A Type Language for Calendars

François Bry, Frank-André Rieß, and Stephanie Spranger

Institute for Informatics, Univ. Munich, Germany
`http://www.pms.ifi.lmu.de`
contact: `spranger@pms.ifi.lmu.de`

Abstract. Time and calendars play an important role in databases, on the Semantic Web, as well as in mobile computing. Temporal data and calendars require (specific) modeling and processing tools. CaTTS is a type language for calendar definitions using which one can model and process temporal and calendric data. CaTTS is based on a "theory reasoning" approach for efficiency reasons. This article addresses type checking temporal and calendric data and constraints. A thesis underlying CaTTS is that types and type checking are as useful and desirable with calendric data types as with other data types. Types enable (meaningful) annotation of data. Type checking enhances efficiency and consistency of programming and modeling languages like database and Web query languages.

1 Introduction

Time and calendars play an important role in databases, on the Semantic Web, as well as in mobile computing. A difficulty of temporal and calendric data is that they often depend on cultural, legal, professional, and/or locational contexts [1]. E.g. 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. Calendars are arbitrary human abstractions of the physical flow of time. They enable to measure 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.

To enable computers to meaningfully process temporal and calendric data and expressions, the data need to be given a well-defined meaning. To this aim, the type language CaTTS has been developed. CaTTS is presented in [2], an article that focuses CaTTS' data and constraint modeling aspects. CaTTS is a type language for calendar definitions following a programming language approach using which one can model and process temporal and calendric data. Using CaTTS, one can easily define arbitrary calendars including rather complicated ones. In particular, time and calendar notions like "day", "month", "teaching term", and "exam week" can be defined as types. Calendric types are declared by (user-defined) predicates that specify the elements belonging to a type. E.g.

a type “exam week” may be declared by a predicate that selects only those elements from type “week” that contain examination days. Such calendric types are partially ordered (by a subtyping relation) in a CaTTS calendar specification, itself a type. E.g. type “exam week” is a subtype of type “week”, following a conversion interpretation of subtyping. In addition to providing means for modeling calendric types and calendars, CaTTS is based on a “theory reasoning” approach using constraint solving techniques for efficient automated reasoning. CaTTS' reasoner is presented in [3], an article that focuses on constraint reasoning with domains over arbitrary calendric types defined in CaTTS.

This article addresses type checking calendric data and constraints typed after calendric types defined in CaTTS. The terminology used is widespread in the type checking community [4]: “type checking” denotes “*static* type checking”, i.e. at compile time, and “dynamic checking” denotes “*dynamic* type checking”, i.e. at evaluation time. CaTTS' type checker is intended to be, in principle, used to type check programs or specifications in *any* language (e.g. SQL, XQuery, Sparql, RDF, OWL), using temporal and calendric data enriched with type annotations after some calendar specified in CaTTS. In particular, it is used to type check calendric constraint programs in CaTTS-CL, the constraint language of CaTTS. The authors of the work reported about in this article claim 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. 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 enable (meaningful) annotation of data. 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, e.g. that some data refer to days. Type checking ensures certain semantics on the data when processing them, e.g. that a week can never be during a day.

2 CaTTS: Programming with Time and Calendars

The type language 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. In what follows, the principal features of CaTTS-DL, i.e. *predicate types* to declaratively define arbitrary temporal and calendric notions like “day” or “teaching term” as types and those of CaTTS-CL, i.e. *time and conversion constraints* to reason over calendric data typed after types defined in CaTTS-DL are presented.

2.1 Subtype Constructors for Calendric Types

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 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 $\{l : \mathbb{Z}^* \mid \mathit{length}(l) = n\}$).

In type theory, predicate sets are used to define dependent types [5] and to define types in specification languages of proof assistents and theorem provers [6]. CaTTS uses predicate sets in a different manner. First, CaTTS uses predicate sets as a means to define calendric types like “month”, “working day”, “teaching term”, or “exam week”. Second, CaTTS restricts the definition of predicate sets to *aggregations* and *inclusions* of time by providing type constructors which are limited to aggregations and inclusions always define subsets isomorphic to the integers.[1] Finally, CaTTS uses predicate sets as a means to define conversions between calendric types, and, hence, constraint solving on calendric data over arbitrary calendric types.

Aggregations. E.g. the infinite set of weeks can be specified by the subset of those intervals of days having a duration of seven days and beginning on Mondays. This predicate can be directly expressed in CaTTS-DL defining the type `week` as an *aggregation subtype* of the type `day` as follows:

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

The type `week` is an aggregation of days such that each week corresponds to an interval of 7 days and such that the week indexed by 1 starts with the day indexed by 1. I.e. the predicate `7 day @ day(1)` of type `week` (that follows the constructor `aggregate`) specifies weeks in terms of days. We say that the type `week` is an *aggregation subtype* of the type `day`, written `week` $\preceq$ `day`.

Definition 1. *Let σ and τ be calendric types defined in CaTTS-DL.*
σ is an **aggregation subtype of** *τ, denoted $\sigma \preceq \tau$, if every element of σ is an interval over τ and every element of τ is included in (exactly) one element of σ.*

Note that CaTTS-DL supports the definition of aggregation subtypes which are neither periodic nor total. In particular, the following irregular aggregations can be defined in CaTTS: (1) aggregations that include elements of different durations involving often complex conditions like Gregorian months:

```
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 == 0) -> 29 day
```

[1] Note that the elements of each calendric type defined in CaTTS can be conveniently represented by integer sets.

```
    | otherwise                              -> 28 day
   end named february , ... , 31 day named december @ day(1);
```

(2) aggregations whose elements have gaps in time like "working week":

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

and (3) aggregations which do not aggregate time completely like "weekend":

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

CaTTS' aggregation subtype constructors are given in Appendix A.

Inclusions. E.g. the infinite set of weekend days can be specified by the subset of those days which are either Saturdays or Sundays. This predicate can be directly expressed in CaTTS-DL defining the type `weekend_day` as an *inclusion subtype* of the type `day` as follows:

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

The type `weekend_day` is an inclusion of days such that only those days are selected which correspond to the last two in each week. I.e. the predicate `relative i in week >= 6 && relative i in week <= 7` of type `weekend_day` (that follows the constructor `select...where`) specifies weekend days in terms of days. We say that the type `weekend_day` is an *inclusion subtype* of the type `day`, written `weekend_day` $\subseteq$ `day`.

Definition 2. *Let σ and τ be calendric types defined in CaTTS-DL.*
σ is an **inclusion subtype of** *τ, denoted $\sigma \subseteq \tau$, if every element of σ is an element of τ.*

Note that CaTTS provides with an appropriate set of language constructs sufficient to define inclusion subtypes corresponding to arbitrary subsets like "exam week", "teaching lesson" or someone's personal vacations, particularly sufficient to declare calendric notions of professional calendars as types. CaTTS' inclusion subtype constructors are common set-theoretic operations as well as the previously used `select` construct that allows for defining predicate sets. The syntactic forms are given in Appendix A.

Calendars. In CaTTS-DL, every calendric type is defined in a CaTTS-DL calendar specification. Such calendars are finite, packaged collections of calendric types and calendar specifications which can be reused and parameterized, i.e. calendars are themselves types. Each CaTTS-DL calendar specification contains a user-defined or a pre-defined *reference type.* CaTTS' pre-defined reference type corresponds to Unix seconds (UTC seconds with midnight at the onset of Thursday, January 1 of year 1970 (Gregorian) as fixed point indexed by 1). Any user-defined reference type can be defined by a parameterless type constructor. Each further type definition in a CaTTS-DL calendar specification is then an (direct or indirect) aggregation or inclusion subtype of the reference type. Thus, $\leq := \preceq \cup \subseteq$ is an order relation on calendric types defined in a CaTTS-DL calendar specification. The following definition formalizes the notion of calendar, as used in CaTTS.

Definition 3. *A* **calendar** $C = \{\tau_1, \ldots \tau_n\}$ *is a finite set of calendric types such that there exists a* $\tau_i \in C$ *and for all* $\tau_j \in C$*,* $i, j \in \{1...n\}$ *and* $i \neq j$ τ_j *is* $\leq$*-comparable with* τ_i*.*

The subsequently illustrated set of calendric types defines a CaTTS-DL calendar; each of the calendric types is $\leq$-comparable with type `day`.

$$\begin{tikzcd}
\text{year} \arrow[rr, dash, "\leq"] & & \text{month} \arrow[dr, dash, "\leq"] & \\
\text{week} \arrow[rrr, dash, "\leq"] & & & \text{day} \\
& & \text{weekend\_day} \arrow[ur, dash, "\leq"'] &
\end{tikzcd}$$

CaTTS-DL calendar specifications of different cultural and professional calendars, in particular the Gregorian, Islamic and Hebrew calendars and variations of the Gregorian calendar like the Japanese calendar are given in [2, 1].

Note that since 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”, or “Tue Oct 5 16:39:36 CEST 2004”, CaTTS provides with the format definition language CaTTS-FDL a means to define date formats that represent values of calendar types defined in a CaTTS-DL calendar specification. CaTTS-FDL has been introduced in [2].

2.2 Domain, Time, and Conversion Constraints

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 different domains of calendric types defined in CaTTS-DL. Such problems are solved by CaTTS-CL’s constraint solver. This solver is presented in [3].

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”, one can specify a problem like planning a meeting of three consecutive working days after 22nd April 2005 that is finished before May 2005. Such a problem can be expressed in CaTTS-CL as follows:

```
Meeting is 3 working_day &&
Meeting after "22.04.2005" && Meeting before "05.2005"
```

`Meeting is 3 working_day` is a CaTTS-CL *domain constraint* that has the following meaning: the variable `Meeting` represents the domain of all 3 working day long intervals. This domain is related to `Meeting` by applying the constraint `is` to `Meeting` and `3 working_day`. Thus, the type of `Meeting` (and the domain represented by `Meeting`) is “interval of working days”, denoted `Meeting: working_day`*, and read as “`Meeting` is a subset of the set of intervals of working days”. Note that since constraint variables like `Meeting` represent domains (i.e. sets of possible values), the symbol “:” is read as “subset of” rather than “element of”. `Meeting after "22.04.2005"` and `Meeting before "05.2005"` are CaTTS-CL *time constraints* that have the following meaning: the constraint `after` is a subset of the Cartesian product of the domains of `Meeting` (of type `working_day`*) and `"22.04.2005"` (of type `day`). Similarly, `before` is a subset of the Cartesian product of the domains of `Meeting` (of type `working_day`*) and `"05.2005"` (of type `month`).[2] Note that since CaTTS-CL is a language with subtyping the type of `Meeting after "22.04.2005"` is `day`$^*\times$`day`*, written `Meeting after "22.04.2005": day`$^*\times$`day`* and read as “`Meeting after "22.04.2005"` is a subset of pairs of intervals over days”. The type of `Meeting before "05.2005"` is `day`$^*\times$`day`*, as well. The types of the variable `Meeting` and the constraints `Meeting is 3 working_day`, `Meeting after "22.04.2005"`, and `Meeting before "05.2005"` are infered by applying CaTTS’ subtyping and typing rules which are given in Appendix B. Under the assumption that the context $\Gamma = (M : w\_day^*)$, the type of the constraint `M before "05.2005"` is inferred as follows[3]:

$$\dfrac{\dfrac{\dfrac{M : w\_day* \in \Gamma \;(\text{T-Var})}{\Gamma \vdash M : w\_day^* \quad w\_day^* \subseteq day^*}\;(\text{IT-Sub})}{\Gamma \vdash M : day^*} \quad \dfrac{\dfrac{\dfrac{\Gamma \vdash \text{"05.2005"} : month \;(\text{T-Date})}{\Gamma \vdash \text{"05.2005"} : month \quad month \preceq month^*}\;(\text{AT-Sub})}{\Gamma \vdash \text{"05.2005"} : month^* \quad month^* \preceq day^*}\;(\text{AT-Sub})}{\Gamma \vdash \text{"05.2005"} : day^* \;(\text{T-Interval})}}{\Gamma \vdash M\ before\ \text{"05.2005"} : day^* \times day^*}$$

Similary, the subtyping judgements appearing in this derivation tree (i.e. $month \preceq month^*$, $month^* \preceq day^*$, and $w\_day^* \subseteq day^*$) are inferred by applying the subtyping rules given in Appendix B.1.

The above given CaTTS-CL program is evaluated as follows: (1) CaTTS-CL constraints are evaluated by type checking them using the typing and subtyping rules given in Appendix B. (2) The typing derivations that result from type checking CaTTS-CL programs are translated into (lower-level) constraints used by CaTTS’ constraint propagation algorithm given in [3]. I.e. typing derivations are used to *convert* CaTTS-CL constraints into lower-level constraints without subtyping. This conversion is achieved by generating *conversion constraints* from the typing derivations. (3) Finally, the evaluation rules of CaTTS’ constraint solver, i.e. the constraint propagation algorithm that is given in [3] are used to obtain the behavior of CaTTS-CL programs. Thus, no evaluation rules, i.e. no operational semantics is (directly) defined for CaTTS-CL. Rather a semantics for CaTTS-CL is given by converting CaTTS-CL into a constraint language without subtyping (the “target” calculus, denoted $CaTTS - CL^{-sub}$) whose semantics is already understood since the constraints of the $CaTTS - CL^{-sub}$ are nothing but finite domain constraints over integer sets.

[2] In Constraint Programming every constraint is a subset of the Cartesian product of the domains of its variables (and it is equal to this Cartesian product when solved).

[3] “M” denotes `Meeting` and “w_day” denotes `working_day`

3 A Type Checker for CaTTS

CaTTS' type checker is straightforwardly defined by subtyping and typing relations which are given in Appendix B. The difference of CaTTS compared to type systems with subtyping is twofold: (1) CaTTS has two different subtype relations (i.e. aggregation and inclusion). (2) Base types like "week" or "teaching term" may be defined by predicates.

3.1 Subtyping

CaTTS' *subtyping relation* is defined as a collection of inference rules deriving *subtyping judgements* of the form $\sigma \leq \tau$ (where $\leq := \subseteq \cup \preceq$), read as "$\sigma$ is a subtype of τ". The subtyping relation $\leq$ defines a pre-order on calendric types defined in a CaTTS-DL calendar specification. The following two rules state that $\leq$ is a pre-order:

$$\sigma \leq \sigma \quad (\text{S-R}{\small\text{EFL}}) \qquad\qquad \frac{\rho \leq \sigma \quad \sigma \leq \tau}{\rho \leq \tau} \; (\text{S-T}{\small\text{RANS}})$$

A complete list of the inference rules to derive subtyping judgements that define CaTTS' subtyping relation is given in Appendix B.1. Note that CaTTS' subtyping relation differs from other (existing) subtyping relations in two aspects: (1) CaTTS' subtyping relation is defined by the union of two different subtyping relations, *aggregation* (cf. Definition 1) and *inclusion* (cf. Definition 2), which apply to different type constructors. (2) CaTTS provides with language constructs to define base types like `day`, `week`, or `teaching_term` by *predicates* either as an inclusion subtype or as an aggregation subtype (of a base type already defined in a CaTTS-DL calendar specification).

The subtyping rules for inclusions and aggregations defined by predicates are characteristic for CaTTS. They are given in the following.

$$\frac{\tau_{type} \qquad p_a(x), x : \tau}{\{z : \tau \mid p_a(x)\}_{type} \preceq \tau} \; (\text{AS-A}{\small\text{GGR}}) \qquad \frac{\tau_{type} \qquad p_i(x), x : \tau}{\{z : \tau \mid p_i(x)\}_{type} \subseteq \tau} \; (\text{IS-S}{\small\text{EL}})$$

The inference rule (AS-AGGR) defines the following: for a (base) type τ defined in CaTTS-DL, and an aggregation predicate $p_a(x)$ (in CaTTS' syntax: `aggregate e {,e} @ e`) with x of type τ, the set of the elements $x \in \tau$ that satisfy the predicate $p_a(x)$ define an aggregation subtype of τ, denoted $\{z : \tau \mid p_a(x)\}_{type} \preceq \tau$. The inference rule (IS-SEL) is defined similarly; $p_i(x)$ denotes an inclusion predicate (in CaTTS' syntax: `select e where c`). I.e. these two inference rules introduce subtyping judgements between pairs of (basic) types defined by some (user-defined) aggregation (or inclusion) predicate.

3.2 Well-Typed CaTTS-CL Programs

CaTTS-CL expressions and constraints (cf. Appendix A for the syntactic froms) are type checked by deriving *typing judgements* of the form $\Gamma \vdash e : \tau$ (i.e.

expression e has type τ in the context Γ). The context Γ is defined recursively as follows: $\emptyset$ is a context; if Γ is a context which does not declare (the variable) X and τ is a calendric type defined in CaTTS-DL, then $\Gamma, X : \tau$ is a context. A complete list of the inference rules to derive typing judgements that define CaTTS' *typing relation* is given in Appendix B.2. Among these inference rules, the following two illustrate the effect inclusion and aggregation subtyping has on type checking:

$$\frac{\Gamma \vdash e : \sigma \quad \sigma \subseteq \tau}{\Gamma \vdash e : \tau} \text{ (IT-SUB)} \qquad \frac{\Gamma \vdash e : \sigma^* \quad \sigma^* \preceq \tau^*}{\Gamma \vdash e : \tau^*} \text{ (AT-SUB)}$$

Those two rules make use of CaTTS' two subtyping judgements, one for inclusions, of the form $\sigma \subseteq \tau$, and one for aggregations, of the form $\sigma \preceq \tau$. In particular, those two rules connect the inference rules for typing with those for subtyping. I.e. whenever the type checker applies one of these two rules in a typing derivation, the subtype checker is called to verify the subtyping judgement (i.e. $\sigma \subseteq \tau$ (resp. $sigma \preceq \tau$). The rule (IT-SUB) applies if σ is an inclusion subtype of τ. The rule has the following meaning: whenever the subtyping judgement $\sigma \subseteq \tau$ is provable for types σ and τ, then the (CaTTS-CL) expression e of type σ can be "considered as" an expression of type τ. E.g. if $e =$ *"18.04.2005"* is an expression of type `working_day`, then *"18.04.2005"* can be also considered as a day (if `working_day` $\subseteq$ `day` can be proved). The rule (AT-SUB) applies if σ^* (read as "interval of σ") is an aggregation subtype of τ^* (read as "interval of τ"). The rule has the following meaning: whenever the subtyping judgement $\sigma^* \preceq \tau^*$ is provable for types σ^* and τ^*, then the (CaTTS-CL) expression e of type σ^* can be "considered as" an expression of type τ^*. E.g. if $e =$ *"first week in April 2005"* is an expression of type `week`*, then *"first week in April 2005"* can also be considered as an interval of days, i.e. the day interval from *"04.04.2005"* to *"10.04.2005"* (if `week`$^* \preceq$ `day`* can be proved). (AT-SUB) is only applicable to interval types because with aggregation subtyping only intervals of elements of the supertype are aggregated to elements of the (new) subtype. Since each element of a calendric type defined in CaTTS-DL is considered as having a duration, (AT-SUB) is a natural way of obtaining subtyping among aggregations of time as they appear in calendars.

Note that some (semantic) aspects of calendric data and constraints can only by checked dynamically like to check for inequality of the elements of different aggregation subtypes (e.g. that a day can never be equal to a week) and to check out-off bound violations of a finite calendric type like "Heisei", the era of the current Japanese emperor. Dynamic checking in CaTTS is out of the scope of this article.

3.3 A Subtyping Semantics for CaTTS

Giving semantics to a language with subtyping has been thoroughly investgated in the literature. E.g. in [7], subtyping has been expressed similarly to polymorphism in ML-style languages, e.g. in [8], subtyping has been expressed through explicit mechanisms as parts of the type checking system, and in [9], subtyping has been expressed as implicit coercion. Good surveys on semantic models of subtyping can be found in [10, 11]. Common to all these approaches is that subtyping has been mainly investigated for lambda-calculi with structural types like records. However, in CaTTS subtyping relies on and refers to predicate sets; thus, calendric types are not *syntactically* defined by means of structure, but, instead, *semantically* by means of predicates. Moreover, calendric types are used to type check the constraint language CaTTS-CL rather than a functional or object-oriented language. CaTTS' approach to a semantic model of subtyping relies on the approach proposed in [9]. This form of subtyping consists of a *set* (of elements) for each type together with a *conversion* from σ to τ whenever $\sigma \leq \tau$ is provable. This form of subtyping is called *conversion interpretation.* Such a conversion interpretation can be defined by a syntactic translation that replaces the rules for subsumption (i.e. (IT-Sub) and (AT-Sub) in CaTTS, given above) by conversion. Obtaining *coherence* (i.e. the logical connection between the subtyping and typing relations and the conversion semantics such that the algorithm implementing the conversion semantics is proved to be sound and complete) is serious to any conversion interpretation.

In CaTTS, the subtyping relation between calendric types is interpreted as a *conversion.* I.e. whenever $\sigma \leq \tau$ is provable from CaTTS' subtyping judgements, a conversion from σ to τ is performed. This conversion remains *implicit* in CaTTS-CL expressions. Subtyping is used in type checking such expressions. I.e. CaTTS-CL programs are evaluated by type checking them using the "high-level" typing and subtyping rules given in Appendix B.The conversion becomes explicit by using a "lower-level" language *without* subtyping to evaluate CaTTS-CL programs; in fact, by using time and conversion constraints of CaTTS' constraint propagation algorithm which is given in [3]. 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 CaTTS-CL constraints without subtyping, i.e. $CaTTS - CL^{-sub}$, and then using the evaluation relation (i.e. the constraint propagation algorithm given in [3]) to obtain the operational behavior of CaTTS-CL programs. This translation interprets subtyping in CaTTS-CL as *conversion constraints* already definable in terms of $CaTTS - Cl^{-sub}$. In particular, the proof that σ is a subtype of τ (i.e. the derivation from applying the high-level subtyping and typing rules on CaTTS-CL expressions) generates a *conversion* c_σ^τ from σ to τ whenever $\sigma \leq \tau$ is provable. The above given subsumption rules (IT-Sub) and (AT-Sub) for inclusions and aggregations are interpreted by the application of c_σ^τ (i.e. the conversion form σ to τ) to the interpretation of the CaTTS-CL expression e as an element of σ.

Formally, CaTTS' conversion interpretation of subtyping consists of conversions for subtyping judgements and for typing judgements.

For subtyping judgments $\sigma \leq \tau$, the conversion is defined by generating conversion contraints from subtyping derivations. We present four of these rules here; the full list is given in Appendix C.

subtyping rule		**conversion constraint**
$\sigma \leq \sigma$	(S-Refl)	$c_{\sigma}^{\sigma} \overset{def}{=} x:\sigma, y:\sigma,\ \mathit{convert}(x,y)$
$\dfrac{\rho \leq \sigma \quad \sigma \leq \tau}{\rho \leq \tau}$	(S-Trans)	$c_{\rho}^{\tau} \overset{def}{=} x:\rho,\ c_{\sigma}^{\tau}(c_{\rho}^{\sigma}(x))$
$\dfrac{\tau_{type} \qquad p_a(x), x:\tau}{\{z:\tau \mid p_a(x)\}_{type} \preceq \tau}$	(AS-Aggr)	$c_{\{z:\tau \mid p_a(x)\}}^{\tau} \overset{def}{=} x:\{z:\tau \mid p_a(x)\}, y:\tau,$ $\mathit{convert}(x,y)$
$\dfrac{\tau_{type} \qquad p_i(x), x:\tau}{\{z:\tau \mid p_i(x)\}_{type} \subseteq \tau}$	(IS-Sel)	$c_{\{z:\tau \mid p_i(x)\}}^{\tau} \overset{def}{=} x:\{z:\tau \mid p_i(x)\}, y:\tau,$ $\mathit{convert}(x,y)$

For the reflexivity rule (S-Refl) the ($CaTTS - CL^{-sub}$) conversion constraint *convert* is nothing but an identity operation. In case of the transitivity rule (S-Trans), the conversion is defined by composition of the conversions generated from the rule's hypotheses on an element of the subtype ρ. For cases of the subtyping rules for aggregations and inclusions defined by predicate, the conversions are basic $CaTTS - CL^{-sub}$ conversion constraints generated from the types' predicates $p_a(x)$ (resp. $p_i(x)$). (The implemenation of $CaTTS - CL^{-sub}$ conversion constraints are presented in [3].)

Having defined this conversion for every provable subtyping judgement, a meaning must be given to typed CaTTS-CL expressions. This is done by induction on the typing derivation of each expression. For typing judgments $\Gamma \vdash e : \tau$, the conversion applies the typing rules as with type checking. The subsumption rules (IT-Sub) and (AT-Sub) for inclusions and aggregations are interpreted by the application of the conversion for subtyping judgements to the interpretation of the CaTTS-CL expression e as an element of σ (i.e. of the subtype):

typing rule		**translation**
$\dfrac{\Gamma \vdash e : \sigma^* \quad \sigma^* \preceq \tau^*}{\Gamma \vdash e : \tau^*}$	(AT-Sub)	if $\Gamma \vdash e : \tau^*$ is derived from $\Gamma \vdash e : \sigma^*$ using $\sigma^* \preceq \tau^*$, then $\mathit{trans}(\Gamma \vdash e : \tau^*) = c_{\sigma^*}^{\tau^*}(\Gamma \vdash e : \sigma^*)$
$\dfrac{\Gamma \vdash e : \sigma \quad \sigma \subseteq \tau}{\Gamma \vdash e : \tau}$	(IT-Sub)	if $\Gamma \vdash e : \tau$ is derived from $\Gamma \vdash e : \sigma$ using $\sigma \subseteq \tau$, then $\mathit{trans}(\Gamma \vdash e : \tau) = c_{\sigma}^{\tau}(\Gamma \vdash e : \sigma)$

The two subsumption rules (AT-Sub) and (IT-Sub) are translated by making the subtyping conversion "explicit". I.e. the conversion constraint c_{σ}^{τ} (resp. $c_{\sigma^*}^{\tau^*}$) is applied to the expression e of type σ (resp. σ^*) whenever the subsumption rule (IT-Sub) (resp. (AT-Sub)) is applied in a typing derivation.

The remaining rules of this translation are given in Appendix C. This *translation* of CaTTS-CL expressions is syntactically performed by translating CaTTS-CL expressions into expressions expressible in $CaTTS - CL^{-sub}$. The translation is syntactic because it uses basic $CaTTS - CL^{-sub}$ conversion constraints and compositions of such constraints.

3.4 Coherence of the Conversion

The proof that the conversion of CaTTS-CL subtyping semantics is coherent is twofold: first, we have two show that the conversion constraints are *unique*. Second, we have to show that the translation is *coherent*.

Uniqueness of Conversion Constraints. To show that conversion constraints are unique by a series of proof transformations that do not change the associated conversion constraints, we begin with the elimination of certain uses of (S-Refl) and (S-Trans), in particular any instance of (S-Refl) that is a hypothesis of (S-Trans) can be eliminated from the proof without changing the conversion constraint. This can be shown by case for (S-Refl) left (resp. right) hypothesis of (S-Trans).

The uniqueness of conversion constraints is formalized as a theory over $CaTTS-CL^{-sub}$ where ϵ denotes the set of all equations of compositions of pairwise disjunct basic conversion constraints. In particular, ϵ says that if there are two different compositions of basic conversion constraints between two types, the two compositions must give the same function. I.e. when drawing the subtyping assumpptions of a proof in a diagram, then ϵ are the equations stating that this diagram commutes.

Proposition 1. *Suppose $\sigma \leq \tau$ and c_σ^τ and $\overline{c}_\sigma^\tau$ are the conversion constraints from σ to τ given by any two proofs of $\sigma \leq \tau$. Then $\epsilon \vdash c_\sigma^\tau = \overline{c}_\sigma^\tau$ in $CaTTS-CL^{-sub}$.*

The proof goes by induction on the structure of the type σ such that for any proofs $\sigma \leq \tau$ without the use of the transitivity rule (S-Trans) (which can be shifted to the end of any proof for $\sigma \leq \tau$) nor the reflexivity rule (S-Refl). The rules (S-Trans) and (S-Refl) and are the only subtyping rules that can be applied independent of the structure of the types σ and τ, i.e. they can be applied in each step of a proof for $\sigma \leq \tau$.

Coherence of Translations. For coherence of the translation, the proof of the typing derivations must be such that the use of the subsumption rules (IT-Sub) and (At-Sub) are postponed as much to the end of the proof as possible. This is for the same reasons as for postponing the subtyping rules (S-Trans) and (S-Refl) which can be applied in each step of the proof as it is the case for the subsumption rules since they are not syntax driven. The result is a typing derivation in which we first derive the minimum type for an expression e, and then use the subsumption rules in the final step in deriving any desired type.

Proposition 2. *Let $\Gamma \vdash e : \sigma$ a CaTTS-CL expression. Suppose there are two typing derivations for $\Gamma \vdash e : \sigma$ and let $\mathring{e}, \widehat{e} = trans(\Gamma \vdash e : \sigma)$ be the translations of e taken according to the two typing derivations. Then $\epsilon \vdash \Gamma \vdash \mathring{e} = \widehat{e} : \sigma$ in $CaTTS-CL^{-sub}$.*

The proof goes by transforming the typing derivations such that the subsumption rules are only applied in the last step of the derivation, in particular, in CaTTS-CL constraints. Then the typing derivation is determined by the structure of the expression e, and, thus, the same for any derivation of e. The final conversion constraint (applied when using one of the subsumption rules) is determined by the type σ. And this conversion does not change the associated expression with explicit conversions.

4 Related Work

Theories like time and calendars can be integrated into a language in two different ways. (1) Using the (automated reasoning) approach of "axiomatic reasoning", the integrated theory is axiomatized in the (general purpose) reasoning language. (2) Using the (automated reasoning) approach of "theory reasoning" [12, 13], the integrated theory is supported by specialized inference rules. This approach is well-known through paramodulation [14]. "Theory reasoning" makes user friendly modeling and efficient processing of data possible.

CaTTS complements data type definition languages and data modeling and reasoning methods for the Semantic Web such as XML Schema [15], RDF [16], and OWL [17]: CaTTS considerably simplifies the modeling of specificities of calendars such as leap years, sun-based cycles like Gregorian years, or lunar-based cycles like Hebrew months, "gaps" in time (e.g. "working-day"), "gapped" data items (e.g. "working-week") using predicate type constructors. 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.

CaTTS complements data modeling and programming languages like database query languages and Web query languages with type checking approaches specific to time and calendars. The well-known advantages of typed languages such as error detection, language safety, efficiency, consistency, abstraction, documentation, and annotation whereas consistency and annotation obtain particular interest due to overloaded semantics of temporal and calendric data equally apply to CaTTS.

CaTTS inherently differs from both temporal database systems (http://www.scism.sbu.ac.uk/cios/paul/Research/tdb_links.html links to research on temporal database systems) and active database systems(http://www.ifi.unizh.ch/dbtg/Links/adbs_sites.html links to research on active database systems). Research on temporal database systems mainly focuses on developing temporal data models, efficient temporal access methods, temporal dependencies, temporal consistencies, and design of temporal query languages. Temporal database systems are developed to store and access previous states of stored objects (possibly according to different temporal dimensions like transaction time, valid time, and event time). Active database systems are developed to initiate processes automatically when some state reaches a certain pre-defined condition. However, CaTTS is designed as a type language with type checking and theory reasoning approaches specialized in not only time but also calendar modeling and reasoning. CaTTS can be, in principle, used to enrich Database systems and languages as well as Web languages with means to manipulate temporal and calendric data and constraints.

CaTTS departs from time ontologies such as the KIF time ontology [18], the DAML time ontology [19], and time in OWL-S [20] in many aspects. 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 [14]. 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 comes along with a constraint solver dedicated to calendar definitions in CaTTS-DL [3]; this dedication makes considerable search space restrictions, hence gains in efficiency, possible.

5 Conclusion

This article has introduced a programming language approach to modeling and reasoning with time and calendars. CaTTS is a type language for calendar definitions. CaTTS enables user-friendly modeling of calendric and temporal data and constraints. CaTTS is based on a "theory reasoning" approach using constraint solving techniques for efficient automated reasoning. An approach to type checking temporal and calendric data typed after calendric types defined in CaTTS has been presented. Calendric types are used to give semantics to temporal and calendric data. Relationships between calendric types are expressed in terms of subtyping. Subtyping can be exploited to convert between calendric types which is necessary for reasoning on calendric data.

CaTTS can be, in principle, used to enrich *any* modeling or programming language like Database and/or Web query languages with means to user-friendly modeling and efficiently processing temporal and calendric data.

Acknowledgment

This research has been funded in part by the PhD Program Logics in Computer Science (GKLI) (cf. http://www.mathematik.uni-muenchen.de/ gkli) and the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Program project REWERSE number 506779 (cf. http://rewerse.net).

References

1. Bry, F., Haußer, J., Rieß, F.A., Spranger, S.: Cultural Calendars for Programming and Querying. In: Proc. 1st Forum on the Promotion of European and Japanese Culture and Traditions in Cyber Society and Virtual Reality. (2005)

2. Bry, F., Rieß, F.A., Spranger, S.: CaTTS: Calendar Types and Constraints for Web Applications. In: Proc. 14th Int. World Wide Web Conference, Japan. (2005)
3. Bry, F., Rieß, F.A., Spranger, S.: A Reasoner for Calendric and Temporal Data. submitted to publication (2005)
4. Pierce, B.C.: Types and Programming Languages. MIT Press (2002)
5. Hofmann, M.: Syntax and Semantics of Dependent Types. In: Semantics and Logic of Computation, Cambridge University Press (1997)
6. Rushby, J., Owre, S., Shankar, N.: Subtypes for Specifications: Predicate Subtyping in PVS. IEEE Transactions on Software Engineering **24** (1998) 709–720
7. Ohori, A., Buneman, P.: Type Inference in a Database Programming Language. In: Proc. of Symp. on Lisp and Functional Programming, USA. (1988) 174–183
8. Cardelli, L., Wegner, P.: On understanding Types, Data Abstraction, and Polymorphism. ACM Computing Surveys **17** (1985) 471–522
9. Breazu-Tannen, V., Coquand, T., Gunter, C., Scedrov, A.: Inheritance as Implicit Coercion. Information and Computation **93** (1991) 172–221
10. Gunter, C.A., Mitchell, J.C.: Theoretical Aspects of Object-Oriented Programming. MIT Press (1994)
11. Mitchell, J.C.: Foundations for Programming Languages. MIT Press (1996)
12. Stickel, M.E.: Automated Deduction by Theory Resolution. Journal of Automated Reasoning **1** (1985) 333–355
13. Bry, F., Marchiori, M.: Ten Theses on Logic Languages for the Semantic Web. In: Proc. W3C Workshop on Rule Languages for Interoperability, USA. (2005)
14. Robinson, G., Wos, L.: Paramodulation and Theorem Proving in First Order Theories. Machine Intelligence **4** (1969) 135–150
15. W3C, World Wide Web Consortium: XML Schema Part 2: Datatypes. (2001)
16. W3C, World Wide Web Consortium: RDF Primer. (2004)
17. W3C, World Wide Web Consortium: OWL Web Ontology Language. (2004)
18. Knowledge Systems Laboratories, Stanford: Time Ontology in KIF. (1994)
19. DARPA Agent Markup Language: A DAML Ontology of Time. (2002)
20. Pan, F., Hobbs, J.R.: Time in OWL-S. In: Semantic Web Services, AAAI Spring Symposium Series. (2004)

A The Syntax of CaTTS

e ::=	*expressions:*	c ::=	*constraints:*
X	*variable*	true	
d	*CaTTS-FDL date*	false	
τ(i)	*part,* $i \in \mathbb{Z}$	X is 1 τ	*event*
n τ	*duration,* $n \in \mathbb{N}$	X is τ	*task*
[e..e]	*endpoint interval*	X is n τ	*task + duration* $n \in \mathbb{N}$
e upto e	*duration interval*	X intervalC Y	*interval constraint*
e downto e	*duration interval*	X metricC Y	*metric constraint*
binOp e e	*binary operation*	e intervalC Z	
unOp e	*unary operation*	e metricC Z	
		c && c	*conjunction*

binOp ::=	shift forward | shift backward | extend by | shorten by | relative to | relative in | $+$ | $-$ | $*$ | mod | div | min | max | avg
unOp ::=	duration | begin | end | index |
intervalC ::=	equals | before | after | starts | started_by | finishes | finished_by | during | contains | meets | met_by | overlaps | overlapped_by | within | on_or_before | on_or_after
metricC ::=	$==$ | $<=$ | $<$ | $>$ | $>=$ | $!=$

τ ::=	*type expressions:*
reference	*(user-defined or predefined) reference type*
refinement n @ e	*refinement,* $n \in \mathbb{N}$
aggregate e {,e} @ e	*(abs. anchored) aggregation*
aggregate e {,e} ˜@ z	*(rel. anchored) aggregation,* $z \in \mathbb{Z}$
select e where c	*selection*
τ^n	*duration*
τ^*	*time interval*
$\tau \& \tau$	*conjunction*
$\tau \mid \tau$	*disjunction*
$\tau \setminus \tau$	*exception*
$\tau \# < \tau$	*restriction*

B CaTTS' Typing and Subtyping Relations

B.1 Subtyping Relation

The subtyping relation $\leq$ (i.e. $\leq := \preceq \cup \subseteq$) defines a pre-order on CaTTS-DL types.

$$\sigma \leq \sigma \quad \text{(S-Refl)} \qquad\qquad \frac{\rho \leq \sigma \quad \sigma \leq \tau}{\rho \leq \tau} \text{ (S-Trans)}$$

Subtyping rule for interval types:

$$\sigma \leq \sigma^* \text{ (S-IntervalCoer)}$$

The subtyping rule for intervals and durations is covariant, as expected:

$$\frac{\sigma \leq \tau}{\sigma^* \leq \tau^*} \text{ (S-Interval)} \qquad\qquad \frac{\sigma \leq \tau}{\sigma^n \leq \tau^n} \text{ (S-Duration)}$$

Aggregation Subtype Rules. The aggregation subtype rules for aggregations and refinements:

$$\frac{\tau_{type} \qquad p_a(x), x : \tau}{\{z : \tau \mid p_a(x)\}_{type} \preceq \tau} \text{ (AS-Aggr)} \qquad \frac{reference_{type} \qquad p_r(x), x : reference}{reference \preceq \{z : reference \mid p_r(x)\}_{type}} \text{ (AS-Ref)}$$

The aggregation subtype rule for restrictions:

$$\sigma \# < \tau \preceq \tau \text{ (AS-Res)}$$

Inclusion Subtype Rules. The inclusion subtype rule for selections:

$$\frac{\tau_{type} \qquad p_i(x), x : \tau}{\{z : \tau \mid p_i(x)\}_{type} \subseteq \tau} \text{ (IS-SEL)}$$

The inclusion subtype rules conjunctions, disjunctions, and exceptions:

$$i \in \{1,2\},\ \tau_i \subseteq \tau_1 \mid \tau_2 \text{ (IS-DJ}_1\text{)} \qquad \frac{i \in \{1,2\},\ \sigma_i \subseteq \tau}{\sigma_1 \mid \sigma_2 \subseteq \tau} \text{ (IS-DJ}_2\text{)}$$

$$i \in \{1,2\},\ \tau_1 \& \tau_2 \subseteq \tau_i \text{ (IS-CJ}_1\text{)} \qquad \frac{i \in \{1,2\},\ \sigma \subseteq \tau_i}{\sigma \subseteq \tau_1 \& \tau_2} \text{ (IS-CJ}_2\text{)}$$

$$\tau \setminus \sigma \subseteq \tau \qquad \text{(IS-EX)}$$

B.2 Typing Relation

To connect subtyping and typing in CaTTS, the following two subsumption rules, the inclusion subtype subsumption rule and the aggregation subtype subsumption rule are given:

$$\frac{\Gamma \vdash e : \sigma \quad \sigma \subseteq \tau}{\Gamma \vdash e : \tau} \text{ (IT-SUB)} \qquad \frac{\Gamma \vdash e : \sigma^* \quad \sigma^* \preceq \tau^*}{\Gamma \vdash e : \tau^*} \text{ (AT-SUB)}$$

Typing Rules for Calendric Data

$$\frac{X : \tau \in \Gamma}{\Gamma \vdash X : \tau} \text{ (T-VAR)} \qquad \Gamma \vdash d : \tau \quad \text{(T-DATE)}$$

$$\frac{\Gamma \vdash i : \mathbb{Z}}{\Gamma \vdash \tau_1(i) : \tau_1} \text{ (T-PART)} \qquad \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash n\ \tau_1 : \tau_1^n} \text{ (T-DUR)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash [e_1..e_2] : \tau_1^*} \text{ (T-ENDPI)} \qquad \frac{\Gamma \vdash e_1 : \tau_1^n \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ to\ e_2 : \tau_1^*} \text{ (T-DURI)}$$

where $to \in \{upto, downto\}$.

Typing Rules for Calendric Operations

$$\frac{\Gamma \vdash e_1 : \tau_1}{\Gamma \vdash duration\ e_1 : \tau_1^n} \text{ (T-D)} \qquad \frac{\Gamma \vdash e_1 : \tau_1^*}{\Gamma \vdash b/e\ e_1 : \tau_1} \text{ (T-BE)} \qquad \frac{\Gamma \vdash e_1 : \tau_1}{\Gamma \vdash index\ e_1 : \mathbb{Z}} \text{ (T-I)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1^* \quad \Gamma \vdash e_2 : \tau_1^n}{\Gamma \vdash shift\ e_1\ f/b\ e_2 : \tau_1^*} \text{ (T-SHIFT)} \qquad \frac{\Gamma \vdash e_1 : \tau_1^* \quad \Gamma \vdash e_2 : \tau_1^n}{\Gamma \vdash e/s\ e_1\ by\ e_2 : \tau_1^*} \text{ (T-EXSH)}$$

$$\frac{\Gamma \vdash (index\ e_1 : \tau_1) : \mathbb{Z} \quad \tau_2 \quad \tau_2 \preceq \tau_1}{\Gamma \vdash relative\ (index\ e_1)\ in\ \tau_2 : \mathbb{Z}} \text{ (T-RELIN)} \qquad \frac{\Gamma \vdash (index\ e_1 : \tau_1) : \mathbb{Z} \quad \tau_2}{\Gamma \vdash relative\ (index\ e_1)\ to\ \tau_2 : \mathbb{Z}} \text{ (T-RELTO)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1^n \quad \Gamma \vdash e_2 : \tau_1^n}{\Gamma \vdash e_1\ ard\ e_2 : \tau_1^n} \text{ (T-ARD)} \qquad \frac{\Gamma \vdash e_1 : \mathbb{Z} \quad \Gamma \vdash e_2 : \mathbb{Z}}{\Gamma \vdash e_1\ arz\ e_2 : \mathbb{Z}} \text{ (T-ARZ)}$$

where where $b/e \in \{begin, end\}$, $f/b \in \{forward, backward\}$, $e/s \in \{extend, shorten\}$, $ard \in \{+, -, *, mod, div, min, max, avg\}$, and $arz \in \{+, -, *, mod, div, min, max, avg\}$.

Typing Rules for Calendric Constraints

$$true : \mathbb{B} \quad \text{(T-True)} \qquad false : \mathbb{B} \quad \text{(T-False)}$$

$$\frac{\Gamma, X : \tau \vdash 1\ \tau : \tau^n}{\Gamma \vdash X\ is\ 1\ \tau : \tau} \quad \text{(T-Event)} \qquad \frac{\Gamma, X : \tau^* \vdash < n >\ \tau : \tau^n}{\Gamma \vdash X\ is\ < n >\ \tau : \tau^*} \quad \text{(T-Task)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ intervalC\ e_2 : \tau_1 \times \tau_1} \quad \text{(T-Interval)} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ \&\&\ e_2 : \tau_1 \times \tau_2} \quad \text{(T-Conj)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1^n \quad \Gamma \vdash e_2 : \tau_1^n}{\Gamma \vdash e_1\ metricD\ e_2 : \tau_1^n \times \tau_1^n} \quad \text{(T-MetD)} \qquad \frac{\Gamma \vdash e_1 : \mathbb{Z} \quad \Gamma \vdash e_2 : \mathbb{Z}}{\Gamma \vdash e_1\ metricZ\ e_2 : \mathbb{Z} \times \mathbb{Z}} \quad \text{(T-MetZ)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ convert\ e_2 : \tau_1 \times \tau_2} \quad \text{(T-Convert)}$$

where $intervalC \in \{equals, before, after, starts, started\_by, finishes, finished\_by, during, contains, meets, met\_by, overlaps, overlapped\_by, within, on\_or\_before, on\_or\_after\}$, $metricD \in \{==, <=, <, >, >=, !=\}$, and $metricZ \in \{==, <=, <, >, >=, !=\}$. $< .. >$ denotes optionals.

C The Conversion

C.1 Conversion Constraints

For every subtyping judgement $\sigma \leq \tau$ (cf. Section B.1), the conversion constraint c_σ^τ is defined by induction on the proof of $\sigma \leq \tau$, i.e. by a tree of judgments in the constraint language without subtyping.

axiom/rule	**conversion constraint**
(S-Refl)	$c_\sigma^\sigma \overset{def}{=} x : \sigma, y : \sigma,\ convert(x, y)$
(S-Trans)	$c_\rho^\tau \overset{def}{=} x : \rho,\ c_\sigma^\tau(c_\rho^\sigma(x))$
(S-IntervalCoer)	$c_\sigma^{\sigma^*} \overset{def}{=} x : \sigma, y : \sigma^*,\ convert(x, y)$
(S-Interval)	$c_{\sigma^*}^{\tau^*} \overset{def}{=} x : \sigma^*,\ (c_\sigma^\tau)^*(x)$
(S-Duration)	$c_{\sigma^n}^{\tau^n} \overset{def}{=} x : \sigma^n,\ (c_\sigma^\tau)^n(x)$
(AS-Aggr)	$c^\tau_{\{z:\tau \mid p_a(x)\}} \overset{def}{=} x : \{z : \tau \mid p_a(x)\}, y : \tau,\ convert(x, y)$
(AS-Ref)	$c_{reference}^{\{z:reference \mid p_r(x)\}} \overset{def}{=} x : reference, y : \{z : reference \mid p_r(x)\},$ $convert(x, y)$
(AS-Res)	$c^\tau_{\sigma\# < \tau} \overset{def}{=} x : \sigma\# < \tau, y : \tau,\ convert(x, y)$
(IS-Sel)	$c^\tau_{\{z:\tau \mid p_i(x)\}} \overset{def}{=} x : \{z : \tau \mid p_i(x)\}, y : \tau,\ convert(x, y)$
(IS-Dj$_1$)	$c_{\tau_i}^{\tau_1 \mid \tau_2} \overset{def}{=} x : \tau_i, y : \tau_1 \mid \tau_2,\ convert(x, y)$
(IS-Dj$_2$)	$c^\tau_{\sigma_1 \mid \sigma_2} \overset{def}{=} x : \sigma_1 \mid \sigma_2,\ c^\tau_{\sigma_1}(x) \mid c^\tau_{\sigma_2}(x)$
(IS-Cj$_1$)	$c^{\tau_i}_{\tau_1 \& \tau_2} \overset{def}{=} x : \tau_1 \& \tau_2, y : \tau_i,\ convert(x, y)$
(IS-Cj$_2$)	$c_\sigma^{\tau_1 \& \tau_2} \overset{def}{=} x : \sigma,\ c_\sigma^{\tau_1}(x) \& c_\sigma^{\tau_2}(x)$
(IS-Ex)	$c^\tau_{\tau \setminus \sigma} \overset{def}{=} x : \tau \setminus \sigma, y : \tau,\ convert(x, y)$

C.2 Translation of CaTTS-CL Expressions

The translation *trans* of CaTTS-CL expressions into constraint expressions without subtyping is inductively defined on the typing judgements (cf. Section B.2).

axiom/rule	**translation**
(T-Var)	$trans(\Gamma \vdash X : \tau) = X : \tau \in \Gamma$
. . .	. . .
(AT-Sub)	if $\Gamma \vdash e : \tau^*$ is derived from $\Gamma \vdash e : \sigma^*$ using $\sigma^* \preceq \tau^*$, then $trans(\Gamma \vdash e : \tau^*) = c_{\sigma^*}^{\tau^*}(\Gamma \vdash e : \sigma^*)$
(IT-Sub)	if $\Gamma \vdash e : \tau$ is derived from $\Gamma \vdash e : \sigma$ using $\sigma \subseteq \tau$, then $trans(\Gamma \vdash e : \tau) = c_\sigma^\tau(\Gamma \vdash e : \sigma)$

Note that since the translation *trans* merely applies the typing rules as with type checking CaTTS-CL expressions except for the subsumption rules (IT-Sub) and (AT-Sub) which are interpreted by applying the conversion constraints for subtyping, only an example and the translation for the two subsumption rules are presented.