P. Ning, S.X. Wang, and S. Jajodia. An Algebraic Representation of Calendars. In *the Annuals of Mathematics and Artificial Intelligence.* Kluwer Academic Publishers, 2002.

[OB88] A. Ohori and P. Buneman. Type Inference in a Database Programming Language. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, pages 174–183. ACM Press, 1988.

[Oez04] P. Oezden. An Ontology of Socio-cultural Time Expressions. Master Thesis, University of Munich, 2004.

[OG98] H.J. Ohlbach and D. Gabbay. Calendar Logic. *Journal of Applied Non-classical Logics*, 8(4):291–324, 1998.

[Ohl03] H.J. Ohlbach. WebCal: An Advanced Calendar Server. Technical Report, University of Munich, 2003.

[ÖS95] G. Özsoyoglu and R. Snodgrass. Temporal and Real-time Databases: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):513–532, 1995.

[Pfe94] F. Pfenning. Elf: A Meta-Language for Deductive Systems. In *Proceedings of the 12th International Conference on Automated Deduction*, LNAI 814, pages 811–815. Springer-Verlag, 1994.

[Pfe96] F. Pfenning. The Practice of Logical Frameworks. In *Proceedings of the Colloquium on Trees in Algebra and Programming*, LNCS 1059, pages 119–134. Springer-Verlag, 1996.

[PH04] F. Pan and J.R. Hobbs. Time in OWL-S. In *Proceedings of AAAI Spring Symposium on Semantic Web Services*, pages 29–36, 2004.

[Pie02] B.C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[PPW03] G. Papamarkos, A. Poulovassilis, and P.T. Wood. Event-Condition-Action Rule Languages for the Semantic Web. In *Proceedings of the 1st International Workshop on Semantic Web and Databases*, pages 309–327, 2003.

[PR93] J. Pinto and R. Reiter. Temporal Reasoning in Logic Programming: A Case for the Situation Calculus. In *Proceedings of the 10th International Conference on Logic Programming*, pages 203–221. MIT Press, 1993.

[Pra76] V. Pratt. Semantical Considerations on Floyd-Hoare Logic. In *Proceedings of the 17th IEEE Symposium on Foundations of Computer Science*, pages 109–121, 1976.

[PSS02] T.R. Payne, R. Singh, and K. Sycara. Calendar Agents on the Semantic Web. *IEEE Intelligent Systems*, pages 84–86, May/June 2002.

[Ram00] J. Ramos. *The Situation and State Calculus: Specification and Verification.* PhD Thesis, IST, Universidade Técnica de Lisboa, 2000.

[RCC92] D.A. Randell, Z. Cui, and A. Cohn. A Spatial Logic Based on Regions and Connection. In *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning*, pages 165–176. Morgan Kaufmann Publishers, 1992.

[Rei92] R. Reiter. On Formalizing Database Updates. In *Proceedings of the 3^{rd} International Conference on Extending Database Technology*, LNCS 580, pages 10–20. Springer-Verlag, 1992.

[Rei93] R. Reiter. Proving Properties of States in the Situation Calculus. *Artificial Intelligence*, 64(2):337–351, 1993.

[Rei95] R. Reiter. On Specifying Database Updates. *Journal of Logic Programming*, 25(1):53–91, 1995.

[Rei01] R. Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, 2001.

[RG68] N. Rescher and J. Garson. Topological Logic. *Journal of Symbolic Logic*, 33:537–548, 1968.

[ROS98] J. Rushby, S. Owre, and N. Shankar. Subtypes for Specifications: Predicate Subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, 1998.

[RS90] H. Reichgelt and N. Shadbolt. A Specification Tool for Planning Systems. In *Proceedings of the 9^{th} European Conference on Artificial Intelligence,*, pages 541–546, 1990.

[RU71] N. Rescher and A. Urquhart. *Temporal Logic*. Library of Exact Philosophy. Springer-Verlag, 1971.

[RW69] G. Robinson and L. Wos. Paramodulation and Theorem Proving in First Order Theories. *Machine Intelligence*, 4:135–150, 1969.

[SD98] S. Sen and E.H. Durfee. A Formal Study of Distributed Meeting Scheduling. *Group Decision and Negotiation*, 7:265–289, 1998.

[SG88] Y. Shoham and N. Goyal. Representing Time and Action in AI. Revised Version of: Problems in Formal Temporal Reasoning. *Artificial Intelligence*, 36(1):49–61, 1988.

[SGdMM96] C. Sierra, L. Godo, R. López de Màntaras, and M. Manzano. Descriptive Dynamic Logic and its Application to Reflective Architectures. *Future Generation Computer Systems*, 12(2–3):157–171, 1996.

[Sha89] M. Shanahan. Prediction is Deduction but Explanation is Abduction. In *Proceedings of the 11^{th}International Joint Conference on Artificial Intelligence*, pages 1055–1060. Morgan Kaufmann Publishers, 1989.

[Sha90] M. Shanahan. Representing Continuous Change in the Event Calculus. In *Proceedings of the 9^{th} European Conference on Artificial Intelligence*, pages 598–603, 1990.

[Sha95] M. Shanahan. A Circumscriptive Calculus of Events. *Artificial Intelligence*, 75(2):249–284, 1995.

[Sho87] Y. Shoham. Temporal Logics in AI: Semantical and Ontological Considerations. *Artificial Intelligence*, 33(1):89–104, 1987.

[Sin03] R. Singh. RCal: An Autonomous Agent for Intelligent Distributed Meeting Scheduling. Master Thesis, Carnegie Mellon University, Pittsburgh, PA, 2003.

[SK95] F. Sadri and R. Kowalski. Variants of the Event Calculus. In *Proceedings of the 12th International Conference on Logic Programming*, pages 67–81. MIT Press, 1995.

[Smi80] R.G. Smith. The Contract Net Protocol: High-Level Communications and Control in a Distributed Problem Solver. *IEEE Transactions on Computers*, C-29(12):1104–1113, 1980.

[Sno95] R. Snodgrass. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, 1995.

[SNP90] J. Smith, B. Nordtström, and K. Petersson. *Programming in Martin-Löf's Type Theory. An Introduction*. Oxford University Press, 1990.

[Spr02] S. Spranger. Representation of Temporal Knowledge for Web-based Applications. Diploma Thesis, University of Munich, 2002.

[SSP02] R. Singh, K. Sycara, and T.R. Payne. Distributed AI, Schedules, and the Semantic Web. *XML Journal*, pages 84–86, November 2002.

[Ste81] M. Stefik. Planning with Constraints (MOLGEN: Part 1). *Artificial Intelligence*, 16:111–139, 1981.

[Sti85] M.E. Stickel. Automated Deduction by Theory Resolution. *Journal of Automated Reasoning*, 1(4):333–355, 1985.

[SWM95] P. Spruit, R. Wieringa, and J. Meyer. Axiomatization, Declarative Semantics and Operational Semantics of Passive and Active Updates in Logic Databases. *Journal of Logic and Computation*, 5(1):27–70, 1995.

[TC95] A. Tuzhilin and J. Clifford. On Periodicity in Temporal Databases. *Information Systems*, 30(5):619–639, 1995.

[Ter00] P. Terenziani. Integrated Temporal Reasoning with Periodic Events. *Computational Intelligence*, 16(2):210–256, 2000.

[Tho91] S. Thompson. *Type Theory and Functional Programming*. Addison Wesley, 1991.

[Tho99] S. Thompson. *Haskell – The Craft of Functional Programming.* Addison Wesley, 1999.

[Tob01] S. Tobies. *Complexity Results and Practical Algorithms for Logics in Knowledge Representation.* PhD Thesis, RWTH Aachen, 2001.

[vB91] J. van Benthem. *The Logic of Time.* D. Reidel Publishing Company, 1983; revised and expanded edition, 1991.

[vBC90] P. van Beek and R. Cohen. Exact and Approximate Reasoning about Temporal Relations. *Computational Intelligence*, 6(3):132–144, 1990.

[vHSD92] P. van Hentenryck, V. Saraswat, and Y. Deville. Constraint Processing in cc(FD). Technical Report, unpublished Manuscript, 1992.

[Vil82] M.B. Vilain. A System for Reasoning about Time. In *Proceedings of the* 2^{nd} *National (US) Conference on Artificial Intelligence*, pages 197–201. AAAI Press, 1982.

[VKvB90] M.B. Vilain, H.A. Kautz, and P. van Beek. Constraint Propagation Algorithms for Temporal Reasoning: A Revised Report. In *Readings in Qualitative Reasoning about Physical Systems*, pages 373–381. Morgan Kaufmann Publishers, 1990.

[VS96] L. Vila and E. Schwalb. A Theory of Time and Temporal Incidence based on Instants and Periods. In *Proceedings of the* 3^{rd} *Workshop on Temporal Representation and Reasoning.* IEEE Computer Society, 1996.

[W3C01] W3C, World Wide Web Consortium. *XML Schema Parts 1 and 2*, 2001.

[W3C03] W3C, World Wide Web Consortium. *SOAP – Messaging Framework and Adjuncts*, 2003.

[W3C04a] W3C, World Wide Web Consortium. *Extensible Markup Language (XML) 1.0 (Third Edition)*, 2004.

[W3C04b] W3C, World Wide Web Consortium. *OWL: Web Ontology Language*, 2004.

[W3C04c] W3C, World Wide Web Consortium. *RDF: Resource Description Framework*, 2004.

[W3C04d] W3C, World Wide Web Consortium. *Requirements for the Internationalization of Web Services, Working Group Note*, 2004.

[W3C05] W3C, World Wide Web Consortium. *Web Services Description Language (WSDL)*, 2005.

[Wal75] D: Waltz. Generating Semantic Descriptions from Drawing of Scenes with Shadows. In *P.H. Winston (ed.), The Psychology of Computer Vision*. McGraw Hill, 1975.

[Wan95] S.X. Wang. Algebraic Query Languages on Temporal Databases with Multiple Time Granularities. In *Proceedings of the 4th International Conference on Information and Knowledge Management*, pages 304–311. ACM Press, 1995.

[Web90] J. Weber. On the Representation of Concurrent Actions in the Situation Calculus. In *Proceedings of the 8th Biennial Conference of the Canadian Society for Computational Studies of Intelligence*, pages 28–32. Morgan Kaufmann Publishers, 1990.

[Web01] WebDAV Corp. *WebDAV*, 2001. *http://www.webdav.org*.

[Wij00] J. Wijsen. A String-based Model for Infinite Granularities. In *Proceedings of the AAAI Workshop on Spatial and Temporal Granularities*, pages 9–16, 2000.

[WJL91] G. Wiederhold, S. Jajodia, and W. Litwin. Dealing with Granularity of Time in Temporal Databases. In *Proceedings of the 3rd International Conference on Advanced Information Systems Engineering*, LNCS 498, pages 124–140. Springer-Verlag, 1991.

[WJS95] S.X. Wang, S. Jajodia, and V. Subrahmanian. Temporal Modules: An Approach Toward Federated Temporal Databases. *Information Sciences – Informatics and Computer Science: An International Journal*, 82(1–2):103–128, 1995.

[YS02] P. Yolum and M. Singh. Flexible Protocol Specification and Execution: Applying Event Calculus Planning Using Commitments. In *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multi Agent Systems*, pages 527–534. ACM Press, 2002.

[Zwa99] J. Zwanenburg. Pure Type Systems with Subtyping. In *Proceedings of the 4th International Conference on Typed Lambda Calculus and Applications*, LNCS 1581, pages 381–396. Springer-Verlag, 1999.