Logic Programming Representation of the Compound Term Composition Algebra

Anastasia Analyti[1], Ioannis Pachoulakis[2,3]

[1] *Institute of Computer Science, FORTH-ICS, Crete, Greece*
[2] *Dept. of Applied Informatics & Multimedia, TEI of Crete, Greece*
[3] *Centre for Technological Research of Crete (CTRC), Greece*
Email : analyti@ics.forth.gr, ip@epp.teicrete.gr

Abstract. The *Compound Term Composition Algebra* (CTCA) is an algebra with four algebraic operators, which can be used to generate the valid (meaningful) compound terms of a given faceted taxonomy, in an efficient and flexible manner. The positive operations allow the derivation of valid compound terms through the declaration of a small set of valid compound terms. The negative operations allow the derivation of valid compound terms through the declaration of a small set of invalid compound terms. In this paper, we show how CTCA can be represented in logic programming with negation-as-failure, according to both Clark's and well-founded semantics. Indeed, the SLDNF-resolution can be used for checking compound term validity and well-formedness of an algebraic expression in polynomial time w.r.t. the size of the expression and the number of terms in the taxonomy. This result makes our logic programming representation a competitive alternative to imperative algorithms. Embedding of our logic programming representation to the programming environment of a web portal for a computer sales company is demonstrated.

Keywords: Faceted Taxonomies, Logic Programming, Computational Complexity, Applications.

1. Introduction

A faceted taxonomy is a set of taxonomies, each describing a given domain from a different aspect, or facet [27]. The indexing of domain objects is done through conjunctive combinations of terms from the facets, called *compound terms*. For example, assume that the domain of interest is a set of Web pages for hotels in Greece, and suppose that we want to provide access to these pages according to the *Location* of the hotels and the *Sports* facilities they offer. Figure 1 shows these two facets. Each object is described using a compound term. For example, a hotel in Crete providing sea ski and wind-surfing facilities would be described by the compound term $\{Crete, SeaSki, Windsurfing\}$.

Faceted taxonomies carry a number of well-known advantages over single hierarchies in terms of building and maintaining them, as well as using them in multicriteria indexing. A drawback, however,

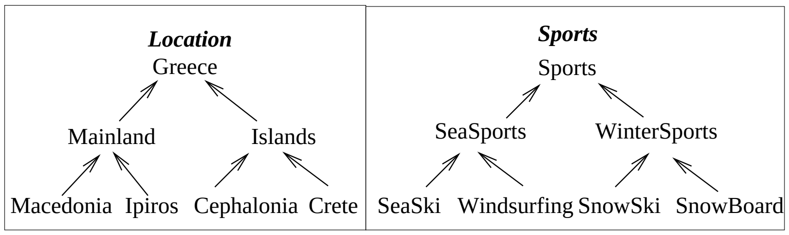

Figure 1. Two facets

is the cost of avoiding *invalid* combinations, i.e. compound terms that do not apply to any object in the domain. For example, the compound term $\{Crete, SnowBoard\}$ is an invalid compound term, as there are no hotels in Crete offering snow-board facilities.

In [31], the *compound term composition algebra* (CTCA) was proposed, whose operators (two positive and two negative) allow the efficient and flexible specification of valid compound terms, thus alleviating the main drawback of faceted taxonomies. Following this approach, given a faceted taxonomy, one can use an *algebraic expression* to define the desired set of compound terms. In each algebraic operation, the designer has to declare either a small set of valid compound terms from which other valid compound terms are inferred, or a small set of invalid compound terms from which other invalid compound terms are inferred. Then, a closed-world assumption is adopted for the remaining of the compound terms in the range of the operation. This is an important feature, as it minimizes the effort needed by the designer. Additionally, only the expression that defines the compound terminology has to be stored, as an inference mechanism (given in [31]) can check compound term validity in polynomial time. For well-formed expressions, the algebra is monotonic with respect to both valid and invalid compound terms, meaning that the valid and invalid compound terms of a subexpression are not invalidated by a larger expression. The semantics of CTCA are formally defined in [33], where it is also shown that CTCA cannot be represented in Description Logics [9] in a straightforward manner (due to the closed-world assumption inherent to the operations of CTCA).

An interesting representation formalism for CTCA is logic programming, where computation is treated as a deduction from a set of facts and rules. Logic programming provides a uniform way for representing data and computations, is declarative (compared to imperative languages), and has a solid semantic basis. In addition, complex data modelling is possible with the aid of logic programming. For example, we can represent concept hierarchies, semantic indexing information, and meta-level knowledge about database objects, allowing for knowledge-based query processing.

In this paper, we represent the algebra in logic programming. For each well-formed expression e, a logic program with negation LP_e and a set of first-order logic formulas are generated, which define compound term validity, equivalently to the algebraic definition. Specifically, we show that a compound term s is valid according to a well-formed expression e iff $valid(e^*, s)$ is derived from the corresponding logic program according to both Clark's semantics [8] and well-founded semantics [12] or, equivalently, $valid(e^*, s)$ is a logical consequence of the first-order logic representation (where e^* is the name of e). In fact, the first-order logic representation is just Clark's completion of LP_e. We also extend LP_e to a logic program $CTCA_e$ that defines well-formedness of an expression e, equivalently to the algebraic definition. We show that the SLDNF-resolution [8] can be used on $CTCA_e$ to check compound term validity and well-formedness of an expression e, in polynomial time with respect to the size of e and the number of terms of the faceted taxonomy. In fact, the SLDNF-resolution has the same time complexity as the corresponding imperative algorithms, making our logic programming representation a competitive alternative representation that can be embedded to any system that takes advantage of the powerful modelling and inferencing capabilities of logic programming and CTCA. Since Clark's semantics and the well-founded semantics of $CTCA_e$ coincide, the SLG-resolution [7, 6] (implementing the well-founded semantics) can also be used on $CTCA_e$ to check compound term validity and well-formedness of an expression e. In the case that some or all of the facets have a tree structure, an optimization is proposed and analyzed, along with the general case.

CTCA has several important applications, including object indexing and browsing, prevention of indexing errors, optimization of object retrieval, configuration management, and consistency control. An application to the implementation of the web-portal for a computer sales company is demonstrated. Rules promise to be widely useful in Internet electronic commerce as a programming mechanism [14, 13]. For example, rules can represent seller offerings of products and services, customer requirements, discount and advertisement policies. We show that our logic programming representation can be embedded to such a programming environment.

As several web applications can take advantage of CTCA, the representation of CTCA in proposed Semantic Web languages should be investigated. The idea of the Semantic Web is to describe the meaning of web data in a way suitable for automated reasoning. RDF(S) [18, 15] is a special predicate logical language that provides the basic constructs for defining web ontologies and is restricted to existentially quantified conjunctions of atomic formulas. OWL-DL [22] is an ontology representation language for the Semantic Web, that is a syntactic variant of the $\mathcal{SHOIN}$(**D**) Description Logic (DL) and a decidable fragment of first-order logic [16]. Rules constitute the next layer over the ontology languages of the Semantic Web. In contrast to DLs, rules allow the arbitrary interaction of variables in the body of the rules and non-monotonic features, such as negation-as-failure. The widely recognized need of having rules in the Semantic Web is also indicated by the Rule Markup Language (RuleML) initiative [29] and the inclusion in the Semantic Web Services Language (SWSL) proposal [5] of the *SWSL-Rules* Knowledge Representation, both supporting negation-as-failure. The proposed semantics for negation-as-failure of SWSL-Rules is the well-founded semantics.

Similarly to semantic web ontologies, encoding logic programs in a Semantic Web rule interchange format facilitates agent-based automated reasoning and sharing of computational logic among systems and applications. Certainly, expressing compound term validity in logic programming allows its direct encoding in a Semantic Web rule interchange format (as long as, it supports negation-as-failure) and thus, its easy embedding in such shared logic programming applications. The benefits of our logic programming representation are also strengthened by the fact that compound term validity according to an expression e cannot be efficiently represented in OWL-DL and RDF(S).

The remaining of this paper is organized as follows: Section 2 reviews the algebra and justifies the definition of a well-formed algebraic expression based on the monotonicity property. Section 3 defines compound term validity in terms of a logic program with negation LP_e. Section 4 defines well-formedness of an expression e in logic programming by extending LP_e. Section 5 presents a use case of our logic programming representation and discusses the representation of CTCA in OWL-DL and RDF(S). Finally, Section 6 concludes the paper. Proofs of all lemmas and propositions are given in Appendix A. The complete logic program, $CTCA_e$, for the expression e of our running example is given in Appendix B.

2. The Compound Term Composition Algebra

In this section, we present in brief the *compound term composition algebra*, defined in [31]. For more explanations, and examples the reader should refer to that article.

A *terminology* is a finite set of names, called *terms*. A *taxonomy* is a pair $(\mathcal{T}, \leq)$, where $\mathcal{T}$ is a *terminology* and $\leq$ is a reflexive, transitive, and antisymmetric relation over $\mathcal{T}$, called *subsumption*. A *compound term* over $\mathcal{T}$ is any subset of $\mathcal{T}$. For example, the following sets of terms are compound terms over the terminology $Sports$ of Figure 1: $\{SeaSki, Windsurfing\}$, $\{SeaSports\}$, and $\emptyset$. A *compound terminology* S over $\mathcal{T}$ is any set of compound terms that contains the compound term $\emptyset$.

The set of all compound terms over $\mathcal{T}$ can be ordered using the *compound ordering* over $\mathcal{T}$, defined as: $s \preceq s'$ iff $\forall t' \in s'$ $\exists t \in s$ such that $t \leq t'$. That is, $s \preceq s'$ iff s contains a narrower term for every term of s'. In addition, s may contain terms not present in s'. Roughly, $s \preceq s'$ means that s carries more specific indexing information than s'. Figure 2(a) shows the compound ordering of several compound terms over the terminology $Sports$ of Figure 1. Note that $s_1 \preceq s_3$, as s_1 contains $SeaSki$ which is a term narrower than the unique term $Sports$ of s_3. On the other hand, $s_1 \not\preceq s_2$, as s_1 does not contain a term narrower than $WinterSports$. Finally, $s_2 \preceq s_3$ and $s_3 \preceq \emptyset$. In fact, $s \preceq \emptyset$, for every compound term s. We say that two compound terms s, s' are *equivalent* iff $s \preceq s'$ and $s' \preceq s$. For example, $\{SeaSki, SeaSports\}$ and $\{SeaSki\}$ are equivalent. Intuitively, equivalent compound terms carry the same information.

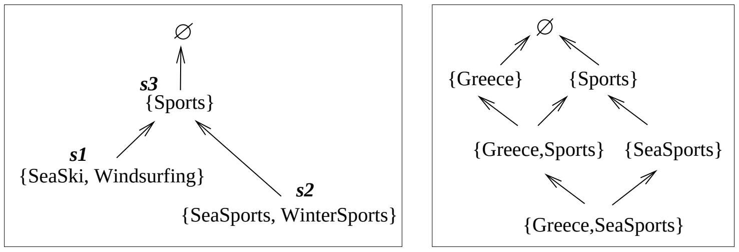

Figure 2. Two examples of compound taxonomies

A *compound taxonomy* over $\mathcal{T}$ is a pair $(S, \preceq)$, where S is a compound terminology over $\mathcal{T}$, and $\preceq$ is the compound ordering over $\mathcal{T}$ restricted to S. Let $P(\mathcal{T})$ be the set of all compound terms over $\mathcal{T}$ (i.e. the powerset of $\mathcal{T}$). Clearly, $(P(\mathcal{T}), \preceq)$ is a compound taxonomy over $\mathcal{T}$.

Let s be a compound term over $\mathcal{T}$. The broader and the narrower compound terms of s are defined as follows: $\text{Br}(s) = \{s' \in P(\mathcal{T}) \mid s \preceq s'\}$ and $\text{Nr}(s) = \{s' \in P(\mathcal{T}) \mid s' \preceq s\}$. Let S be a set of compound terms over $\mathcal{T}$. The broader and the narrower compound terms of S are defined as follows: $Br(S) = \cup\{\text{Br}(s) \mid s \in S\}$ and $Nr(S) = \cup\{\text{Nr}(s) \mid s \in S\}$.

We assume an underlying domain of discourse and a corresponding denumerable set of objects Obj, such that each object in Obj is indexed by a set of terms in $\mathcal{T}$. In particular, we consider a function ind: $Obj \rightarrow \mathcal{P}(\mathcal{T})$, called $index$, such that for all $o \in Obj$, it holds that: if $t \in ind(o)$ and $t \leq t'$ then $t' \in ind(o)$. That is, if an object o is indexed by a term t then it is also indexed by all terms that are broader than t. We say that a compound term s over $\mathcal{T}$ is *valid* (resp. *invalid*), if there is at least one (resp. no) object of the underlying domain indexed by all terms in s. We assume that every term of $\mathcal{T}$ is valid. However, a compound term over $\mathcal{T}$ may be invalid. Obviously, if s is a valid compound term, all compound terms in $\text{Br}(s)$ are valid. Additionally, if s is an invalid compound term, all compound terms in $\text{Nr}(s)$ are invalid. The formal definition of validity is given in [33].

Let $\{F_1, ..., F_k\}$ be a finite set of taxonomies, where $F_i = (\mathcal{T}_i, \leq_i)$, and assume that the terminologies $\mathcal{T}_1$, ... ,$\mathcal{T}_k$ are pairwise disjoint. Then the pair $\mathcal{F} = (\mathcal{T}, \leq)$, where $\mathcal{T} = \bigcup_{i=1}^{k} \mathcal{T}_i$ and $\leq\ = \bigcup_{i=1}^{k} \leq_i$, is a taxonomy which we shall call the *faceted taxonomy generated* by $\{F_1, ..., F_k\}$. We shall call the taxonomies $F_1, ..., F_k$ the *facets* of $\mathcal{F}$.

Clearly, all definitions introduced so far apply also to $(\mathcal{T}, \leq)$. For example, the set $S = \{\{Greece\}, \{Sports\}, \{SeaSports\}, \{Greece, Sports\}, \{Greece,\ SeaSports\}, \emptyset\}$ is a compound terminology over the terminology $\mathcal{T}$ of the faceted taxonomy shown in Figure 1. Additionally, the pair $(S, \preceq)$ is a compound taxonomy over $\mathcal{T}$ (see Figure 2(b)).

Let $\mathcal{F} = (\mathcal{T}, \leq)$ be the faceted taxonomy generated by a given set of facets $\{F_1, ..., F_k\}$. The problem is that $\mathcal{F}$ does not itself specify which compound terms, i.e. which elements of $P(\mathcal{T})$, are valid and which are not. To alleviate this problem, we introduce an algebra for defining a compound terminology over $\mathcal{T}$ (i.e. a subset of $P(\mathcal{T})$) which consists of the valid compound terms, only.

To begin with, we associate every facet $F_i = (\mathcal{T}_i, \leq_i)$ with a compound terminology T_i that we call the *basic compound terminology* of F_i. Specifically: $T_i = \cup\{\ \text{Br}(\{t\}) \mid t \in \mathcal{T}_i\}$.

We use the basic compound terminologies as the "building blocks" of the algebra. For defining the desired compound taxonomy the designer has to formulate an algebraic expression e, using the operations *plus-product*, *minus-product*, *plus-self-product*, *minus-self-product*, and initial operands the basic compound terminologies $\{T_1, ..., T_k\}$.

2.1. Algebraic operations

In this subsection, we describe each algebraic operation, in brief.

Let $\mathcal{S}$ be the set of compound terminologies over $\mathcal{T}$. First, we define the auxiliary n-ary operation $\oplus$ over $\mathcal{S}$, called *product*. This operation results in an "unqualified" compound terminology whose compound terms are all possible combinations of compound terms from its arguments. Specifically, let $S_1, ..., S_n$ be compound terminologies, we define: $S_1 \oplus ... \oplus S_n = \{s_1 \cup ... \cup s_n \mid s_i \in S_i\}$.

In general, not all compound terms in $S_1 \oplus ... \oplus S_n$ are valid. Therefore, we introduce two operations, namely *plus-product* and *minus-product*, which allow to specify the valid combinations of compound terms from the input compound terminologies $S_1, ..., S_n$, based on the parameters P and N, respectively. The set P is a set of compound terms that are certainly valid. On the other hand, the set N is a set of compound terms that are certainly invalid. These parameters are declared by domain experts that perform the indexing and allow to infer all compound terms in $S_1 \oplus ... \oplus S_n$ that are valid or invalid.

To proceed we need to distinguish what we shall call *genuine compound terms*. Intuitively, a genuine compound term combines non-empty compound terms from more than one compound terminology. The set of *genuine* compound terms over a set of compound terminologies $S_1, ..., S_n$, denoted by $G_{S_1,...,S_n}$, is defined as follows:

$$G_{S_1,...,S_n} = S_1 \oplus ... \oplus S_n - \bigcup_{i=1}^{n} S_i$$

For example if $S_1 = \{\{Greece\}, \{Islands\}, \emptyset\}$, $S_2 = \{\{Sports\}, \{WinterSports\}, \emptyset\}$, and $S_3 = \{\{Pensions\}, \{Hotels\}, \emptyset\}$ then $\{Greece, WinterSports, Hotels\} \in G_{S_1,S_2,S_3}$, $\{WinterSports, Hotels\} \in G_{S_1,S_2,S_3}$, but $\{Hotels\} \notin G_{S_1,S_2,S_3}$.

Assume that the compound terms of $S_1, ..., S_n$ are valid. We are interested in characterizing the validity of all combinations of compound terms of $S_1, ..., S_n$. As we already know the validity of the compound terms of $S_1, ..., S_n$, we are basically interested in characterizing the validity of the compound terms in $G_{S_1,...,S_n}$. This is done through the following operations, *plus-product* and *minus-product*.

We now define the *plus-product* operation, $\oplus_P$, an n-ary operation over $\mathcal{S}$ ($\oplus_P : \mathcal{S} \times ... \times \mathcal{S} \rightarrow \mathcal{S}$), where the parameter P is a set of valid compound terms from the product of the input compound terminologies. Specifically, the set P is a subset of $G_{S_1,...,S_n}$, as we assume that all compound terms in the input parameters are valid.

Definition 2.1. *Let $S_1, ..., S_n$ be compound terminologies and $P \subseteq G_{S_1,...,S_n}$. The* plus-product *of $S_1, ..., S_n$ with respect to P, denoted by $\oplus_P(S_1, ..., S_n)$, is defined as follows: $\oplus_P(S_1, ...S_n) = S_1 \cup ... \cup S_n \cup Br(P)$.*

This operation results in a compound terminology consisting of the compound terms of the initial compound terminologies, *plus* the compound terms which are broader than an element of P. This is because, if a compound term p is valid then all compound terms in Br(p) are also valid.

Now we define the *minus-product* operation, $\ominus_N$, an n-ary operation over $\mathcal{S}$ ($\ominus_N : \mathcal{S} \times ... \times \mathcal{S} \rightarrow \mathcal{S}$), where the parameter N is a set of invalid compound terms from the product of the input compound terminologies. Specifically, the set N is a subset of $G_{S_1,...,S_n}$, as we assume that all compound terms in the input operands are valid.

Definition 2.2. *Let $S_1, ..., S_n$ be compound taxonomies and $N \subseteq G_{S_1,...,S_n}$. The* minus-product *of $S_1, ..., S_n$ with respect to N, denoted by $\ominus_N(S_1, ..., S_n)$, is defined as follows: $\ominus_N(S_1, ...S_n) = S_1 \oplus ... \oplus S_n - Nr(N)$.*

This operation results in a compound terminology consisting of all compound terms in the product of the initial compound terminologies, *minus* all compound terms which are narrower than an element of N. This is because, if a compound term n is invalid then every compound term in Nr(n) is invalid.

For example, consider the compound terminologies S and S' shown in the left part of Figure 3[3], and suppose that we want to define a compound terminology that does not contain the compound terms $\{Islands, WinterSports\}$ and $\{Islands, SnowSki\}$, because they are invalid. For this purpose, we can use either a *plus-product* or a *minus-product* operation.

Specifically, we can use a plus-product operation, $\oplus_P(S, S')$, where $P = \{\{Islands, Seasports\}, \{Greece, SnowSki\}\}$. The compound taxonomy defined by this operation is shown in the right part of Figure 3. In this figure we enclose in squares the elements of P. Note that the compound terminology $\oplus_P(S, S')$ contains the compound term $s = \{Greece, Sports\}$, as $s \in Br(\{Islands, SeaSports\})$. However, it does not contain the compound terms $\{Islands, WinterSports\}$ and $\{Islands, SnowSki\}$, as they do not belong to $S \cup S' \cup Br(P)$.

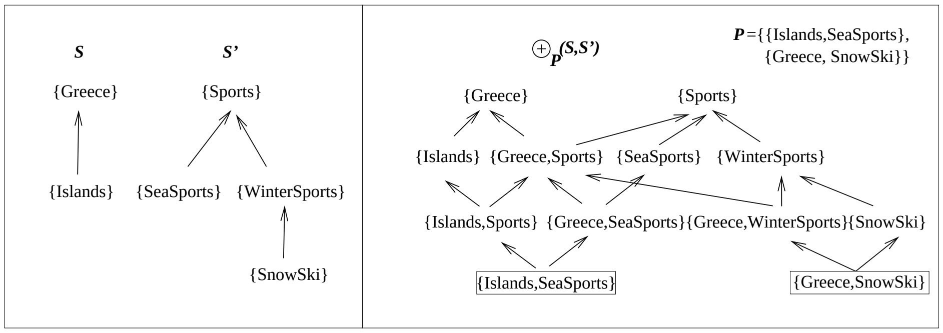

Figure 3. An example of a *plus-product*, $\oplus_P$, operation

Alternatively, we can obtain the compound taxonomy shown at the right part of Figure 3 by using a *minus-product* operation, i.e. $\ominus_N(S, S')$, with $N = \{\{Islands, WinterSports\}\}$. The result does not contain the compound terms $\{Islands, WinterSports\}$ and $\{Islands, SnowSki\}$, as they are elements of $Nr(N)$.

The operators introduced so far allow defining a compound terminology which consists of compound terms that contain at most one compound term from each basic compound terminology. However, a valid compound term may contain any set of terms of the same facet (multiple classification). To capture such cases, we define the *self-product*, $\overset{*}{\oplus}$, a unary operation which gives all possible compound terms of one facet. Subsequently, we shall modify this operation with the parameters P and N.

Let $\mathcal{BS}$ be the set of basic compound terminologies, that is $\mathcal{BS} = \{T_1, ..., T_k\}$. The *self-product* of T_i is defined as: $\overset{*}{\oplus}(T_i) = P(\mathcal{T}_i)$.

The notion of genuine compound terms is also necessary here. The set of *genuine* compound terms over T_i is defined as: $G_{T_i} = \overset{*}{\oplus}(T_i) - T_i$.

Now we define the *plus-self-product* operation, $\overset{*}{\oplus}_P$, a unary operation ($\overset{*}{\oplus}_P$: $\mathcal{BS} \rightarrow \mathcal{S}$) where the parameter P is a set of compound terms that are certainly valid. The set P is a subset of G_{T_i}.

[3]The compound term $\emptyset$ is omitted from the compound terminologies in the figure.

Definition 2.3. *Let T_i be a basic compound terminology and $P \subseteq G_{T_i}$. The* plus-self-product *of T_i with respect to P, denoted by $\overset{*}{\oplus}_P\ (T_i)$, is defined as follows: $\overset{*}{\oplus}_P\ (T_i) = T_i \cup Br(P)$.*

For example, the result of the operation $\overset{*}{\oplus}_P\ (Sports)$, where $P = \{\{SeaSki, Windsurfing\}, \{SnowSki, SnowBoard\}\}$ is shown in Figure 4.

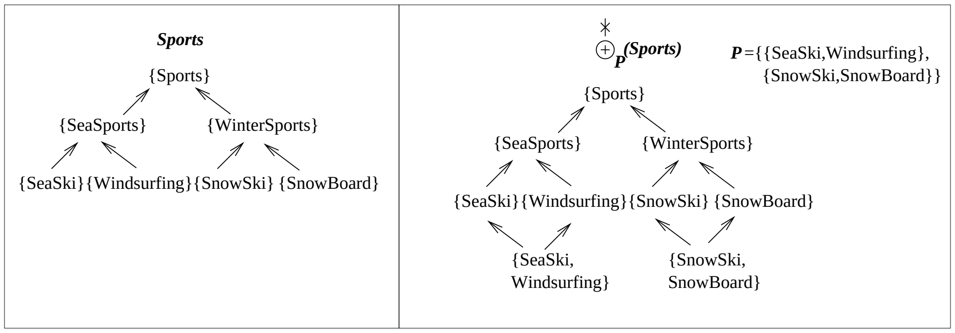
Sports
{Sports}
{SeaSports}
{WinterSports}
{SeaSki}{Windsurfing}{SnowSki} {SnowBoard}
⊕* P (Sports)
P={{SeaSki,Windsurfing}, {SnowSki,SnowBoard}}
{Sports}
{SeaSports}
{WinterSports}
{SeaSki}{Windsurfing}{SnowSki} {SnowBoard}
{SeaSki, Windsurfing}
{SnowSki, SnowBoard}

Figure 4. An example of a *plus-self-product*,$\overset{*}{\oplus}_P$, operation

The following definition introduces the *minus-self-product* operation, $\overset{*}{\ominus}_N$, a unary operation ($\overset{*}{\ominus}_N$: $\mathcal{BS} \rightarrow \mathcal{S}$) where the parameter N is a set of compound terms that are certainly invalid. The set N is a subset of G_{T_i}.

Definition 2.4. *Let T_i be a basic compound terminology and $N \subseteq G_{T_i}$. The* minus-self-product *of T_i with respect to N, denoted by $\overset{*}{\ominus}_N\ (T_i)$, is defined as follows: $\overset{*}{\ominus}_N\ (T_i) = \overset{*}{\oplus}\ (T_i) - Nr(N)$.*

For example, we can obtain the compound terminology of Figure 4 by the operation $\overset{*}{\ominus}_N\ (Sports)$, where $N = \{\{SeaSports, WinterSports\}\}$.

2.2. Algebraic Expressions

For defining the desired compound taxonomy, the designer has to formulate an expression e, where an expression is defined as follows:

Definition 2.5. *An expression over a set of facets $\{F_1, ..., F_k\}$ is defined according to the following grammar:*

$$e ::= \ \oplus_P(e, ..., e) \mid \ \ominus_N\ (e, ..., e) \mid \ \overset{*}{\oplus}_P\ T_i \mid \ \overset{*}{\ominus}_N\ T_i \mid T_i$$

The outcome of the evaluation of an expression e is denoted by S_e and is called the *compound terminology* of e. In addition, $(S_e, \preceq)$ is called the *compound taxonomy* of e.

Let $\mathcal{T}_e$ be the union of the terminologies of the facets appearing in an expression e. The expression e actually partitions the set $P(\mathcal{T}_e)$ into two sets:
(a) the set of valid compound terms, S_e, and
(b) the set of invalid compound terms $P(\mathcal{T}_e) - S_e$.

However, not all expressions are desirable. For example, consider the faceted taxonomy of Figure 1 and the expression $e = (Location \oplus_P Sports) \ominus_N Location$, where $P = \{\{Crete, SeaSki\}\}$ and $N = \{\{Islands, SeaSports\}\}$. Note that although the compound term $s = \{Islands, SeaSports\}$ is valid according to the expression $Location \oplus_P Sports$, s is invalid according the the larger expression e (in other words, there is a conflict). In general, we are interested only in *well-formed* expressions, defined as follows:

Definition 2.6. *An expression e is* well-formed *iff:*

(i) each basic compound terminology T_i appears at most once in e,

(ii) each parameter P that appears in e, is a subset of the associated set of genuine compound terms, e.g. if $e = \oplus_P(e_1, e_2)$ or $e = \overset{*}{\oplus}_P\ (T_i)$ then it should be $P \subseteq G_{S_{e_1}, S_{e_2}}$ or $P \subseteq G_{T_i}$, respectively, and

(iii) each parameter N that appears in e, is also a subset of the associated set of genuine compound terms, e.g. if $e = \ominus_N(e_1, e_2)$ or $e = \overset{*}{\ominus}_N\ (T_i)$ then it should be $N \subseteq G_{S_{e_1}, S_{e_2}}$ or $N \subseteq G_{T_i}$, respectively.

For example, the expression $(T_1 \oplus_P T_2) \ominus_N T_1$[4] is not well-formed, as T_1 appears twice in the expression.

As we have shown in [33], constraints (i), (ii), and (iii) ensure that the evaluation of an expression is monotonic, meaning that the valid and invalid compound terms of an expression e increase as the length of e increases (in other words, there are no conflicts). For example, if we omit constraint (i) then an invalid compound term according to an expression $T_1 \oplus_P T_2$ could be valid according to a larger expression $(T_1 \oplus_P T_2) \oplus_{P'} T_1$. If we omit constraint (ii) then an invalid compound term according to an expression $T_1 \oplus_{P_1} T_2$ could be valid according to a larger expression $(T_1 \oplus_{P_1} T_2) \oplus_{P_2} T_3$. Additionally, if we omit constraint (iii) then a valid compound term according to an expression $T_1 \oplus_P T_2$ could be invalid according to a larger expression $(T_1 \oplus_P T_2) \ominus_N T_3$.

This monotonic behavior in the evaluation of a well-formed expression results in a number of useful properties. Specifically, due to their monotonicity, well-formed expressions can be formulated in a systematic, gradual manner (intermediate results of subexpressions are not invalidated by larger expressions).

3. Representation of CTCA in Logic Programming

In this section, we show how compound term validity can be defined in logic programming. In particular, we show that for each well-formed expression e, we can generate a logic program with negation, LP_e, such that $s \in S_e$ iff there is a successful SLDNF-derivation of $LP_e \cup \{\leftarrow valid(e^*, s)\}$, where e^* is the name of e. We call this logic program, the *logic program* of e. As a by-product, we also define compound

[4]For binary plus-product and minus-product operations, we use also the infix notation.

term validity in first-order logic. We show that both Clark's semantics and the well-founded semantics of LP_e give the desired meaning to $valid(e^*, s)$. Thus, the SLG-resolution implementing the well-founded semantics can be used as an alternative to the SLDNF-resolution. In the case that all facets have a tree structure, an optimization is proposed and analyzed.

3.1. Logic Programming Background

Before we proceed, let us review a few concepts and results of logic programming [20, 4, 26] that will use in our representation.

Consider a first-order alphabet. An *atom* is a formula $p(t_1, ..., t_n)$, where p is an n-ary predicate symbol, and $t_1, ..., t_n$ are terms of the alphabet. A *literal* is an atom or the negation of an atom. A *ground atom* (resp. *ground literal*) is an atom (resp. literal) with no variables. A *definite program* is a set of rules of the form: $A \leftarrow A_1, ...A_n$, where A, A_i, $i = 1, ..., n$, are atoms. A *normal program* P is a set of rules of the form: $A \leftarrow L_1, ...L_n$, where A is an atom and L_i, $i = 1, ..., n$, are literals. A *definite goal* (resp. *normal goal*) is a special rule of the form $\leftarrow L_1, ..., L_n$, where L_i, $i = 1, ..., n$, are atoms (resp. literals). The *Herbrand Universe* of P is the set of all ground terms that can be formed from the constants and function symbols appearing in P. The *Herbrand Base*, denoted by $HB(P)$, is the set of all ground atoms that can be formed from the predicates in P and the terms in the Herbrand Universe of P.

The semantics of a definite program P are easily defined as the least Herbrand model of P. For normal programs matters are more complicated, as there may exist more than one minimal Herbrand model of P. Clark defined for each normal program P, the *completion* of P, $comp(P)$, which is a set of first-order logic closed formulas over the Herbrand Universe of P [8]. Specifically, if

$$p(t_1^1, ..., t_n^1) \leftarrow E_1 \quad ... \quad p(t_1^l, ..., t_n^l) \leftarrow E_l$$

is the set of rules in P that define a predicate p, then $comp(P)$ contains the following closed formula:

$$\begin{aligned}\forall x_1, ..., x_n \quad p(x_1, ..., x_n) \leftrightarrow \quad & \exists \bar{y}_1 (x_1 = t_1^1 \wedge ... \wedge x_n = t_n^1 \wedge E_1) \vee ... \vee \\ & \exists \bar{y}_l (x_1 = t_1^l \wedge ... \wedge x_n = t_n^l \wedge E_l)\end{aligned}$$

where $x_1, ..., x_n$ are variables not appearing in any of the rules that define p, and $\bar{y}_i$ are the variables of the i^{th} rule that defines p, for $i = 1, ..., l$.

If a predicate p appears in the body of a rule in P, but p does not appear in the head of any rule, then $comp(P)$ contains the closed formula $\forall x_1, ..., x_n \, \neg p(x_1, ..., x_n)$. In addition $comp(P)$ contains an equality theory, called *Clark's equality theory*, that is essentially a set of axioms which constraint the equality relation.

A literal L is considered true according to Clark's semantics of P iff $comp(P) \models L$.

Having defined validity model-theoretically, procedural methods for deriving validity are needed. A computation rule R is a function from the set of goals to the set of literals such that the value of the function for a goal is a literal of the goal, called the *selected literal*. For definite programs and definite goals, the *SLD-resolution* is a sound and complete resolution procedure. Specifically, let P be a definite program and let G_1 be a definite goal. Then, a *successful SLD-derivation* of $P \cup G_1$, via a computation rule R, is a sequence of goals $G_1, ..., G_n$ such that $G_n = \leftarrow$ and $\forall i < n, G_{i+1}$ is obtained from $G_i = \leftarrow B_1, ..., B_k$ as follows: (i) B_m is the atom in G_i, selected by the computation rule R, (ii) $C \leftarrow C_1, ..., C_l$ is a variant of a rule in P and θ_i is the *most general unifier* (*mgu*) of B_m and C, and (iii) $G_{i+1} = (B_1, ..., B_{m-1}, C_1, ..., C_l, B_{m+1}, ..., B_k)\theta_i$.

If a successful SLD-derivation exists for $P \cup \{\leftarrow A_1, ..., A_n\}$ then $(A_1, ..., A_n)\theta_1 ... \theta_n$ is true w.r.t the least Herbrand model of P. For normal goals, Clark introduced the non-monotonic inference rule, *negation-as-failure* [8]. The negation-as-failure rule is used to infer negative information, and intuitively states that if there is a goal $\leftarrow \neg A$, then try the goal $\leftarrow A$. If $\leftarrow A$ succeeds then $\leftarrow \neg A$ finitely fails. If $\leftarrow A$ finitely fails then $\leftarrow \neg A$ succeeds. Based on the negation-as-failure rule, Clark defined the *SLDNF-resolution* for normal programs. Below, we intuitively define a successful SLDNF-derivation and a finitely failed SLDNF-tree, two notions that are used throughout the rest of the paper. However, for the precise definitions, see [20, 8].

Let P be a normal program and let G_1 be a normal goal. Then, a *successful SLDNF-derivation* of $P \cup G_1$, via a computation rule R, is a sequence of goals $G_1, ..., G_n$ such that $G_n = \leftarrow$ and $\forall i < n, G_{i+1}$ is obtained from $G_i = \leftarrow L_1, ..., L_k$ as follows: Suppose that L_m is the literal in G_i, selected by the computation rule R. Then, either
(i) L_m is an atom, (ii) $C \leftarrow C_1, ..., C_l$ is a variant of a rule in P and θ_i is the *most general unifier* (*mgu*) of L_m and C, and (iii) $G_{i+1} = (L_1, ..., L_{m-1}, C_1, ..., C_l, L_{m+1}, ..., L_k)\theta_i$, or
(i) $L_m = \neg A_m$ is a ground negative literal, (ii) there is a finitely failed SLDNF-tree for $P \cup \{\leftarrow A_m\}$, and (iii) $G_{i+1} = L_1, ..., L_{m-1}, L_{m+1}, ..., L_k$.

Let P be a normal program and let G be a normal goal. Then, a *finitely failed SLDNF-tree* for $P \cup G$, via a computation rule R, is a tree satisfying the following:

1. The tree is finite and each node of the tree is a non-empty normal goal.

2. The root node is G.

3. Let $\leftarrow L_1, ..., L_n$ be a non-leaf node in the tree and suppose that L_m is selected. Then, either
   (i) L_m is an atom and for each variant $C \leftarrow C_1, ..., C_l$ of a rule in P, such that L_m and C are unifiable with mgu θ, the node has a child $\leftarrow (L_1, ..., L_{m-1}, C_1, ..., C_l, L_{m+1}, ..., L_k)\theta$, or
   (ii) $L_m = \neg A_m$ is a ground negative literal, there is a finitely failed SLDNF-tree for $P \cup \{\leftarrow A_m\}$, and the only child of the node is $\leftarrow L_1, ..., L_{m-1}, L_{m+1}, ..., L_k$.

4. Let $\leftarrow L_1, ..., L_n$ be a leaf node in the tree and suppose that L_m is selected. Then, either
   (i) L_m is an atom and there is no variant $C \leftarrow C_1, ..., C_l$ of a rule in P, such that L_m and C are unifiable, or
   (ii) $L_m = \neg A_m$ is a ground negative literal and there is a successful SLDNF-derivation for $P \cup \{\leftarrow A_m\}$.

Regarding SLDNF-resolution, Clark proved the following soundness results[5] [8], which we call *Clark's results*:

(i) Let P be a normal program. If L is a ground literal and $P \cup \{\leftarrow L\}$ has a successful SLDNF-derivation then $comp(P) \models L$.

(ii) Let P be a normal program. If L is a ground literal and $P \cup \{\leftarrow L\}$ has a finitely failed SLDNF-tree then $comp(P) \models \neg L$.

The following definition of a hierarchical logic program and the above results will be used to show that compound term validity and invalidity can be defined through Clark's semantics of our logic programming representation.

Definition 3.1. *A normal program P is $hierarchical$ if there is a mapping from its set of predicate symbols to the non-negative integers such that: for every rule r in P, the level of the predicate symbol of every literal in the body of r is less than the level of the predicate symbol in the head of r.*

Due to several drawbacks of Clark's semantics on normal programs (see Section 2 of [26]), the well-founded semantics (WFS) [12] is proposed as the appropriate semantics for normal programs. Moreover, in [5, 3], the WFS is defended as the appropriate semantics for the Semantic Web rule engines processing normal programs. As the SLDNF-resolution is used for implementing Clark's semantics, the SLG-resolution [7, 6] is used for implementing the WFS. We will show that there is a class of *stratified logic programs* (to which our logic programming representation belongs), for which Clark's semantics and the WFS coincide. A stratified logic program is defined as follows:

Definition 3.2. *A normal program P is $stratified$ if there is a mapping from its set of predicate symbols to the non-negative integers such that for every rule r in P, (i) the level of the predicate symbol of every positive literal in the body of r is less than or equal to the level of the predicate symbol p in the head of r, and (ii) the level of the predicate symbol of every negative literal in the body of r is less than the level of the predicate symbol p.*

Let us denote by $CLARK(P)$ and $WFS(P)$ the set of ground literals A and $\neg A$ ($A \in HB(P)$), which are true according to Clark's semantics and the WFS of P, respectively.

Lemma 3.1. *Let P be a stratified logic program such that for all $A \in HB(P)$, it holds that $comp(P) \models A$ or $comp(P) \models \neg A$. Then, $CLARK(P) = WFS(P)$.*

3.2. Logic Programming Representation

Finishing background information, we will now define compound term validity in logic programming. In the sequel, we denote the set of parameters of an algebraic expression e by $Params(e)$, the set of facets appearing in e by $Facets(e)$, and the list of facet names appearing in e by FL_e. Additionally, we

[5]Here, we present simplified versions of the corresponding theorems.

denote the name of a parameter X by X^*, the name of an expression e by e^*, and the name of a facet F by F^*. In our logic programming representation, sets are represented by lists.

Let $\mathcal{F}= (\mathcal{T}, \leq)$ be a faceted taxonomy generated by the taxonomies $\{F_1, ..., F_k\}$, and let e be an algebraic expression over $\{F_1, ..., F_k\}$. The alphabet of our logic programming representation consists of:

- a set of constants that contains the name of each subexpression of e, the names of all facets in $\{F_1, ..., F_k\}$, all terms in $\mathcal{T}$, and the symbol "[]" (for the empty list),
- the variable symbols $t, t', t_{mid}, s, s', n, p, s_1, ..., s_k, L, L_1, L_2, FL$,
- the function symbol •, allowing to define lists. Specifically, the list $[t_1, ..., t_n]$ corresponds to the compound term $\bullet(t_1, \bullet(..., \bullet(t_n, [])...))$,
- the base predicates $belongsFacet(t, facet)$, $subsumed^r(t, t')$, and $belongsParam(s, param)$, and
- the derived predicates $belongs(t, s)$, $belongsFL(t, facetList)$, $subsetFL(s, facetList)$, $intersectsFL(s, facetList, s')$, $invalid_0(expr, s)$, and $valid(expr, s)$.

The base predicates have the following meaning:

- The predicate $belongsFacet(t, facet)$ means that the term t belongs to the facet with name $facet$.
- The predicate $subsumed^r(t, t')$ means that the pair (t, t') belongs to the transitive reduction $\leq^r$ of $\leq$ on $\mathcal{T}$. The *transitive reduction* of a binary relation R on a set X, is defined as [11] $R^r = R_1 \backslash R_1^2$, where $R_1 = R \setminus \{(a, a) \mid a \in X\}$ and $R_1^2 = R_1 \circ R_1$. In practice, R^r is R without the reflexive and transitive relationships, and its graphical rendering is generally known as the *Hasse diagram* of R.
- The predicate $belongsParam(s, param)$ means that the compound term s belongs to the expression parameter with name $param$.

The base predicates are defined through the following ground facts, denoted by LP_e^{fact}:

(a) $belongsFacet(t, F_i^*) \leftarrow$, $\quad \forall t \in\mathcal{T}, F_i \in Facets(e): \quad t \in\mathcal{T}_i$,
(b) $subsumed^r(t, t') \leftarrow$, $\quad \forall t, t' \in \mathcal{T}: \quad t \leq^r t'$,
(c) $belongsParam(s, X^*) \leftarrow$, $\quad \forall s \in P(\mathcal{T}), X \in Params(e): \quad s \in X$.

Our derived predicates have the following meaning:

- The predicate $belongs(t, s)$ means that element t belongs to set s.
- The predicate $belongsFL(t, facetList)$ means that the term t belongs to the union of the terminologies of the facets in $facetList$.

- The predicate $subsetFL(s, facetList)$ means that the compound term s is subset of the union of the terminologies of the facets in $facetList$.
- The predicate $intersectsFL(s, facetList, s')$ means that the set s' is the intersection of the set s and the union of the terminologies of the facets in $facetList$.
- The predicate $subsumed(t, t')$ means that the term t is subsumed by the term t', that is $t \leq t'$. In fact, $subsumed(t, t')$ is derived by taking the reflexive and transitive closure of the base predicate $subsumed^r(t, t')$.
- The predicate $narrower(s, s')$ means that the compound term s is narrower than the compound term s' (i.e., $s \preceq s'$).
- The predicate $invalid_0(expr, s)$ means that the compound term s is narrower than the parameter N of the (negative) expression with name $expr$.
- The predicate $valid(expr, s)$ means that the compound term s is valid according to the expression with name $expr$.

We are now ready to define the logic program of e. The definition will be done again in two stages. First, we will present the algorithm $GenLP'(e)$ which takes as arguments an algebraic expression e and returns a logic program LP'_e that defines the predicates $valid(expr, s)$ and $invalid_0(expr, s)$. Then, we will expand LP'_e with the definitions of the rest of the derived predicates and the ground facts in LP_e^{fact}.

In Algorithm 3.1, the rules of LP'_e are generated recursively by following the parse tree of e. Note that the parameters e^*, P^*, N^*, FL_e, FL_{e_1}, ...,FL_{e_l}, F_i^* are replaced by constants, as the expression e is parsed.

Algorithm 3.1. $GenLP'(e)$
Input: *A well-formed expression* e
Output: *The logic program* LP'_e

```
( 1) case(e)  {
( 2) $\oplus_P(e_1, ..., e_l)$: $LP = \{valid(e^*, []) \leftarrow$
( 3)                 $valid(e^*, s) \leftarrow belongsParam(p, P^*), narrower(p, s)$
( 4)                 $valid(e^*, s) \leftarrow subsetFL(s, FL_{e_1}), valid(e_1^*, s)$
( 5)                 ...
( 6)                 $valid(e^*, s) \leftarrow subsetFL(s, FL_{e_l}), valid(e_l^*, s)\}$
( 7)        For $i = 1, ..., l$
( 8)                 $LP = LP \cup GenLP'(e_i)$
( 9)        return(LP)
(10) $\ominus_N(e_1, ..., e_l)$: $LP = \{invalid_0(e^*, s) \leftarrow belongsParam(n, N^*), narrower(s, n)$
(11)                 $valid(e^*, []) \leftarrow$
(12)                 $valid(e^*, s) \leftarrow subsetFL(s, FL_e), \neg invalid_0(e^*, s),$
(13)                           $intersectsFL(s, FL_{e_1}, s_1), valid(e_1^*, s_1), ...,$
(14)                           $intersectsFL(s, FL_{e_l}, s_l), valid(e_l^*, s_l)\}$
(15)        For $i = 1, ..., l$
(16)                        $LP = LP \cup GenLP'(e_i)$
(17)                return(LP)
(18) $\overset{*}{\oplus}_P\ (T_i)$:   $LP = \{valid(e^*, []) \leftarrow$
(19)                        $valid(e^*, s) \leftarrow belongsParam(p, P^*), narrower(p, s)$
(20)                        $valid(e^*, s) \leftarrow belongsFacet(t, F_i^*), narrower([t], s)\}$
(21)                return(LP)
(22) $\overset{*}{\ominus}_N\ (T_i)$:   $LP = \{invalid_0(e^*, s) \leftarrow belongsParam(n, N^*), narrower(s, n)$
(23)                        $valid(e^*, []) \leftarrow$
(24)                        $valid(e^*, s) \leftarrow subsetFL(s, [F_i^*]), \neg invalid_0(e^*, s)\}$
(25)                return(LP)
(26) $T_i$:             $LP = \{valid(e^*, []) \leftarrow$
(27)                        $valid(e^*, s) \leftarrow belongsFacet(t, F_i^*), narrower([t], s)\}$
(28)                return(LP)
(29)        }
```

We now expand LP'_e with the ground facts in LP_e^{fact}, as well as the following auxiliary rules, denoted by LP^{aux}:

$$\begin{array}{lcl}
belongs(t, [t|L]) & \leftarrow & \\
belongs(t, [t'\ |L]) & \leftarrow & belongs(t, L)
\end{array}$$

$$belongsFL(t, FL) \leftarrow belongs(F, FL), belongsFacet(t, F)$$

$$\begin{array}{lcl}
subsetFL(s, FL) & \leftarrow & \neg notsubsetFL(s, FL) \\
notsubsetFL(s, FL) & \leftarrow & belongs(t, s), \neg belongsFL(t, FL)
\end{array}$$

$$\begin{array}{lcl}
intersectsFL([\ ], FL, [\ ]) & \leftarrow & \\
intersectsFL([t|L1], FL, [t|L2]) & \leftarrow & belongsFL(t, FL), intersectsFL(L1, FL, L2) \\
intersectsFL([t|L1], FL, L2) & \leftarrow & \neg belongsFL(t, FL), intersectsFL(L1, FL, L2)
\end{array}$$

$$\begin{array}{lcl}
subsumed(t, t) & \leftarrow & \\
subsumed(t, t') & \leftarrow & subsumed^r(t, t_{mid}), subsumed(t_{mid}, t')
\end{array}$$

$$\begin{array}{lcl}
narrower(s, s') & \leftarrow & \neg\ notNarrower(s, s') \\
notNarrower(s, s') & \leftarrow & belongs(t', s'), \neg\ narrower_{aux}(s, t') \\
narrower_{aux}(s, t') & \leftarrow & belongs(t, s), subsumed(t, t')
\end{array}$$

We call the derived logic program, the *logic program of* e, denoted by LP_e, that is $LP_e = LP'_e \cup LP_e^{fact} \cup LP^{aux}$.

Note that in our logic programming representation, ground facts define the faceted taxonomy and the parameters of the algebraic expression, whereas program rules are generated based only on the parse tree of the algebraic expression. This makes our representation flexible and customizable, as program rules do not need to be modified in the case that facets and/or expression parameters are updated.

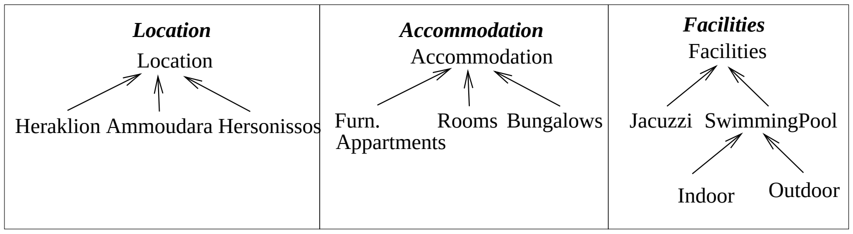

Figure 5. An example faceted taxonomy

As a running example, consider the faceted taxonomy $\mathcal{F}$ in Figure 5, indexing hotels located in Greece. A well-formed expression defining the valid compound terms of $\mathcal{F}$ is the following[6]:

$$e = (Location \ominus_N Accommodation) \oplus_P Facilities, \quad \text{where}$$

$$\begin{aligned} N = \quad & \{\{Heraklion, Bungalows\}\}, \text{ and} \\ P = \quad & \{\{Hersonissos, Rooms, Indoor\}, \\ & \{Hersonissos, Bungalows, Outdoor\}, \\ & \{Ammoudara, Bungalows, Jacuzzi\}\} \end{aligned}$$

Let $e_0 = Location \ominus_N Accommodation$. Additionally, let $\texttt{e}$, $\texttt{e}_0$, $\texttt{P}$, $\texttt{N}$, $\texttt{Loc}$, $\texttt{Acc}$, $\texttt{Fac}$ be the names of the expressions e, e_0, the parameters P, N, and the facets $Location$, $Accomodation$, and $Facilities$, respectively. Then, the ground facts defining the base predicate $belongsParam(s, param)$ are the following:

$$\begin{aligned} & belongsParam([Heraklion, Bungalows], \texttt{N}) \\ & belongsParam([Hersonissos, Rooms, Indoor], \texttt{P}) \\ & belongsParam([Hersonissos, Bungalows, Outdoor], \texttt{P}) \\ & belongsParam([Ammoudara, Bungalows, Jacuzzi], \texttt{P}) \end{aligned}$$

Moreover, the logic program of e is the following:

$$\begin{aligned} LP_e = \{\ & valid(\texttt{e}, []) && \leftarrow \\ & valid(\texttt{e}, s) && \leftarrow belongsParam(p, \texttt{P}), narrower(p, s) \\ & valid(\texttt{e}, s) && \leftarrow subsetFL(s, [\texttt{Loc}, \texttt{Acc}]), valid(\texttt{e}_0, s) \\ & valid(\texttt{e}, s) && \leftarrow subsetFL(s, [\texttt{Fac}]), valid(\texttt{Fac}, s) \\ \\ & valid(\texttt{e}_0, []) && \leftarrow \\ & valid(\texttt{e}_0, s) && \leftarrow subsetFL(s, [\texttt{Loc}, \texttt{Acc}]), \neg invalid_0(\texttt{e}_0, s), \\ & && \quad intersectsFL(s, [\texttt{Loc}], s_1), valid(\texttt{Loc}, s_1), \\ & && \quad intersectsFL(s, [\texttt{Acc}], s_2), valid(\texttt{Acc}, s_2) \end{aligned}$$

[6]Note that the faceted taxonomy $\mathcal{F}$ includes 13 terms, 890 compound terms, and 96 valid compound terms which can be specified by providing *only* 4 (carefully selected) compound terms and an appropriate algebraic expression.

$$
\begin{array}{lll}
valid(\texttt{Loc}, []) & \leftarrow & \\
valid(\texttt{Loc}, s) & \leftarrow & belongsFacet(t, \texttt{Loc}), narrower([t], s) \\
valid(\texttt{Acc}, []) & \leftarrow & \\
valid(\texttt{Acc}, s) & \leftarrow & belongsFacet(t, \texttt{Acc}), narrower([t], s) \\
valid(\texttt{Fac}, []) & \leftarrow & \\
valid(\texttt{Fac}, s) & \leftarrow & belongsFacet(t, \texttt{Fac}), narrower([t], s) \\
 & & \\
invalid_0(\texttt{e}_0, s) & \leftarrow & belongsParam(n, \texttt{N}), narrower(s, n) \\
\} \cup L_e^{fact} \cup LP^{aux} & &
\end{array}
$$

In the following proposition, we show that the SLDNF-resolution, implemented by any standard Prolog system, can be used as an inference mechanism to check if a compound term s is valid according to a well-formed expression e, that is if $s \in S_e$. Standard Prolog systems employ the computation rule which always selects the leftmost atom in a goal, together with a depth-first search rule and a fixed order for trying rules, given by their ordering in the program.

Proposition 3.1. *Let e be a well-formed expression and let $s \in P(\mathcal{T})$. It holds that:*

$$s \in S_e \ \textit{ iff } \ \exists \textit{ a successful SLDNF-derivation of } LP_e \cup \{\leftarrow valid(e^*, s)\}$$

Continuing our running example, note that the third rule of LP_e, for $s = [Heraklion, Rooms]$ succeeds. Therefore, there is a successful SLDNF-derivation of $LP_e \cup \{\leftarrow valid(\texttt{e}, [Heraklion, Rooms])\}$. Thus, from Proposition 3.1, it follows that $\{Heraklion, Rooms\} \in S_e$. Moreover, note that the first four rules of LP_e for $s = [Heraklion, Rooms, Jacuzzi]$ fail. Therefore, there is not a successful SLDNF-derivation of $LP_e \cup \{\leftarrow valid(\texttt{e}, [Heraklion, Rooms, Jacuzzi])\}$. Thus, from Proposition 3.1, it follows that $\{Heraklion, Rooms, Jacuzzi\} \notin S_e$.

We would like to note that removing the atoms $subsetFL(s, FL_{e_i})$, for $i = 1, ..., l$, from the bodies of the rules, generated at lines (4-6) of Algorithm 3.1, will not affect the correctness of Proposition 3.1. This is because, even removing these atoms, $valid(e_i^*, s)$ will succeed only if s is subset of $\mathcal{T}_{e_i}$, ($i = 1, ..., l$). Indeed, the atoms $subsetFL(s, FL_{e_i})$, for $i = 1, ..., l$, were added to the bodies of the corresponding rules for improving the efficiency of the SLDNF-resolution on LP_e, since $valid(e_i^*, s)$ will not be evaluated unnecessarily, in the case that s is not a subset of $\mathcal{T}_{e_i}$.

Similarly, the facts $valid(e_0^*, [])$ generated at lines (2), (11), (18), (23), and (26) of Algorithm 3.1, where e_0 is any subexpression of e, were added to LP_e only for efficiency reasons, as the corresponding goals $\leftarrow valid(e_0^*, [])$ will succeed even if these facts are removed from LP_e. The same is not true for the atom $subsetFL(s, FL_e)$ in the body of the rules generated at line (12) of Algorithm 3.1. Indeed, removing the atom $subsetFL(s, FL_e)$, the result of of Proposition 3.1 will no longer be correct. To see this, consider our running example and assume that $subsetFL(s, [\texttt{Loc}, \texttt{Acc}])$ is removed from the body of the rule with head $valid(\texttt{e}_0, s)$. Then, there is a successful SLDNF-derivation of $LP_{e_0} \cup \{\leftarrow valid(\texttt{e}_0, [Heraklion, Rooms, Jacuzzi])\}$, even though $\{Heraklion, Rooms, Jacuzzi\} \notin S_{e_0}$.

Up to now we have considered the general case that facets are directed acyclic graphs. However, LP_e can be optimized in the case that one or more facets have a tree-structure. In particular, if a facet F_i has a tree structure and $e_0 = \overset{*}{\oplus}_P (T_i)$ or $e_0 = T_i$ is a subexpression of e, then the rule:

$valid(e_0^*, s) \leftarrow belongsFacet(t, F_i^*), narrower([t], s),$

generated by Algorithm 3.1 at line (20) or line (27), respectively, can be replaced by the more efficient rule[7]:

$valid(e_0^*, s) \leftarrow belongs(t, s), belongsFacet(t, F_i^*), narrower([t], s).$

This is because if the facet F_i, for $i = 1, ..., k$, has a tree structure and $s \subseteq \mathcal{P}(\mathcal{T})$, it holds that the body of the unoptimized rule is satisfied iff the body of the corresponding optimized rule is satisfied. In proof, if there is a term $t \in \mathcal{T}_i$ such that $\{t\} \preceq s$ then $s \subseteq \mathcal{P}(\mathcal{T}_i)$ and there is a permutation $(t_1, ..., t_n)$ of the terms in s, such that $t \leq t_1 \leq ... \leq t_n$. Thus, there is a term t' in s (in fact, $t' = t_1$) such that $t' \in \mathcal{T}_i$ and $\{t'\} \preceq s$. Reversely, if there is a term $t \in s$ such that $t \in \mathcal{T}_i$ and $\{t\} \preceq s$ then, trivially, there is a term $t \in \mathcal{T}_i$ such that $\{t\} \preceq s$.

In the case that all facets have a tree structure, we denote the optimized logic program of e by LP_e^{tree}. In this case, the result of Proposition 1 holds also for LP_e^{tree}.

Proposition 3.2. *Let e be a well-formed expression over a set of facets with a tree structure and let $s \in P(\mathcal{T})$. It holds that:*

$$s \in S_e \textit{ iff } \exists \textit{ a successful SLDNF-derivation of } LP_e^{tree} \cup \{\leftarrow valid(e^*, s)\}$$

The following proposition gives (i) the space complexity of LP_e and LP_e^{tree} and (ii) the (worst-case) time complexity of standard Prolog's SLDNF-resolution on $LP_e \cup \{\leftarrow valid(e^*, s)\}$ and $LP_e^{tree} \cup \{\leftarrow valid(e^*, s)\}$, for deciding the validity of a compound term s. We assume that defining rules for $belongsFacet(t, facet)$ and $belongsParam(s, param)$ are indexed on the second predicate argument[8], while defining rules for $subsumed^r(t, t')$, $valid(expr, s)$, and $invalid_0(expr, s)$ are indexed on the first predicate argument. We denote the union of all P parameters in e by $\mathcal{P}$, the union of all N parameters in e by $\mathcal{N}$, and the largest compound term in $\mathcal{P} \cup \mathcal{N}$ by s_{max}.

Proposition 3.3. *Let e be a well-formed expression over the facets $\{F_1, ..., F_k\}$ and let $s \in P(\mathcal{T})$.*

1. *Deciding if there is a successful SLDNF-derivation of $LP_e \cup \{\leftarrow valid(e^*, s)\}$ takes $O(|\mathcal{T}| * |\leq^r| * |s| * |\mathcal{P} \cup \mathcal{N}|) = \mathcal{O}(|\mathcal{T}|^3 * |s| * |\mathcal{P} \cup \mathcal{N}|)$ time, while LP_e takes $O(|\mathcal{T}| + |\leq^r| + |s_{max}| * |\mathcal{P} \cup \mathcal{N}|)$=$O(|\mathcal{T}|^2 + |s_{max}| * |\mathcal{P} \cup \mathcal{N}|)$ space.*

2. *If all F_i, $i = 1, ..., k$, have a tree structure then deciding if there is a successful SLDNF-derivation of $LP_e^{tree} \cup \{\leftarrow valid(e^*, s)\}$ takes $O(|\mathcal{T}| * |s| * max(|s_{max}|, |s|) * |\mathcal{P} \cup \mathcal{N}|)$ time, while LP_e^{tree} takes $O(|\mathcal{T}| + |s_{max}| * |\mathcal{P} \cup \mathcal{N}|)$ space.*

[7]In the new rule, the computationally heavy goal $narrower([t], s)$ is evaluated only for all $t \in s$, and not for all $t \in \mathcal{T}_i$.

[8]In the case that the logic programming system supports indexing only on the first predicate argument, the order of the predicate arguments is switched, along with an appropriate change of the predicate name.

We would like to note that in [31], we presented the algorithm $IsValid(e,s)$ that checks the validity of a compound term s according to a well-formed expression e in $O(|\mathcal{T}|^3 * |s| * |\mathcal{P} \cup \mathcal{N}|)$ time, in the case that only the transitive reduction $\leq^r$ of $\leq$ is stored[9]. Thus, the complexity of the SLDNF-resolution on $LP_e \cup \{\leftarrow valid(e^*, s)\}$ is the same as that of the imperative algorithm $IsValid(e,s)$. Though the algorithm $IsValid(e,s)$ considers only the general case, its corresponding optimization, in the case that all facets have a tree structure, has the same time complexity as that of the SLDNF-resolution on $LP_e^{tree} \cup \{\leftarrow valid(e^*, s)\}$. These results make our logic programming representations LP_e and LP_e^{tree}, competitive alternatives to the corresponding imperative algorithms.

Let us denote by $ground(LP_e)$ the ground version of LP_e, where the variables of the atoms of the base predicates in the bodies of the rules are replaced according to the corresponding facts and the rest of the variables are replaced by constants or lists of constants up to size[10] $2^{|\mathcal{T}|}$. Since $\leq$ is antisymmetric, no cycle appears in the rules defining $subsumed(t,t')$ in $ground(LP_e)$. Now it is easy to see that $ground(LP_e)$ is a hierarchical propositional logic program. Therefore, $LP_e \cup \{\leftarrow valid(e^*, s)\}$ (for $s \in P(\mathcal{T})$) has either a successful SLDNF-derivation, or a finitely failed SLDNF-tree[11]. From this, Clark's results (i) and (ii), and Proposition 3.1, it follows that we can model compound term validity and invalidity in first-order logic, through $comp(LP_e)$. Specifically:

Proposition 3.4. *Let e be a well-formed expression and $s \in P(\mathcal{T})$. It holds that:*

$$\begin{array}{lll} s \in S_e & \textit{iff} & comp(LP_e) \models valid(e^*, s) \\ s \notin S_e & \textit{iff} & comp(LP_e) \models \neg valid(e^*, s) \end{array}$$

Continuing with our running example[12],

$$\begin{array}{lll}
\multicolumn{3}{l}{comp(LP_e) = \{} \\
\quad \forall ex, s\ valid(ex, s) & \leftrightarrow & (ex = \mathtt{e} \ \wedge\ s = []) \ \ \vee \\
 & & \exists p \quad (ex = \mathtt{e} \ \wedge\ belongsParam(p, \mathtt{P}) \ \wedge\ narrower(p, s)) \ \ \vee \\
 & & (ex = \mathtt{e} \ \wedge\ subsetFL(s, [\mathtt{Loc}, \mathtt{Acc}]) \ \wedge\ valid(\mathtt{e}_0, s)) \ \ \vee \\
 & & (ex = \mathtt{e} \ \wedge\ subsetFL(s, [\mathtt{Fac}]) \ \wedge\ valid(\mathtt{Fac}, s)) \ \ \vee \\
 & & \\
 & & (ex = \mathtt{e}_0 \ \wedge\ s = []) \ \ \vee \\
 & & \exists s_1, s_2 (ex = \mathtt{e}_0 \ \wedge\ subsetFL(s, [\mathtt{Loc}, \mathtt{Acc}]) \ \wedge\ \neg invalid_0(\mathtt{e}_0, s) \ \wedge \\
 & & \qquad intersectsFL(s, [\mathtt{Loc}], s_1) \ \wedge\ valid(\mathtt{Loc}, s_1) \ \wedge \\
 & & \qquad intersectsFL(s, [\mathtt{Acc}], s_2) \ \wedge\ valid(\mathtt{Acc}, s_2)) \ \ \vee \\
 & & \\
 & & (ex = \mathtt{Loc} \ \wedge\ s = []) \ \ \vee
\end{array}$$

[9] Assuming that the full subsumption relation $\leq$ is stored and indexed on any of its arguments, both the SLDNF-resolution on LP_e and the corresponding imperative algorithm have time complexity in $O(|\mathcal{T}|^2 * |s| * |\mathcal{P} \cup \mathcal{N}|)$.

[10] This is the size of the largest compound term over $\mathcal{T}$.

[11] In fact, the same is true for any goal $\{\leftarrow A\}$, where $A \in HB(LP_e)$.

[12] Note that redundant variable symbols have been eliminated from $comp(LP_e)$. Additionally, we do not show Clark's equality theory.

$$\begin{array}{ll}
& \exists t \quad (ex = \texttt{Loc} \wedge belongsFacet(t, \texttt{Loc}) \wedge narrower([t], s)) \vee \\
& (ex = \texttt{Acc} \wedge s = []) \vee \\
& \exists t \quad (ex = \texttt{Acc} \wedge belongsFacet(t, \texttt{Acc}) \wedge narrower([t], s)) \vee \\
& (ex = \texttt{Fac} \wedge s = []) \vee \\
& \exists t \quad (ex = \texttt{Fac} \wedge belongsFacet(t, \texttt{Fac}) \wedge narrower([t], s)) \\
\\
\forall ex, s \ invalid_0(ex, s) \leftrightarrow & \exists n \quad (ex = \texttt{e}_0 \wedge belongsParam(n, N) \wedge narrower(s, n)) \\
\} \bigcup comp(LP_e^{fact} \cup LP^{aux}) &
\end{array}$$

It now follows from Proposition 3.4 that a compound term s is valid according to the algebraic expression e of our running example iff $comp(LP_e) \models valid(\texttt{e}, s)$. It is easy to see that if $comp(LP_e) \models invalid_0(\texttt{e}_0, s)$ then s is invalid. However, the inverse does not hold. Indeed, let $s = \{Hersonissos, Rooms, Outdoor\}$. Then, the compound term s is invalid, yet $comp(LP_e) \not\models invalid_0(\texttt{e}_0, s)$. This is because, s is invalid due to the closed-world assumption of the $\oplus_P$ operation, and not due to the user-declared N parameter.

In general, for every well-formed expression e, it holds that

$$\begin{array}{ll}
\{s \in P(\mathcal{T}) \mid comp(LP_e) \models invalid_0(e_0^*, s)\} & \subseteq P(\mathcal{T}) \setminus S_e \\
\{s \in P(\mathcal{T}) \mid comp(LP_e) \models \neg valid(e^*, s)\} & = P(\mathcal{T}) \setminus S_e,
\end{array}$$

where e_0 is any *minus-product* or *minus-self-product* subexpression of e.

Proposition 3.4 indicates that a compound term s is valid (resp. invalid) w.r.t. a well-formed expression e iff $valid(e^*, s)$ (resp. $\neg valid(e^*, s)$) is true according to Clark's semantics [8] of LP_e. We will now show that Clark's semantics and the well-founded semantics (WFS) [12] of LP_e coincide.

Similarly to $valid(e^*, s)$, for $s \subseteq \mathcal{P}(\mathcal{T})$, we can show that for any ground atom $A \in HB(LP_e)$, it holds that $comp(LP_e) \models A$ or $comp(LP_e) \models \neg A$. Consider now the mapping that maps the predicates $notNarrower$, $notsubsetFL$, and $intersectsFL$ to number 1, the predicates $narrower$, $subsetFL$, and $invalid_0$ to number 2, the predicate $valid$ to number 3, and all the rest of the predicates appearing in LP_e to number 0. Based on this mapping, it is easy to see that LP_e is stratified. Now, as LP_e satisfies both conditions of Lemma 3.1, it follows that Clark's semantics and the WFS of LP_e coincide. Thus, the SLG-resolution [7, 6], implementing the WFS, can also be used on LP_e for checking compound term validity.

It is easy to see that Proposition 3.4 and all the later results also hold for LP_e^{tree}, in the case that all facets have a tree structure.

As a final remark, we would like to note that an algebraic expression e should be indeed well-formed, for the statements in Propositions 3.1 and 3.2 to be correct. To see this, consider the facets $A = (\{a\}, \emptyset)$, $B = (\{b\}, \emptyset)$, and $C = (\{c, c'\}, \emptyset)$. Now consider, the algebraic expression $e = (A \oplus_P B) \ominus_N C$, where $P = \{\{a, b, c\}\}$ and $N = \{\}$. As $P \not\subseteq G_{A,B} = \{\{a, b\}\}$, the expression e is not well formed. Now it is easy to see that there is not a successful SLDNF-derivation of $LP_e \cup \{\leftarrow valid(e^*, [a, b, c, c'])\}$, even though $\{a, b, c, c'\} \in S_e$. As another example, consider the expression $e' = (B \oplus_{P'} C) \ominus_{N'} C$, where $P' = \{\{b, c\}\}$ and $N' = \{\}$. As the facet C appears twice in e, the expression e is not well formed. Now it is easy to see that there is not a successful SLDNF-derivation of $LP_e \cup \{\leftarrow valid(e^*, [b, c, c'])\}$, even though $\{b, c, c'\} \in S_e$.

4. Checking well-formedness of an algebraic expression

In this section, we show how the well-formedness of an expression e can be defined in logic programming. Below, we define a set of rules, which will be used together with the logic program LP_e (or LP_e^{tree}) to determine if an expression e is well-formed.

First, we introduce three new predicates $basicCTermin(facet, s)$, $wellFormed(param)$, and $wellFormedExpr(expr)$. Intuitively, the predicate $basicCTermin(facet, s)$ represents that the compound term s belongs to the basic compound terminology of the facet with name $facet$. The predicate $wellFormed(param)$ represents that the parameter with name $param$ satisfies the constraints (ii) and (iii) of a well-formed expression given in subsection 2.2. The predicate $wFormedExpr(expr)$ expresses that the expression with name $expr$ is well-formed. To define the predicate $wellFormed(param)$, two auxiliary predicates are used.

First, we introduce a set of rules based on the parse tree of e. In these rules, parameters e_0^*, ... e_l^*, X^*, F_i^*, FL_{e_0}, ..., FL_{e_l} are replaced by constants, and symbols s, s_1,...,s_l denote variables. In particular:

- For each subexpression $e_0 = \oplus_X(e_1, ..., e_l)$ or $e_0 = \ominus_X(e_1, ..., e_l)$ of e, we introduce the rules:

$$\begin{array}{lll} wellFormed(X^*) & \leftarrow & \neg notWellFormed(X^*) \\ notWellFormed(X^*) & \leftarrow & belongsParam(s, X^*), \neg wellFormed_{aux}(e_0^*, s) \\ wellFormed_{aux}(e_0^*, s) & \leftarrow & subsetFL(s, FL_{e_0}), \neg subsetFL(s, FL_{e_1}),..., \\ & & \neg subsetFL(s, FL_{e_l}), intersectsFL(s, FL_{e_1}, s_1), \\ & & valid(e_1^*, s_1), ... , intersectsFL(s, FL_{e_l}, s_l), valid(e_l^*, s_l) \end{array}$$

- For each subexpression $e_0 = \overset{*}{\oplus}_X (T_i)$ or $e_0 = \overset{*}{\ominus}_X (T_i)$ of e, we introduce the rules:

$$\begin{array}{lll} wellFormed(X^*) & \leftarrow & \neg notWellFormed(X^*) \\ notWellFormed(X^*) & \leftarrow & belongsParam(s, X^*), \neg wellFormed_{aux}(e_0^*, s) \\ wellFormed_{aux}(e_0^*, s) & \leftarrow & subsetFL(s, [F_i^*]), \neg basicCTermin(F_i^*, s) \\ basicCTermin(F_i^*, s) & \leftarrow & belongsFacet(t, F_i^*), narrower([t], s) \end{array}$$

Moreover, we introduce the following rule:

$$wFormedExpr(e^*) \leftarrow wellFormed(X_1^*), ..., wellFormed(X_n^*),$$

where $X_1, ..., X_n$ are the parameters of e.

We call the union of the above rules and LP_e, the *CTCA logic program* of e, denoted by $CTCA_e$.

In the case that a facet F_i, for $i = 1, ..., k$, has a tree structure, then the rule:

$$basicCTermin(F_i^*, s) \leftarrow belongsFacet(t, F_i^*), narrower([t], s),$$

can be replaced by the more efficient rule:

$basicCTermin(F_i^*, s) \leftarrow belongs(t, s), narrower([t], s).$

This is because if the facet F_i, for $i = 1, ..., k$, has a tree structure and $s \subseteq \mathcal{P}(\mathcal{T}_i)$,[13] then the body of the unoptimized rule is satisfied iff the body of the corresponding optimized rule is satisfied. In proof, if there is a term $t \in \mathcal{T}_i$ such that $\{t\} \preceq s$ then there is a permutation $(t_1, ..., t_n)$ of the terms in s, such that $t \leq t_1 \leq ... \leq t_n$. Thus, there is a term t' in s (in fact, $t' = t_1$) such that $\{t'\} \preceq s$. Reversely, if there is a term $t \in s$ such that $\{t\} \preceq s$ then, trivially, there is a term $t \in \mathcal{T}_i$ such that $\{t\} \preceq s$.

In addition to the above optimization, the LP_e component of $CTCA_e$ can be optimized as indicated in subsection 3.2. In the case that all facets have a tree structure, we denote the optimized CTCA logic program of e by $CTCA_e^{tree}$. Obviously, $LP_e^{tree} \subseteq CTCA_e^{tree}$.

Let e be an algebraic expression such that each basic compound terminology T_i appears once in e. It is easy to see that:

- if $\oplus_X(e_1, ..., e_l)$ or $\ominus_X(e_1, ..., e_l)$ is a subexpression of e, and $e_1, ..., e_l$ are well-formed expressions then: $X \subseteq G_{S_{e_1},...,S_{e_l}}$ iff
  $\exists$ a successful SLDNF-derivation of $CTCA_e \cup \{\leftarrow wellFormed(X^*)\}$,
- if $\overset{*}{\oplus}_X (T_i)$ or $\overset{*}{\ominus}_X (T_i)$ is a subexpression of e then: $X \subseteq G_{T_i}$ iff
  $\exists$ a successful SLDNF-derivation of $CTCA_e \cup \{\leftarrow wellFormed(X^*)\}$.

Based on the above results, we are able to define a well-formed expression through logic programming. This is shown in the following proposition.

Proposition 4.1. *An algebraic expression e is well-formed iff:*

1. *Each basic compound terminology T_i appears once in e,*
2. *$\exists$ a successful SLDNF-derivation of $CTCA_e \cup \{\leftarrow wFormedExpr(e^*)\}$.*

Proposition 4.1 also holds for $CTCA_e^{tree}$, in the case that all facets have a tree structure. Obviously, $CTCA_e$ (resp. $CTCA_e^{tree}$) and LP_e (resp. LP_e^{tree}) have the same space complexity. The following proposition gives the time complexity of standard Prolog's SLDNF-resolution on $CTCA_e \cup \{\leftarrow wFormedExpr(e^*)\}$ and $CTCA_e^{tree} \cup \{\leftarrow wFormedExpr(e^*)\}$, for deciding the well-formedness of the expression e. We denote the union of all P parameters in e by $\mathcal{P}$, the union of all N parameters in e by $\mathcal{N}$, and the largest compound term appearing in $\mathcal{P} \cup \mathcal{N}$ by s_{max}.

Proposition 4.2. *Let e be an algebraic expression over the facets $\{F_1, ..., F_k\}$.*

1. *Deciding if there is a successful SLDNF-derivation of $CTCA_e \cup \{\leftarrow wFormedExpr(e^*)\}$ takes $O(|\mathcal{T}| * |\leq^r| * |s_{max}| * |\mathcal{P} \cup \mathcal{N}|^2) = O(|\mathcal{T}|^3 * |s_{max}| * |\mathcal{P} \cup \mathcal{N}|^2)$ time.*
2. *If all F_i, $i = 1, ..., k$, have a tree structure, deciding if there is a successful SLDNF-derivation of $CTCA_e^{tree} \cup \{\leftarrow wFormedExpr(e^*)\}$ takes $O(|\mathcal{T}| * |s_{max}|^2 * |\mathcal{P} \cup \mathcal{N}|^2)$ time.*

[13]Note that when $basicCTermin(F_i^*, s)$ is called, $subsetFL(s, [F_i^*])$ is satisfied. Therefore, $s \subseteq \mathcal{P}(\mathcal{T}_i)$.

We would like to note that the time complexity of the SLDNF-resolution on $CTCA_e \cup \{\leftarrow wFormedExpr(e^*)\}$ and $CTCA_e^{tree} \cup \{\leftarrow wFormedExpr(e^*)\}$ is the same as that of the corresponding imperative algorithms for checking the well-formedness of an expression e. Similarly to LP_e (resp. LP_e^{tree}), Clark's semantics and the WFS of $CTCA_e$ (resp. $CTCA_e^{tree}$) coincide. Thus, the SLG-resolution on $CTCA_e$ and $CTCA_e^{tree}$ can also be used for checking the well-formedness of an expression e.

5. Use Case and Discussion of other Representations

CTCA can be used in any application that indexes objects using a faceted taxonomy. For example, it can be used for designing compound taxonomies for products, for fields of knowledge (e.g. for indexing the books of a library), etc.

As we can infer the valid compound terms of a faceted taxonomy, we are able to generate a single hierarchical navigation tree *on the fly*, having only valid compound terms as nodes. The algorithm for deriving navigation trees on the fly is given in [31]. Alternatively, we can design a user interface that consists of one subwindow per facet, and guides the user through only meaningful compound term selections. Initially, a facet subwindow lists the terms of the facet at the first-level of the facet hierarchy (considering that the zero-level is the top node of the facet). The user may select a term (*selected term*) of a facet subwindow (*selected facet*). Then, all terms that do no combine with the current selection are eliminated from the remaining facet subwindows. Additionally, the list of terms in the selected-facet subwindow is replaced by the list of children of the selected term. Previous actions can now be repeated (building step-by-step a *selected compound term*), until the user specifies the valid compound term of his/her interest. Such a user interface is presented in [36]. Both of these interfaces can be used for object indexing, preventing indexing errors, as well as for object retrieval, guiding the user to only meaningful selections.

The algebra can also be used for *retrieval optimization*. For example, consider the faceted taxonomy of Figure 1, and assume that the user wants to retrieve all hotels located in Greece and offer winter sports. As $\{Islands, WinterSports\}$ is an invalid compound term, the system (optimizing execution) does not have to look for hotels located in islands at all.

Another application of the algebra is *configuration management*. Consider a product whose configuration is determined by a number of parameters, each associated with a finite number of values. However, some configurations may be unsupported, unviable, or unsafe. For this purpose, the product designer can employ an expression which specifies all valid configurations, thus ensuring that the user selects only among these.

Below we give a demonstrating example of the above applications of CTCA and a use case of our logic programming representation.

Consider a computer sales company that allows customers to order a computer with parts of their choice, through a Web portal. However, not all configurations of computer parts are valid. For example, a selected motherboard may not support all available processors, or memory types. In this case, it will be convenient to organize components into facet hierarchies and use the algebra to form an expression that specifies all valid configurations. Additionally, a facet can be added to guide the user through configurations suggested by the company for various levels of performance. An example faceted taxonomy is presented in Figure 6 consisting of 4 facets, *Processor, Motherboard, Memory,* and *Suggestions*. The facet $Processor$ is a hierarchy of Intel processors. The facet $Motherboard$ is a hierarchy of Intel motherboards. The facet $Memory$ is a hierarchy of DDR memory types. Finally, the facet $Suggestions$ is a hierarchy of performance levels (in this example, we consider only two: home-use and professional-use performance).

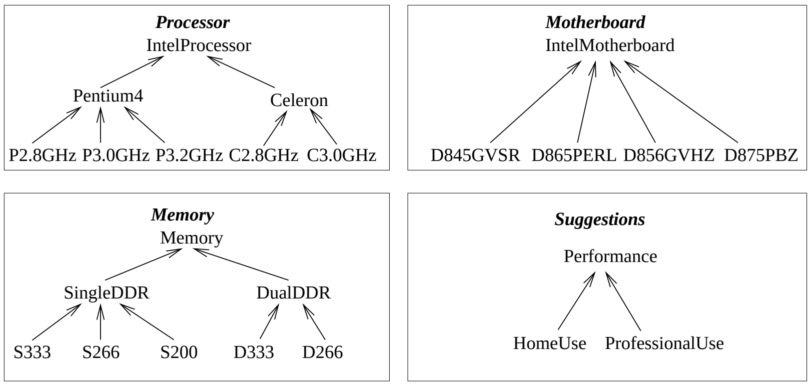

Figure 6. A faceted taxonomy for computer configurations

The subexpression $e' = \ominus_N(Processor, Motherboard, Memory)$ is the most convenient for giving all valid configurations between processors, motherboards, and memory types, as most of these components combine with only a few exceptions. For this particular example,

$$\begin{aligned} N = \quad & \{\{D845GVSR, DualDDR\}, \{D865PERL, SingleDDR\}, \\ & \{D856GVHZ, S200\}, \{D856GVHZ, DualDDR\}, \\ & \{D875PBZ, Celeron\}, \{D875PBZ, SingleDDR\}\} \end{aligned}$$

Now we have to consider the configurations suggested by the company for the various performance levels. However, as suggested configurations are only a few, it is more convenient to synthesize the previous subexpression e' with the facet $Suggestions$ with a $\oplus_P$ operation. So, the final expression e yielding all desired compound terms has the form:

$$e = \ominus_N(Processor, Motherboard, Memory) \oplus_P Suggestions$$

Though we omit the exact value of P, it holds $P \subseteq G_{S_{e'}, T_S}$, where $S_{e'}$ is the compound terminology of e' and T_S is the basic compound terminology of the facet $Suggestions$.

Now, a user interface, similar to those discussed at the beginning of this Section, can guide the user through all valid configurations, including those suggested by the company for home or professional use. For example, if the user selects from the $Suggestions$ subwindow the term $ProfessionalUse$, then the terms of the subwindows $Processor$, $Motherboard$, and $Memory$ that do not correspond to a suggested professional configuration are eliminated. The user can now proceed to browse processor, motherboard, and memory configurations for professional use, only.

Assume that at each browsing step (corresponding to a valid compound term s), a list of computers is displayed, indexed by s and satisfying certain user-declared criteria, such as maximum price ($maxPrice$) and maximum shipping days ($maxShipDays$). Consider that the company stores information about computers in the base relation $computer(computerId,\ info,\ price,\ shipDays)$, as well as computer indexing information in the base relation
$computerIndex(term, computerId)$. Then, the list of computers indexed by the desired compound term s and satisfying the user-declared criteria can be retrieved through the rules[14]:

$retrieveComputer(computerId, info, price, shipDays, s, maxPrice, maxShipDays)$
$\leftarrow \quad computerCompoundIndex(s, computerId),$
$\quad computer(computerId,\ info,\ price,\ shipDays),$
$\quad price \leq maxPrice,\ shipDays \leq maxShipDays$

$computerCompoundIndex([\,], computerId) \leftarrow$

$computerCompoundIndex([t|L], computerId)$
$\leftarrow \quad subsumed(t', t),\ computerIndex(t', computerId),$
$\quad computerCompoundIndex(L, computerId)$

The derived predicate $computerCompoundIndex(s, computerId)$ represents that the computer with id $computerId$ is indexed by the compound term s.

Naturally, more involved rules may be written to express various discount and advertisement policies, possibly in accordance to customer's previous orders and profile.

In the case that the web portal interface allows the user to select any compound term, then the $retrieveComputer$ rule can be modified as follows:

$retrieveComputer(computerId, info, price, shipDays, s, maxPrice, maxShipDays)$
$\leftarrow \quad valid(e^*, s),\ computerCompoundIndex(s, computerId),$
$\quad computer(computerId,\ info,\ price,\ shipDays),$
$\quad price \leq maxPrice,\ shipDays \leq maxShipDays$

In this case, the system first checks if s is a valid compound term, and then calls the time-consuming $computerCompoundIndex(s, computerId)$ to retrieve the ids of the computers indexed by the compound term s. Our logic programming representation can thus be embedded to such a programming environment for evaluating $valid(e^*, s)$.

In general, since compound term validity can be efficiently represented in logic programming, logic programming-based applications, taking advantage of CTCA, do not need to call external procedures for checking compound term validity. Calling an external procedure to check compound term validity implies that the logic programming application is not self-contained, restricting interchangeability between different systems.

[14]All terms appearing in these rules are variables, except e^* which is a constant.

The benefits of our logic programming representation are also strengthened by the fact that compound term validity according to an expression e cannot be efficiently represented in OWL-DL [22] and RDF(S) [18, 15]. Indeed, in [33], we show that in order to express compound term validity according to an expression e in Description Logics (DLs), either all *minus-product* and *minus-self-product* operations of e should be transformed to equivalent *plus-product* and *plus-self-product* operations, or all *plus-product* and *plus-self-product* operations of e should be transformed to equivalent *minus-product* and *minus-self-product* operations. Then, based on the transformed expression e' and the faceted taxonomy $F = (\mathcal{T}, \leq)$, a DL knowledge base $\Sigma_{e'}$ can directly be derived such that compound term validity can be defined based on the classical services of DLs [9] on $\Sigma_{e'}$[15]. However, (i) the cost of deriving e' is exponential to $|\mathcal{T}|$, (ii) the size of $\Sigma_{e'}$ is much larger than the size of LP_e, and (iii) the cost of reasoning on $\Sigma_{e'}$ is much higher than the cost of reasoning on LP_e.

Based on a similar reasoning, we can show that we can express compound term validity according to an expression e in RDFS, by transforming all *minus-product* and *minus-self-product* operations to equivalent *plus-product* and *plus-self-product* operations. Then, based on the transformed expression e' and the faceted taxonomy $F = (\mathcal{T}, \leq)$, an RDF graph $G_{e'}$ can directly be derived such that the validity of a compound term s is defined based on RDFS-entailment. Specifically, starting from $G_{e'} = \{\}$,[16]
(i) for each subsumption relation $t \leq^r t'$, the triple $\langle ex{:}t\ rdfs{:}subclass\ ex{:}t' \rangle$ is added to $G_{e'}$, expressing that the class $ex{:}t$ is a subclass of the class $ex{:}t'$,
(ii) for each $t \in \mathcal{T}$, the triple $\langle \_{:}x_t\ rdf{:}type\ ex{:}t \rangle$ is added to $G_{e'}$, expressing that there is an anonymous web resource $\_{:}x_t$ that belongs to the class $ex{:}t$, and
(iii) for each parameter P in e' and each $p = \{t_1, ..., t_n\} \in P$, the triples $\langle \_{:}x_p\ rdf{:}type\ ex{:}t_1 \rangle,\ ...,$ $\langle \_{:}x_p\ rdf{:}type\ ex{:}t_n \rangle$ are added to $G_{e'}$, expressing that there is an anonymous web resource $\_{:}x_p$ that belongs to all classes $ex{:}t_1, ..., ex{:}t_n$ (that is p is valid).

Then, the compound term $s = \{t'_1, ..., t'_n\}$ is valid according to e iff $G_{e'}$ *RDFS-entails* G_s, where $G_s = \{\langle \_{:}x\ rdf{:}type\ ex{:}t'_1 \rangle,\ ...,\ \langle \_{:}x\ rdf{:}type\ ex{:}t'_n \rangle\}$, expressing that there is an anonymous web resource $\_{:}x$ that belongs to all classes $ex{:}t'_1, ..., ex{:}t'_n$. However, this representation suffers from the same drawbacks as the representation in Description Logics.

6. Conclusion

Although faceted classification was suggested quite long ago (by Ranganathan in the 1920s [27]), the associated issues have not received adequate attention by the computer science community. However, there are several works about facet analysis (e.g. see [10], [34],[19]). Facets have also been studied in library and information science (for a review see [21]). For instance, thesauri ([17]) may have facets that group the terms of the thesaurus in classes. Ruben Prieto-Diaz ([23, 24]) has proposed "faceted classification" for a reusable software library. A faceted structure for organizing an institutional website was proposed in [25]. The contribution of the *compound term composition algebra* (CTCA) lies in enriching a faceted

[15]Specifically, *query answering*, if e' contains only positive operations, and *concept satisfiability*, if e' contains only negative operations.
[16]We assume that for each $t \in \mathcal{T}$, there is a corresponding URI reference $ex{:}t$, representing a class.

scheme with a rigorous method for specifying the valid combinations of terms. The algebra can facilitate several tasks, such as object indexing and browsing, retrieval optimization, configuration management, and consistency control.

Current interest in faceted taxonomies is also indicated by several recent projects (like FATKS[17], FACET[18], FLAMENGO[19]), and the emergence of XFML (Core-eXchangeable Faceted Metadata Language) [1]. XFML is a model to organize topics into facets, and to assign topics to any page on the web. XFML lets you publish this information in an XML format. CTCA can also be applied in this context in order to prescribe the set of valid compound terms. This can prevent some of the indexing errors that may occur in an open and collaborative environment like the Web. Indeed, the markup language XFML+CAMEL (Compound term composition Algebraically-Motivated Expression Language) was defined [2], which allows publishing and exchanging faceted taxonomies *and* expressions of the compound term composition algebra in an XML format.

This paper represents the algebra in logic programming. For each well-formed expression e, a logic program with negation LP_e is generated which defines compound term validity, in equivalence to the algebraic definition. We showed that the SLDNF-resolution [8] on LP_e can be used to check compound term validity in polynomial time. As Clark's semantics [8] and the well-founded semantics [12] of LP_e coincide, the SLG-resolution [7, 6] (implementing the well-founded semantics) can also be used for checking compound term validity. As a by-product, we also show that we can define compound term validity in first-order logic, through $comp(LP_e)$.

Additionally, we extended LP_e to the *CTCA logic program* of e ($CTCA_e$) that can be used to check well-formedness of an expression e, in addition to compound term validity, through the SLDNF-resolution and the SLG-resolution. In fact, the SLDNF-resolution on $CTCA_e$ has the same time complexity as the corresponding imperative algorithms. This result makes our logic programming representation a competitive alternative representation that can be embedded to any system seeking to capitalize on the powerful modelling and inferencing capabilities of logic programming and CTCA. An application to the implementation of a web-portal for a computer sales company is demonstrated. As Clark's semantics and the well-founded semantics of $CTCA_e$ coincide, standard Prolog systems, such as SWI-Prolog [35], and systems implementing the well-founded semantics, such as XSB[20] [30, 28], can be used for the evaluation of the desired goals. Our logic programming representation has been tested in both of these systems. In Appendix B, we give the complete CTCA logic program for the expression e of our running example, in Prolog.

A feature of our logic programming representation is that the faceted taxonomy, as well as the parameters of the algebraic expression are represented in the ground facts, whereas program rules are generated based only on the parse tree of the algebraic expression. This makes our logic programming representation flexible and customizable, as program rules do not need to be modified in the case that facets and/or

[17]http://www.ucl.ac.uk/fatks/database.htm

[18]http://www.glam.ac.uk/soc/research/hypermedia/facet_proj/index.php

[19]http://bailando.sims.berkeley.edu/flamenco.html

[20]In fact, XSB supports both standard Prolog's negation and well-founded negation, through the `not/1` and `tnot/1` operators, respectively.

expression parameters are updated. If all facets have a tree structure, an optimization is proposed and analyzed, in detail. In particular, we show that the optimized logic programs LP_e^{tree} and $CTCA_e^{tree}$ share the same properties with LP_e and $CTCA_e$, but are more efficient both in terms of space and time.

At last we should mention that a system that supports the design of faceted taxonomies and the interactive formulation of CTCA expressions has already been implemented by VTT and Helsinki University of Technology (HUT), under the name FASTAXON [32]. Future extensions of the FASTAXON system include a module for exporting the user-designed algebraic expression in XFML+CAMEL format. Parsing an expression e encoded in XFML+CAMEL format and generating the logic program $CTCA_e$ (or $CTCA_e^{tree}$) in a desired Semantic Web rule interchange format is straightforward.

Future work will concentrate on methodologies for the formulation of algebraic expressions that reflect the desire of the designer. Specifically, how the four operations of our algebra should be combined, so as the number of compound terms of the associated parameters P, N is minimal, and designer effort is minimized.

Acknowledgements: The authors would like to thank the reviewers for their valuable comments.

References

[1] "XFML: eXchangeable Faceted Metadata Language". http://www.xfml.org.

[2] "XFML+CAMEL:Compound term composition Algebraically-Motivated Expression Language". http://www.csi.forth.gr/markup/xfml+camel.

[3] J. J. Alferes, C. V. Damásio, and L. M. Pereira. "Semantic Web Logic Programming Tools". In *International Workshop on Principles and Practice of Semantic Web Reasoning (PPSWR'03)*, pages 16–32, 2003.

[4] Krzysztof R. Apt and Roland N. Bol. "Logic Programming and Negation: A Survey". *Journal of Logic Programming*, 19/20:9–71, 1994.

[5] S. Battle, A. Bernstein, H. Boley, B. Grosof, M. Gruninger, R. Hull, M. Kifer, D. Martin, S. McIlraith, D. McGuinness, J. Su, and S. Tabet. "Semantic Web Services Language (SWSL)". W3C Member Submission, 9 September 2005. Available at `http://www.w3.org/Submission/2005/SUBM-SWSF-SWSL-20050909/`.

[6] W. Chen, T. Swift, and D. S. Warren. "Efficient Top-Down Computation of Queries under the Well-Founded Semantics". *Journal of Logic Programming*, 24(3):161–199, 1995.

[7] W. Chen and D. S. Warren. "Query Evaluation under the Well Founded Semantics". In *12th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS-1993)*, pages 168–179. ACM Press, May 1993.

[8] K. L. Clark. " Negation as Failure". In *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[9] F.M. Donini, M. Lenzerini, D. Nardi, and A. Schaerf. "Reasoning in Description Logics". In Gerhard Brewka, editor, *Principles of Knowledge Representation*, chapter 1, pages 191–236. CSLI Publications, 1996.

[10] Elizabeth B. Duncan. "A Faceted Approach to Hypertext". In Ray McAleese, editor, *HYPERTEXT: theory into practice, BSP*, pages 157–163, 1989.

[11] P. A. Fejer and D. A. Simovici. *“Mathematical Foundations of Computer Science. Volume 1: Sets, Relations, and Induction”*. Springer-Verlag, 1991.

[12] A. Van Gelder, K. A. Ross, and J. S. Schlipf. “The Well-Founded Semantics for General Logic Programs”. *Journal of the ACM*, 38(3):620–650, 1991.

[13] B. N. Grosof. “Representing e-commerce rules via situated courteous logic programs in RuleML”. *Electronic Commerce Research and Applications*, 3(1):2–20, 2004.

[14] B. N. Grosof, Y. Labrou, and H. Y. Chan. “A declarative approach to business rules in contracts: courteous logic programs in XML”. In *1st ACM Conference on Electronic Commerce (EC-1999)*, pages 68–77, 1999.

[15] Patrick Hayes. “RDF Semantics”. W3C Recommendation, 10 February 2004. Available at `http://www.w3.org/TR/2004/REC-rdf-mt-20040210/`.

[16] I. Horrocks and P. F. Patel-Schneider. “Reducing OWL Entailment to Description Logic Satisfiability”. In *2nd International Semantic Web Conference (ISWC-2003)*, pages 17–29, October 2003.

[17] International Organization For Standardization. “Documentation - Guidelines for the establishment and development of monolingual thesauri”, 1986. Ref. No ISO 2788-1986.

[18] G. Klyne and J. J. Carroll. “Resource Description Framework (RDF): Concepts and Abstract Syntax”. W3C Recommendation, 10 February 2004. Available at `http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/`.

[19] P. H. Lindsay and D. A. Norman. *Human Information Processing*. Academic press, New York, 1977.

[20] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, second edition, 1987.

[21] Amanda Maple. “Faceted Access: A Review of the Literature”, 1995. http://theme.music.indiana.edu/tech_s/mla/facacc.rev.

[22] D. L. McGuinness and F. van Harmelen. “OWL Web Ontology Language Overview”. W3C Recommendation, 10 February 2004. Available at `http://www.w3.org/TR/2004/REC-owl-features-20040210/`.

[23] Ruben Prieto-Diaz. “Classification of Reusable Modules”. In *Software Reusability. Volume I*, chapter 4, pages 99–123. ACM Press, 1989.

[24] Ruben Prieto-Diaz. “Implementing Faceted Classification for Software Reuse”. *Communications of the ACM*, 34(5):88–97, 1991.

[25] U. Priss and E. Jacob. “Utilizing Faceted Structures for Information Systems Design”. In *Proceedings of the ASIS Annual Conf. on Knowledge: Creation, Organization, and Use (ASIS’99)*, October 1999.

[26] T. C. Przymusinski. “Well-founded and Stationary Models of Logic Programs”. *Annals of Mathematics and Artificial Intelligence*, 12(3-4):141–187, 1994.

[27] S. R. Ranganathan. “The Colon Classification”. In Susan Artandi, editor, *Vol IV of the Rutgers Series on Systems for the Intellectual Organization of Information*. New Brunswick, NJ: Graduate School of Library Science, Rutgers University, 1965.

[28] P. Rao, K. F. Sagonas, T. Swift, D. S. Warren, and J. Freire. “XSB: A System for Efficiently Computing WFS”. In *Proceedings of 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR’97)*, pages 1070–1080, July 1997.

[29] "The Rule Markup Initiative (RuleML)". Available at `http://www.ruleml.org`.

[30] K. Sagonas, T. Swift, and D. S. Warren. "XSB as an efficient deductive database engine". In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data (SIGMOD'94)*, pages 442–453, 1994.

[31] Y. Tzitzikas, A. Analyti, N. Spyratos, and P. Constantopoulos. "An Algebraic Approach for Specifying Compound Terms in Faceted Taxonomies". In *Information Modelling and Knowledge Bases XV, 13th European-Japanese Conference on Information Modelling and Knowledge Bases, EJC'03*, pages 67–87. IOS Press, 2004.

[32] Y. Tzitzikas, R. Launonen, M. Hakkarainen, P. Kohonen, T. Leppanen, E. Simpanen, H. Tornroos, P. Uusitalo, and P. Vanska. "FASTAXON: A system for FAST (and Faceted) TAXONomy design". In *Procs. of 23th Int. Conf. on Conceptual Modeling, ER'2004*, Shanghai, China, November 2004. (an on-line demo is available at http://fastaxon.erve.vtt.fi/).

[33] Yannis Tzitzikas, Anastasia Analyti, and Nicolas Spyratos. "Compound Term Composition Algebra: The Semantics". *LNCS Journal on Data Semantics*, 2:58–84, 2005.

[34] B. C. Vickery. "Knowledge Representation: A Brief Review". *Journal of Documentation*, 42(3):145–159, 1986.

[35] Jan Wielemaker. "An overview of the SWI-Prolog Programming Environment". In *13th International Workshop on Logic Programming Environments*, pages 1–16, December 2003.

[36] K. Yee, K. Swearingen, K. Li, and M. Hearst. "Faceted Metadata for Image Search and Browsing". In *Proceedings of the Conf. on Human Factors in Computing Systems (CHI'03)*, pages 401–408, April 2003.

Appendix A: Proofs

In this Appendix, we give the proofs of the lemmas and propositions appearing in the paper.

Lemma 3.1 Let P be a stratified logic program such that for all $A \in HB(P)$, it holds that $comp(P) \models A$ or $comp(P) \models \neg A$. Then, $CLARK(P) = WFS(P)$.
Proof: In [26], it is shown that if P is (locally) stratified then $CLARK(P) \subseteq PERFECT(P) = WFS(P)$. We will show that if $comp(P) \models A$ or $comp(P) \models \neg A$, $\forall A \in HB(P)$, then it also holds that $WFS(P) \subseteq CLARK(P)$.

Let $L \in WFS(P)$. Assume that $L \notin CLARK(P)$. Then, $comp(P) \models \neg L$. Thus, $\neg L \in CLARK(P)$. But then $\neg L \in WFS(P)$, which is impossible. $\diamond$

Proposition 3.1 Let e be a well-formed expression and $s \in P(\mathcal{T})$. It holds that:

$$s \in S_e \text{ iff } \exists \text{ a successful SLDNF-derivation of } LP_e \cup \{\leftarrow valid(e^*, s)\}$$

Proof: We will prove Proposition 3.1 by induction.
Let $e = T_i$.
$\Rightarrow$) Assume that $s \in S_e$. Then, $s \in T_i$. Thus, $\exists t \in \mathcal{T}_i$ such that $\{t\} \preceq s$. Therefore, from the rule generated in line (27) of Algorithm 3.1, there is a successful SLDNF-derivation of $LP_e \cup \{\leftarrow valid(e^*, s)\}$.

$\Leftarrow$) Assume that there is a successful SLDNF-derivation of $LP_e \cup \{\leftarrow valid(e^*, s)\}$. As LP_e consists of only two rules, generated in lines (26), (27) of Algorithm 3.1, it holds that $s = \{\}$ or that $\exists t \in \mathcal{T}_i$ such that $\{t\} \preceq s$. Thus, $s \in T_i$. Therefore, $s \in S_e$.
In a similar manner, we can easily prove Proposition 3.1, for $e = \overset{*}{\oplus}_P\ (T_i)$ and $e = \overset{*}{\ominus}_N\ (T_i)$.
Assumption: Assume now that the proposition holds for the subexpressions of $e_1, ..., e_l$ of e.

Let $e = \oplus_P(e_1, ..., e_l)$.
$\Rightarrow$) Assume that $s \in S_e$. If $s \in S_{e_i}$, for an $i = 1, ..., l$, then $s \subseteq \mathcal{T}_i$ and there is a successful SLDNF-derivation of $LP_{e_i} \cup \{\leftarrow valid(e_i^*, s)\}$ (due to the $Assumption$). Therefore, from the rules generated in lines (4-6) of Algorithm 3.1 at the level[21] of e, there is a successful SLDNF-derivation of $LP_e \cup \{\leftarrow valid(e^*, s)\}$. If $s \notin S_{e_i}$, $\forall i = 1, ..., l$, then $s \in Br(P)$ and there is a successful SLDNF-derivation of $LP_e \cup \{\leftarrow valid(e^*, s)\}$ from the rule generated in line (3) of Algorithm 3.1 at the level of e.
$\Leftarrow$) Assume that there is a successful SLDNF-derivation of $LP_e \cup \{\leftarrow valid(e^*, s)\}$. If there is a successful SLDNF-derivation of $LP_{e_i} \cup \{\leftarrow valid(e_i^*, s)\}$, for an $i = 1, ..., l$, then $s \in S_{e_i}$ (due to the $Assumption$), which implies $s \in S_e$. Otherwise, $s \neq \{\}$ and the rule generated in line (3) of Algorithm 3.1 at the level of e should be used in the successful SLDNF-derivation. Therefore, $s \in Br(P)$, which implies $s \in S_e$.

Let $e = \ominus_N(e_1, ..., e_l)$.
$\Rightarrow$) Assume that $s \in S_e$. Since e is a well-formed expression, it follows that $s \subseteq \mathcal{T}_e$, $s \cap \mathcal{T}_i \in S_{e_i}$, $\forall i = 1, ..., l$, and $s \notin Nr(N)$. Therefore, there is a successful SLDNF-derivation of $LP_{e_i} \cup \{\leftarrow valid(e_i^*, s \cap \mathcal{T}_i)\}$, $\forall i = 1, ..., l$ (due to the $Assumption$), and a finitely failed SLDNF-tree for $LP_e \cup \{\leftarrow invalid_0(e^*, s)\}$. Thus, there is a successful SLDNF-derivation of $LP_e \cup \{\leftarrow valid(e^*, s)\}$.
$\Leftarrow$) Assume that there is a successful SLDNF-derivation of $LP_e \cup \{\leftarrow valid(e^*, s)\}$. If $s = \{\}$ then $s \in S_e$. Assume that $s \neq \{\}$. Then, necessarily the successful SLDNF-derivation uses the rule generated in line (12) of Algorithm 3.1 at the level of e. Thus, $s \subseteq \mathcal{T}_e$, and there is a successful SLDNF-derivation of $LP_e \cup \{\leftarrow valid(e_i^*, s \cap \mathcal{T}_{e_i})\}$, $\forall i = 1, ..., l$. Additionally, there is a finitely failed SLDNF-tree for $LP_e \cup \{\leftarrow invalid_0(e^*, s)\}$. Since in the evaluation of $valid(e_i^*, s \cap \mathcal{T}_{e_i})\}$ on LP_e, for $i = 1, ..., l$, only the rules in LP_{e_i} are used, it follows that there is a successful SLDNF-derivation of $LP_{e_i} \cup \{\leftarrow valid(e_i^*, s \cap \mathcal{T}_{e_i})\}$, $\forall i = 1, ..., l$. Thus, $s \subseteq \mathcal{T}_e$, $s \cap \mathcal{T}_{e_i} \in S_{e_i}, \forall i = 1, ..., l$, and $s \notin Nr(N)$. Therefore, it follows that $s \in S_e$. $\diamond$

Proposition 3.2 Let e be a well-formed expression over a set of facets with a tree structure and let $s \in P(\mathcal{T})$. It holds that:

$$s \in S_e \text{ iff } \exists \text{ a successful SLDNF-derivation of } LP_e^{tree} \cup \{\leftarrow valid(e^*, s)\}$$

Proof: We will prove Proposition 3.2 by induction.
Let $e = T_i$.
$\Rightarrow$) Assume that $s \in S_e$. Then, $s \in T_i$. Thus, there is a term $t \in \mathcal{T}_i$ such that $\{t\} \preceq s$. Now since F_i has a tree structure, it follows that there is a permutation $(t_1, ..., t_n)$ of the terms in s, such that $t \leq t_1 \leq ... \leq t_n$. Thus, there is a term t' in s (in fact, $t' = t_1$) such that $t' \in \mathcal{T}_i$ and $\{t'\} \preceq s$. Therefore, from the rule:

$valid(e^*, s) \leftarrow belongs(t, s), belongsFacet(t, F_i^*), narrower([t], s),$

it follows that there is a successful SLDNF-derivation of $LP_e^{tree} \cup \{\leftarrow valid(e^*, s)\}$.
$\Leftarrow$) Assume that there is a successful SLDNF-derivation of $LP_e^{tree} \cup \{\leftarrow valid(e^*, s)\}$. As LP_e^{tree} consists of only two rules:

[21]Rules *generated at the level of* e are the rules generated by Algorithm $GenLP'(e)$ before recursive calls are made.

$valid(e^*, []) \leftarrow$, and
$valid(e^*, s) \leftarrow belongs(t, s), belongsFacet(t, F_i^*), narrower([t], s)$,

it holds that $s = \{\}$ or that $\exists t \in \mathcal{T}_i$ such that $\{t\} \preceq s$. Thus, $s \in T_i$. Therefore, $s \in S_e$.

In a similar manner, we can easily prove Proposition 3.2, for $e = \overset{*}{\oplus}_P (T_i)$ and $e = \overset{*}{\ominus}_N (T_i)$.

The rest of the proof continues as the proof of Proposition 3.1 (starting from *Assumption*). ⋄

Proposition 3.3 Let e be a well-formed expression over the facets $\{F_1, ..., F_k\}$ and let $s \in P(\mathcal{T})$.

1. Deciding if there is a successful SLDNF-derivation of $LP_e \cup \{\leftarrow valid(e^*, s)\}$ takes $O(|\mathcal{T}| * |\leq^r| * |s| * |\mathcal{P} \cup \mathcal{N}|) = \mathcal{O}(|\mathcal{T}|^3 * |s| * |\mathcal{P} \cup \mathcal{N}|)$ time, while LP_e takes $O(|\mathcal{T}| + |\leq^r| + |s_{max}| * |\mathcal{P} \cup \mathcal{N}|)$=$O(|\mathcal{T}|^2 + |s_{max}| * |\mathcal{P} \cup \mathcal{N}|)$ space.
2. If all F_i, $i = 1, ..., k$, have a tree structure then deciding if there is a successful SLDNF-derivation of $LP_e^{tree} \cup \{\leftarrow valid(e^*, s)\}$ takes $O(|\mathcal{T}| * |s| * max(|s_{max}|, |s|) * |\mathcal{P} \cup \mathcal{N}|)$ time, while LP_e^{tree} takes $O(|\mathcal{T}| + |s_{max}| * |\mathcal{P} \cup \mathcal{N}|)$ space.

Proof:
1) Let $VT(e)$ be the maximum cost for deciding if there is a successful SLDNF-derivation of $LP_e \cup \{\leftarrow valid(e^*, s)\}$. In the worst case[22], all of the following goals have to be evaluated:

(i) $belongsParam(x, X^*)$, $\forall X \in Params(e)$, where x is a variable.

(ii) $narrower(p, s')$, $\forall p \in \mathcal{P}$, where s' is a constant $s' \subseteq s$.

(iii) $narrower(s', n)$, $\forall n \in \mathcal{N}$, where s' is a constant $s' \subseteq s$.

(iv) $narrower([t], s_i)$, $\forall t \in \mathcal{T}_i$, $\forall i = 1, ..., k$, where k is the number of facets and s_i is a constant $s_i \subseteq s$.

(v) $subsetFL(s', FL_{e_i})$, $\forall$ subexpressions e_i of e, where s' is a constant $s' \subseteq s$.

(vi) $intersectsFL(s', FL_{e_i}, x)$, $\forall$ subexpressions e_i of e, where s' is a constant $s' \subseteq s$ and x is a variable.

(vii) $belongsFacet(t, F_i^*)$, $\forall i = 1, ..., k$, where t is a variable.

We will first compute the complexity of the goals in (i). Since the facts $belongParam(s, param)$ are indexed on the second argument, the evaluation of $belongsParam(x, X^*)$, where $X \in Params(e)$ and x is a variable, takes $O(|X|)$ time. Thus, the complexity of all goals in (i) is $|\mathcal{P} \cup \mathcal{N}|$.

We will now compute the complexity of the goals in (ii) and (iii). Let s_1, s_2 be compound terms. To evaluate $narrower(s_1, s_2)$, the goal $subsumed(t_1, t_2)$ is evaluated, at most for all $t_2 \in s_2$ and $t_1 \in s_1$. We will now compute the complexity of the goal $subsumed(t_1, t_2)$, where t_1, t_2 are constants. To evaluate $subsumed(t_1, t_2)$, the goal $subsumed^r(t, t_{mid})$, where t is a constant and t_{mid} is a variable, is called at least for all terms t broader than t_1. Since the facts $subsumed^r(t, t')$ are indexed on the first argument, it follows that the complexity of the goal $subsumed(t_1, t_2)$, where t_1, t_2 are constants, is $O(|\leq^r|)$. Thus, the complexity of the goal $narrower(s_1, s_2)$ is $O(|\leq^r| * |s_1| * |s_2|)$.

As we have $|\mathcal{P} \cup \mathcal{N}|$ goals of the form $narrower(p, s')$ and $narrower(s', n)$, where $p, n \in \mathcal{P} \cup \mathcal{N}$ and $|s'| \leq |s|$, the complexity of all goals in (ii) and (iii) is $O(|\leq^r| * |s_{max}| * |s| * |\mathcal{P} \cup \mathcal{N}|)$.

Now we will compute the complexity of the goals in (iv). Suppose that i is fixed. As we have shown above, the call $narrower([t], s_i)$, for a term $t \in \mathcal{T}_i$ and $s_i \subseteq s$, takes $O(|\leq^r| * |s|)$ time. Thus, the complexity of the goals

[22]Note that the worst case complexity of LP_e is not affected if all negative literals $\neg A$ in the body of the rules are replaced by the atoms A.

$narrower([t], s_i)$, $\forall t \in \mathcal{T}_i$, is $O(|\mathcal{T}_i| * |\leq^r| * |s|)$. Therefore, the complexity of all goals in (iv) is $O(k * |\mathcal{T}| * |\leq^r| * |s|)$.

Now we will compute the complexity of the goals in (v). To evaluate $subsetFL(s', FL_{e_i})$, where e_i is a subexpression of e, the goal $belongsFL(t, FL_{e_i})$ is evaluated, for all $t \in s'$. To evaluate the goal $belongsFL(t, FL_{e_i})$, where t is a constant, the goal $belongsFacet(t, F_j^*)$ is evaluated for all $F_j^* \in FL_{e_i}$. Now, since the facts $belongsFacet(t, facet)$ are indexed on the second argument, the goal $belongsFacet(t, F_j^*)$ takes $O(|\mathcal{T}_j|)$ time. Therefore, the goal $belongsFL(t, FL_{e_i})$ takes $O(|\mathcal{T}_{e_i}|)$ time. Thus, the goal $subsetFL(s', FL_{e_i})$ takes $O(|s'| * |\mathcal{T}_{e_i}|)$ time. Therefore, the complexity of all goals in (v) is $O(k * |s| * |\mathcal{T}|)$.

Now we will compute the complexity of the goals in (vi). To evaluate $intersectsFL(s', FL_{e_i}, x)$, where e_i is a subexpression of e and x is a variable, the goal $belongsFL(t, FL_{e_i})$ is evaluated, for all $t \in s'$. Thus, the goal $intersectsFL(s', FL_{e_i}, x)$ takes $O(|s'| * |\mathcal{T}_{e_i}|)$ time. Therefore, the complexity of all goals in (vi) is $O(k * |s| * |\mathcal{T}|)$.

Now since the facts $belongsFacet(t, facet)$ are indexed on the second argument, the complexity of all goals in (vii) is $O(|\mathcal{T}|)$.

In total, $VT(e) = O(|\mathcal{P} \cup \mathcal{N}| + |\leq^r| * |s_{max}| * |s| * |\mathcal{P} \cup \mathcal{N}| + k * |\mathcal{T}| * |\leq^r| * |s| + k * |s| * |\mathcal{T}| + |\mathcal{T}|)$. Note that $O(|\leq^r|) = O(|\mathcal{T}|^2)$,[23] and $|s_{max}| \leq |\mathcal{T}|$. Therefore, assuming that in general $k \leq |\mathcal{P} \cup \mathcal{N}|$, it follows that: $VT(e) = O(|\mathcal{T}| * |\leq^r| * |s| * |\mathcal{P} \cup \mathcal{N}|) = O(|\mathcal{T}|^3 * |s| * |\mathcal{P} \cup \mathcal{N}|)$.

We will now compute the size of LP_e. Assuming that each term has a manimum number of characters, the size of the facts $belongsFacet(t, facet)$ is $O(|\mathcal{T}|)$. The size of the facts $subsumed^r(t, t')$ is $O(|\leq^r|)$. The size of the facts $belongsParam(s, param)$ is $O(|\mathcal{P} \cup \mathcal{N}| * |s_{max}|)$. We will now compute the size of LP'_e. Since e is well-formed, the number of subexpressions of e is at most $2 * k - 1$. Therefore, the size of all the rules in LP'_e, generated at lines (4-6) and (12) of Algorithm 3.1 is $O(k)$. Additionally, the size of the rest of the rules in LP_e is also $O(k)$. Thus, the size of LP'_e is $O(k)$. It is easy to see that the size of LP^{aux} is constant. Since $k \leq |\mathcal{T}|$, it follows that the size of LP_e is $O(|\mathcal{T}| + |\leq^r| + |s_{max}| * |\mathcal{P} \cup \mathcal{N}|) = O(|\mathcal{T}|^2 + |s_{max}| * |\mathcal{P} \cup \mathcal{N}|)$.

2) Assume now that all facets F_i, $i = 1, ..., k$, have a tree structure. Let $VT^{tree}(e)$ be the maximum cost for deciding if there is a successful SLDNF-derivation of $LP_e^{tree} \cup \{\leftarrow valid(e^*, s)\}$. In the worst case, all of the following goals have to be evaluated:

(i) $belongsParam(x, X^*)$, $\forall X \in Params(e)$, where x is a variable.

(ii) $narrower(p, s'), \forall p \in \mathcal{P}$, where s' is a constant $s' \subseteq s$.

(iii) $narrower(s', n), \forall n \in \mathcal{N}$, where s' is a constant $s' \subseteq s$.

(iv) $narrower([t], s_i)$, $\forall t \in s_i$, $\forall i = 1, ..., k$, where s_i is a constant $s_i \subseteq s$.

(v) $subsetFL(s', FL_{e_i})$, $\forall$ subexpressions e_i of e, where s' is a constant $s' \subseteq s$.

(vi) $intersectsFL(s', FL_{e_i}, x)$, $\forall$ subexpressions e_i of e, where s' is a constant $s' \subseteq s$ and x is a variable.

(vii) $belongsFacet(t, F_i^*)$, $\forall i = 1, ..., k$, where t is a variable.

The complexity of the goals in (i), (ii), (iii), (v), (vi), and (vii) is as in the proof of Proposition 3.3(1).

Now we will compute the complexity of the goals in (iv). Assume that i is fixed. As we have shown in the proof of Proposition 3.3(1), the call $narrower([t], s_i)$, for a term $t \in s_i$, takes $O(|\leq^r| * |s_i|)$ time. Thus, the

[23]Indeed, consider $2 * n$ terms $\{t_1, ..., t_n, t'_1, ..., t'_n\}$ such that $t_i \leq t'_j$, $\forall i, j \in \{1, ..., n\}$. Then, $\leq^r = \leq$. From this, it follows that $O(|\leq^r|) = O(n^2)$.

complexity of the goals $narrower([t], s_i)$, $\forall t \in s_i \subseteq s$, is $O(|\leq^r| * |s|^2)$. Therefore, the complexity of all goals in (iv) is $O(k * |\leq^r| * |s|^2)$.

In total, $VT^{tree}(e) = O(|\mathcal{P} \cup \mathcal{N}| + |\leq^r| * |s_{max}| * |s| * |\mathcal{P} \cup \mathcal{N}| + k * |\leq^r| * |s|^2 + k * |s| * |\mathcal{T}| + |\mathcal{T}|)$. Since the number of edges of a tree is less than the number of nodes, it follows that $O(|\leq^r|) = O(|\mathcal{T}|)$. Therefore, assuming that in general $k \leq |\mathcal{P} \cup \mathcal{N}|$, it follows that $VT(e) = O(|\mathcal{T}| * |s| * max(|s_{max}|, |s|) * |\mathcal{P} \cup \mathcal{N}|)$.

Since $O(|\leq^r|) = O(|\mathcal{T}|)$, it follows similarly to the proof for the size of LP_e in Proposition 3.3(1) that the size of LP_e^{tree} is $O(|\mathcal{T}| + |s_{max}| * |\mathcal{P} \cup \mathcal{N}|)$. $\diamond$

Proposition 3.4 Let e be a well-formed expression and $s \in P(\mathcal{T})$. It holds that:

$$\begin{array}{lll} s \in S_e & \text{iff} & comp(LP_e) \models valid(e^*, s) \\ s \notin S_e & \text{iff} & comp(LP_e) \models \neg valid(e^*, s) \end{array}$$

Proof:
First statement:
$\Rightarrow$) Let $s \in S_e$. Then from Proposition 3.1, $\exists$ a successful SLDNF-derivation of $LP_e \cup \{\leftarrow valid(e^*, s)\}$. From Clark's result (i), it follows that $comp(LP_e) \models valid(e^*, s)$.
$\Leftarrow$) Assume that $comp(LP_e) \models valid(e^*, s)$. If $\exists$ a finitely failed SLDNF-tree of $LP_e \cup \{\leftarrow valid(e^*, s)\}$ then, due to Clark's result (ii), $comp(LP_e) \models \neg valid(e^*, s)$, which is impossible. Therefore, $\exists$ a successful SLDNF-derivation of $LP_e \cup \{\leftarrow valid(e^*, s)\}$, and due to Proposition 3.1, $s \in S_e$.

Second statement:
$\Rightarrow$) Let $s \notin S_e$. Then from Proposition 3.1, it follows that $\not\exists$ a successful SLDNF-derivation of $LP_e \cup \{\leftarrow valid(e^*, s)\}$. Therefore, $\exists$ a finitely failed SLDNF-tree of $LP_e \cup \{\leftarrow valid(e^*, s)\}$. For Clark's result (ii), it follows that $comp(LP_e) \models \neg valid(e^*, s)$.
$\Leftarrow$) Assume that $comp(LP_e) \models \neg valid(e^*, s)$. If $\exists$ a successful SLDNF-derivation of $LP_e \cup \{\leftarrow valid(e^*, s)\}$, then due to Clark's result (i), $comp(LP_e) \models valid(e^*, s)$, which is impossible. Therefore, due to Proposition 3.1, $s \notin S_e$. $\diamond$

Proposition 4.1 An algebraic expression e is well-formed iff:

1. Each basic compound terminology T_i appears once in e,
2. $\exists$ a successful SLDNF-derivation of $CTCA_e \cup \{\leftarrow wFormedExpr(e^*)\}$.

Proof: We will prove Proposition 4.1 by induction, using the results presented just before Proposition 4.1 in Section 4.
Let $e = \overset{*}{\oplus}_X (T_i)$ or $e = \overset{*}{\ominus}_X (T_i)$. If $\exists$ a successful SLDNF-derivation of $CTCA_e \cup \{\leftarrow wFormedExpr(e^*)\}$ then $\exists$ a successful SLDNF-derivation of $CTCA_e \cup \{\leftarrow wellFormed(X^*)\}$. Thus, $X \subseteq G_{T_i}$ holds. Therefore, e is well-formed. Reversely, if e is well-formed then $X \subseteq G_{T_i}$ holds. Therefore, $\exists$ a successful SLDNF-derivation of $CTCA_e \cup \{\leftarrow wellFormed(X^*)\}$. Thus, $\exists$ a successful SLDNF-derivation of $CTCA_e \cup \{\leftarrow wFormedExpr(e^*)\}$.

Assumption: Assume now that the proposition holds for subexpressions $e_1, ..., e_l$ of e.

Let $e = \oplus_X(e_1, ..., e_l)$ or $e = \ominus_X(e_1, ..., e_l)$.

$\Rightarrow$) If e is well-formed then 1) holds and $e_1, ..., e_l$ are well-formed. Additionally, $X \subseteq G_{S_{e_1},...,S_{e_l}}$. Based on the *Assumption*, it follows that $\exists$ a successful SLDNF-derivation of $CTCA_{e_i} \cup \{\leftarrow wFormedExpr(e_i^*)\}$, for all $i = 1, ..., l$. Therefore, for all $i = 1, ..., l$, $\exists$ a successful SLDNF-derivation of $CTCA_{e_i} \cup \{\leftarrow wellFormed(Y^*)\}$, for all $Y \in Params(e_i)$. Additionally, based on the results presented just before Proposition 4.1, $\exists$ a successful SLDNF-derivation of $CTCA_e \cup \{\leftarrow wellFormed(X^*)\}$. Therefore, $\exists$ a successful SLDNF-derivation of $CTCA_e \cup \{\leftarrow wellFormed(X_i^*)\}$, for all $X_i \in Params(e)$, and thus 2) holds.
$\Leftarrow$) Assume that 1) and 2) hold. Then $\exists$ a successful SLDNF-derivation of $CTCA_e \cup \{\leftarrow wellFormed(X_i^*)\}$, for all $X_i \in Params(e)$. Therefore, for all $i = 1, ..., l$, $\exists$ a successful SLDNF-derivation of $CTCA_{e_i} \cup \{\leftarrow wellFormed(Y^*)\}$, for all $Y \in Params(e_i)$. Therefore, $\exists$ a successful SLDNF-derivation of $CTCA_{e_i} \cup \{\leftarrow wFormedExpr(e_i^*)\}$, for all $i = 1, ..., l$. Based on the *Assumption*, it follows that $e_1, ..., e_l$ are well-formed. Additionally, from 2) it follows that $\exists$ a successful SLDNF-derivation of $CTCA_e \cup \{\leftarrow wellFormed(X^*)\}$. Thus, based on the results presented just before Proposition 4.1, $X \subseteq G_{S_{e_1},...,S_{e_l}}$. From this, 1), and the fact that $e_1, ..., e_l$ are well-formed, it follows that e is well-formed. $\diamond$

Proposition 4.2 Let e be an algebraic expression over the facets $\{F_1, ..., F_k\}$.

1. Deciding if there is a successful SLDNF-derivation of $CTCA_e \cup \{\leftarrow wFormedExpr(e^*)\}$ takes $O(|\mathcal{T}| * |\leq^r| * |s_{max}| * |\mathcal{P} \cup \mathcal{N}|^2)$= $O(|\mathcal{T}|^3 * |s_{max}| * |\mathcal{P} \cup \mathcal{N}|^2)$ time.
2. If all F_i, $i = 1, ..., k$, have a tree structure, deciding if there is a successful SLDNF-derivation of $CTCA_e^{tree} \cup \{\leftarrow wFormedExpr(e^*)\}$ takes $O(|\mathcal{T}| * |s_{max}|^2 * |\mathcal{P} \cup \mathcal{N}|^2)$ time.

Proof:
1) Let $WFT(e)$ be the maximum cost[24] for deciding if there is a successful SLDNF-derivation of $CTCA_e \cup \{\leftarrow wFormedExpr(e^*)\}$.
We distinguish two cases:
Case 1) Let $e_0 = \oplus_X(e_1, ..., e_l)$ or $e_0 = \ominus_X(e_1, ..., e_l)$ be a subexpression of e.
To evaluate $wellFormed(X^*)$, all of the following goals have to be evaluated:

(i) $belongsParam(x, X^*)$, where x is a variable.

(ii) $\neg subsetFL(s, FL_{e_0})$, $\forall s \in X$, and
$subsetFL(s, FL_{e_i})$, $\forall s \in X,\ \forall i = 1, ..., l$.

(iii) $intersectsFL(s, FL_{e_i}, x)$, $\forall s \in X,\ \forall i = 1, ..., l$, where x is a variable.

(iv) $valid(e_i^*, s \cap \mathcal{T}_{e_i})$, $\forall i = 1, ..., l,\ \forall s \in X$.

According to the proof of Proposition 3.3(1), we have the following complexity results. The complexity of the goal in (i) is $O(|X|)$. The complexity of all goals in (ii) is $O(|X| * |s_{max}| * |\mathcal{T}|)$. The complexity of all goals in (iii) is $O(|X| * |s_{max}| * |\mathcal{T}|)$. From Proposition 3.3(1) and since $\Sigma_{i=1}^{l} |s \cap \mathcal{T}_{e_i}| \leq |s_{max}|$, the complexity of all goals in (iv) is $O(|X| * |\mathcal{T}| * |\leq^r| * |s_{max}| * |\mathcal{P} \cup \mathcal{N}|)$. In total, the complexity of the call $wellFormed(X^*)$ is $O(|X| * |\mathcal{T}| * |\leq^r| * |s_{max}| * |\mathcal{P} \cup \mathcal{N}|)$.

Case 2) Let $\overset{*}{\oplus}_X\ (T_i)$ or $\overset{*}{\ominus}_X\ (T_i)$ be a subexpression of e.
To evaluate $wellFormed(X^*)$, all of the following goals have to be evaluated:

(v) $belongsParam(x, X^*)$, where x is a variable.

[24]Note that the worst case complexity of $CTCA_e$ is not affected if all negative literals $\neg A$ in the body of the rules are replaced by the atoms A.

(vi) $subsetFL(s, [F_i^*])$, $\forall s \in X$.

(vii) $\neg basicCTermin(F_i^*, s)$, $\forall s \in X$.

According to the proof of Proposition 3.3(1), we have the following complexity results. The complexity of the goal in (v) is $O(|X|)$. The complexity of all goals in (vi) is $O(|X| * |s_{max}| * |\mathcal{T}_i|)$.

We will now compute the complexity of the goals in (vii). To evaluate $basicCTermin(F_i^*, s)$, for an $s \in X$, the goal $belongsFacet(t, F_i^*)$, where t is a variable, as well as the goals $narrower([t], s)$, $\forall t \in \mathcal{T}_i$, should be evaluated. According to the proof of Proposition 3.3(1), the evaluation of $belongsFacet(t, F_i^*)$, where t is a variable, takes $O(|\mathcal{T}_i|)$ time. Additionally, the evaluation of $narrower([t], s)$, $\forall t \in \mathcal{T}_i$, takes $O(|\mathcal{T}_i| * |\leq^r| * |s|) = O(|\mathcal{T}_i| * |\leq^r| * |s_{max}|)$ time. Thus, the complexity of all goals in (vii) is $O(|X| * |\mathcal{T}_i| * |\leq^r| * |s_{max}|)$. In total, the complexity of the goal $wellFormed(X^*)$ is $O(|X| * |\mathcal{T}| * |\leq^r| * |s_{max}|)$.

To evaluate $wFormedExpr(e^*)$, the goal $wellFormed(X^*)$ is evaluated for all $X \in Params(e)$. Additionally, recall from the proof of Proposition 3.3(1) that $O(|\leq^r|) = O(|\mathcal{T}|^2)$. Therefore, $WFT(e) = O(|\mathcal{T}| * |\leq^r| * |s_{max}| * |\mathcal{P} \cup \mathcal{N}|^2)$= $O(|\mathcal{T}|^3 * |s_{max}| * |\mathcal{P} \cup \mathcal{N}|^2)$.

2) Let $WFT^{tree}(e)$ be the maximum cost for deciding if there is a successful SLDNF-derivation of $CTCA_e^{tree} \cup \{\leftarrow wFormedExpr(e^*)\}$.
We distinguish two cases:
Case 1) Let $e_0 = \oplus_X(e_1, ..., e_l)$ or $e_0 = \ominus_X(e_1, ..., e_l)$ be a subexpression of e.
To evaluate $wellFormed(X^*)$, all of the following goals have to be evaluated:

(i) $belongsParam(x, X^*)$, where x is a variable.

(ii) $\neg subsetFL(s, FL_{e_0})$, $\forall s \in X$, and
$subsetFL(s, FL_{e_i})$, $\forall s \in X,\ \forall i = 1, ..., l$.

(iii) $intersectsFL(s, FL_{e_i}, x)$, $\forall s \in X,\ \forall i = 1, ..., l$, where x is a variable.

(iv) $valid(e_i^*, s \cap \mathcal{T}_{e_i})$, $\forall i = 1, ..., l,\ \forall s \in X$.

The complexity of the goals in (i-iii) is as in the proof of the proof of Proposition 4.2(1). From Proposition 3.3(2) and since $\Sigma_{i=1}^{l} |s \cap \mathcal{T}_{e_i}| \leq |s_{max}|$, the complexity of all goals in (iv) is $O(|X| * |\mathcal{T}| * |s_{max}|^2 * |\mathcal{P} \cup \mathcal{N}|)$. In total, the complexity of the call $wellFormed(X^*)$ is $O(|X| * |\mathcal{T}| * |s_{max}|^2 * |\mathcal{P} \cup \mathcal{N}|)$.

Case 2) Let $\overset{*}{\oplus}_X (T_i)$ or $\overset{*}{\ominus}_X (T_i)$ be a subexpression of e.
To evaluate $wellFormed(X^*)$, all of the following goals have to be evaluated:

(v) $belongsParam(x, X^*)$, where x is a variable.

(vi) $subsetFL(s, [F_i^*])$, $\forall s \in X$.

(vii) $\neg basicCTermin(F_i^*, s)$, $\forall s \in X$.

The complexity of the goals in (v-vi) is as in the proof of Proposition 4.2(1). We will now compute the complexity of the goals in (vii). To evaluate $basicCTermin(F_i^*, s)$, for an $s \in X$, the goal $belongs(t, s)$, where t is a variable, as well as the goals $narrower([t], s)$, $\forall t \in s$, should be evaluated. The evaluation of $belongs(t, s)$, where t is a variable, takes $O(|s|)$ time. Now, as we have shown in the proof of Proposition 3.3(1), the evaluation of $narrower([t], s)$, $\forall t \in s$, takes $O(|\leq^r| * |s|^2)$ time. Thus, the complexity of all goals in (vii) is $O(|X| * |\leq^r| * |s_{max}|^2)$. Recall from the proof of Proposition 3.3(2) that $O(|\leq^r|) = O(|\mathcal{T}|)$. Therefore, in total, the complexity of the goal $wellFormed(X^*)$ is $O(|X| * |\mathcal{T}| * |s_{max}|^2)$.

To evaluate $wFormedExpr(e^*)$, the goal $wellFormed(X^*)$ is evaluated for all $X \in Params(e)$. Therefore, $WFT^{tree}(e) = O(|\mathcal{T}| * |s_{max}|^2 * |\mathcal{P} \cup \mathcal{N}|^2)$. $\diamond$

Appendix B: Complete CTCA Logic Program in Prolog

In this Appendix, we give the complete CTCA logic program of the expression e of our running example, in standard Prolog. Terms starting with a capital letter indicate variables. To execute this program in XSB [30, 28], using the SLG-resolution, the operator `not/1` should be replaced by the operator `tnot/1`, and the ":- auto_table." directive should be added.

```
%% Define LP_e^{fact}

belongsFacet(location,loc).
belongsFacet(heraklion,loc).
belongsFacet(ammoudara,loc).
belongsFacet(hersonissos,loc).

belongsFacet(accommodation,acc).
belongsFacet(rooms,acc).
belongsFacet(bungalows,acc).

belongsFacet(facilities,fac).
belongsFacet(jacuzzi,fac).
belongsFacet(swimmingPool,fac).
belongsFacet(indoor,fac).
belongsFacet(outdoor,fac).

subsumedR(heraklion, location).
subsumedR(ammoudara,location).
subsumedR(hersonissos,location).

subsumedR(furnAppartment,accommodation).
subsumedR(rooms,accommodation).
subsumedR(bungalows,accommodation).

subsumedR(jacuzzi,facilities).
subsumedR(swimmingPool,facilities).
subsumedR(indoor,swimmingPool).
subsumedR(outdoor,swimmingPool).

belongsParam([heraklion,bungalows],n).
belongsParam([hersonissos,rooms,indoor],p).
belongsParam([hersonissos,bungalows,outdoor],p).
belongsParam([ammoudara,bungalows,jacuzzi],p).

%% Define LP'_e

valid(e,[]).
valid(e,S) :- belongsParam(P,p), narrower(P,S).
valid(e,S):- subsetFL(S,[loc,acc]), valid(e0,S).
valid(e,S) :- subsetFL(S,[fac]), valid(fac,S).

valid(e0,[]).
valid(e0,S) :- subsetFL(S,[loc,acc]), not(invalid0(e0,S)),
               intersectsFL(S,[loc],S1), valid(loc,S1),
               intersectsFL(S,[acc],S2), valid(acc,S2).

valid(loc,[]).
valid(loc,S) :- belongsFacet(T,loc), narrower([T],S).

valid(acc,[]).
valid(acc,S) :- belongsFacet(T,acc), narrower([T],S).

valid(fac,[]).
valid(fac,S) :- belongsFacet(T,fac), narrower([T],S).

invalid0(e0,S) :- belongsParam(N,n), narrower(S,N).

%% Define LP^{aux}

belongs(T, [T|L]).
belongs(T, [T1|L]) :- belongs(T,L).

belongsFL(T,FL) :- belongs(F,FL), belongsFacet(T,F).

subsetFL(S,FL) :- not(notsubsetFL(S,FL)).
notsubsetFL(S,FL) :- belongs(T,S), not(belongsFL(T,FL)).

intersectsFL([],FL,[]).
intersectsFL([T|L1],FL,[T|L2]) :- belongsFL(T,FL), intersectsFL(L1,FL,L2).
intersectsFL([T|L1],FL,L2) :- not(belongsFL(T,FL)), intersectsFL(L1,FL,L2).

subsumed(T,T).
subsumed(T,T1) :- subsumedR(T,Tmid),subsumed(Tmid,T1).

narrower(S,S1) :- not(notNarrower(S,S1)).
notNarrower(S,S1) :- belongs(T1,S1), not(narrowerAux(S,T1)).
narrowerAux(S,T1) :- belongs(T,S), subsumed(T,T1).

%% Define the well-formedness rules

wellFormed(p) :- not(notWellFormed(p)).
notWellFormed(p) :- belongsParam(S,p), not(wellFormedAux(e,S)).
wellFormedAux(e,S) :- subsetFL(S,[loc,acc,fac]), not(subsetFL(S,[loc,acc])),
                      not(subsetFL(S,[fac])), intersectsFL(S,[loc,acc],S1),
                      valid(e0,S1), intersectsFL(S,[fac],S2), valid(fac,S2).
```

```
wellFormed(n) :- not(notWellFormed(n)).
notWellFormed(n) :- belongsParam(S,n), not(wellFormedAux(e0,S)).
wellFormedAux(e0,S) :- subsetFL(S,[loc,acc]), not(subsetFL(S,[loc])),
                       not(subsetFL(S,[acc])), intersectsFL(S,[loc],S1),
                       valid(loc,S1), intersectsFL(S,[acc],S2), valid(acc,S2).

wFormedExpr(e) :- wellFormed(p),wellFormed(n).
```