Universität des Saarlandes

FR Computerlinguistik

Well-formed Default Unification in Multiple Inheritance Hierarchies

Diplomarbeit

vorgelegt von Christian Husodo Schulz

Erstgutachter: Prof. Dr. Dr. h.c. mult. Wolfgang Wahlster
Zweitgutachter: Dr. Jan Alexandersson

April 17, 2008

Declaration

I, the undersigned hereby declare that the work contained in this thesis is my own original work, and has not previously in its entirety or in part been submitted at any university for any degree.

Saarbrücken, April 17, 2008

Christian Schulz

Acknowledgements

I am very greatful to prof. Wolfgang Wahlster for allowing me to write my thesis at the DFKI and working at interesting issues.

I would like to express my gratitude to my supervisor doc. Jan Alexandersson for spending much of his time in reading my drafts, for having conversations and especially for having the needed patience to attend this work and keeping me focussed in my research.

Also I have to mention the fructuous exchange with my friends at the DFKI Massimo Romanelli, Jan Schehl.

Contents

1 Introduction **3**
1.1 Outline 10

2 Prerequisites—Theory **11**
2.1 Type Hierarchy 12
2.2 TFS 14
2.3 Well-Formed Unification 18
2.4 Credulous Default Unification 21

3 A Prescription of Well-formed Default Unification **23**
3.1 The Assimilation Process 24
3.1.1 Specialization 25
3.1.2 Generalization 26
3.2 Type Preprocessing 27
3.2.1 Account for Multiple Results during Type Preprocessing 28
3.2.2 Termination of the Search for valid Type Configurations 32
3.2.3 Reformulation of Carpenter's Definition for BPO Hierarchies 36
3.3 The Algorithm 39
3.3.1 Algorithm of the Default Unification 43

4 Prerequisites—Practice **47**
4.1 Type Description Language—$\mathcal{TDL}$ 48
4.1.1 The Structure of $\mathcal{TDL}$ Grammars 49
4.1.2 Open World vs. Closed World 50
4.2 The flop Preprocessor 52
4.2.1 The Functions of flop 52
4.2.2 ShUG 55

4.3 Implementing the Lattice Operations 56
4.3.1 Proper Types 56
4.3.2 Synthetic Types 56
4.3.3 Leaf Types 58
4.3.4 *mlb* Calculation between Proper Types 58
4.3.5 *mlb* Calculation between Proper Types and Leaf Types 61
4.3.6 *mlb* Calculation between Leaf Types 62
4.3.7 *mub* Calculation 63
4.3.8 *mlb* Calculation between Leaf Types Revised 67
4.3.9 Grouping the Hierarchy 69

5 An Implementation of Well-formed Default Unification 71
5.1 Precomputation on Types 72
5.1.1 Delta Iterator 73
5.1.2 Detection of Deltas due to Non-Determinism 75
5.2 AVM related Functions 80
5.2.1 The Design of TFS 80
5.2.2 The Unifier Procedures 82
5.2.3 An Extract of an Example 94

6 Conclusions and Future Work 99
6.1 Summary 99
6.2 Future Work 101

A Import of the Type Hierarchy 105
A.1 Binary file generated by flop 105
A.1.1 Sections 105
A.2 Binary file read in by ShuG 110
A.2.1 undump int 110
A.2.2 undump short 110
A.2.3 undump string 110
A.2.4 undump node 110
A.2.5 undump arc 111
A.2.6 undump bitcode 111
A.2.7 Header section 111
A.2.8 TOC section 111
A.2.9 Symbol-table section 112
A.2.10 Hierarchy section 112

A.2.11 Constraints section 112
A.2.12 Supertypes section 113

B Building the Type Hierarchy **115**

C Commented Output of a Concrete Default Unification **119**
References 177

List of Figures

1.1 The slash operator of the priority union identifies which input structure is default or non-default. In the example taken from (Kaplan, 1987, P. 180), the part of non-default structure A associated to q, overwrites the corresponding counterpart in B. 5

1.2 An excerpt from the SmartKom ontology showing the more general *entertainment* frame and the two specialized frames *performance* and *broadcast*. 8

2.1 A prototypical hierarchy featuring maximal upper bounds and minimal lower bounds. 13

2.2 Within the dotted circles we have the type definitions together with their appropriate maximal extensions. We say that all structures that are subsumed by the type definitions are well-formed. 16

2.3 A type hierarchy sample that motivates the well-formed unification in (Copestake,1992). 19

3.1 Multiple results due to two abstract background types that have no subsumption relation. The grey cloud indicates an area within all possible types except *top* and *bottom* may occur in a partial order, where the subsumption relation between the types below and above the cloud still holds. 29

3.2 Multiple results due to two different abstract cover types, that are not under subsumption relation. 30

3.3 The number of results corresponds to the cartesian product of the number of "different" abstract backgrounds and abstract covers. 31

3.4 The break condition during the search for abstract background types in BCPO hierarchy holds if the supertype of the background type is not subsumed by the least upper bound between the background type and the cover type. In this case the type of the cover is at the same time the abstract cover type. 32
3.5 On the left the abstract background type t_1 is subsumed by the *mub*, $t_1 \in \sqcap_t(t_{co}, t_{bg})$, while on the right it is not subsumed by a *mub* of cover type and background type. 33
3.6 Is t_1 a valid abstract background type, though it subsumes the *mub* t_3 between t_{co} and t_{bg} ? 35
3.7 This sample hierarchy features type configurations that produce also results during default unification that are not most specific. The observation reveals that the members of the type configurations $\langle t_0, t_{co} \rangle$, $\langle t_2, t_3 \rangle$ and $\langle t_6, t_7 \rangle$ are related in parallel by the subsumption ordering. We argue that given the latter type configuration, the two other options are not relevant anymore. 37

4.1 With $\mathcal{TDL}$ as starting point the system passes the type hierarchy in an intermediate format generated by the flop preprocessor to a wrapper class called ShUG ("Shallow Unification Grammars", see 4.2.2) that stores the type hierarchy in a java object. 47
4.2 A sample of type definitions in $\mathcal{TDL}$ 50
4.3 The open world case to the left incorporates $\mathcal{TDL}$ to integrate a newly generated type definition into the type hierarchy if we have no explicit join of the *avm types* a and b. The same case would produce the *bottom* type as output when accepting the closed world assumption.The grey area represents the domain of the type definitions in our type hierarchy. 51
4.4 PET—System Overview and Experimental Setup 53
4.5 The emergence of synthetic types 57
4.6 Handling multiple results in *mlb* processing 60
4.7 Finding *mlb* between a leaf type and a proper type 61
4.8 Finding *mlb* between a leaf type and a leaf type 62
4.9 Failure of the original encoding when calculating *mub*. Note that missing bit-positions are equal to 0. 64

4.10 Type encoding for *mub* processing. 68

4.11 The encoding into groups 69

5.1 Given the Delta queue is empty the expansion to the next higher layers in the hierarchy happens as long expansion is possible, see line 7, 8 and none among background supertypes has a *mlb* with the cover type, see line 15. 73

5.2 Sorting out already visited background candidates. The blue lines show the path $\langle t_{bg}/t_5/t_4/t_3\rangle$ and the red lines display the movement on the path $\langle t_{bg}/t_5/t_7/t_6/t_3\rangle$. 74

5.3 On the right side the synthetic type is placed so that we have a unique lower bound between t_2 and the type of the cover. The red arcs substitute the original direct subsumption relation represented by the green arc between t_{co}, t_3 and between t_2 and t_4 (Option 1). On the left side the synthetic type is placed so that we have a unique lower bound between t_0 and t_1. The red arcs substitute the original direct subsumption relation represented by the green arc between t_1 and t_4 (Option 2). . . 77

5.4 The expansion of layers depends on the fact if the supertype is still on the path to the *mub*, see line 8. If it is the case that the supertype yields a Delta, the background supertype gets removed in line 20. Though before discarding a path it is crucial to check if the specialization succeeds. Other Deltas are identified by the iteration over the synthetic types in 24-30. 78

5.5 Equivalence relation between an ambiguous TFS and the resolution to all its interpretations. 80

5.6 A TFS Object consists of a list containing Feature Value Pair Objects and a list containing Feature Value List Objects . . . 81

5.7 The incremental construction of the resulting TFS during the default unification process. 90

5.8 Overview of the interactions of the single parts in the implementation. After the cover is passed to the default unifier, the delta iterator gets instantiated. The query of the next Delta follows the specialization that decides if the processing is delayed by the query of the next Delta or if it continues with the generalization and at last the overlay operation. The latter component comprises the recursive application of the entire procedure to the internal structure of the assimilated structures. 93
5.9 This image represents the effect of the first Delta request, entailing the expansion to all supertypes in the first layer atop the background type. The hierarchy refers to a fragment of the implemented type hierarchy in the figure B.1. The green lines indicate the navigated route computed by the Delta iterator. The red lines indicate that the unification between the TFS of the cover argument and the prototype structure of the abstract cover type fails, whereas the blue line says that unification succeeds. 94
5.10 The second request of a Delta effectuate the expansion to all supertypes in the second layer atop the background type. . . . 96

B.1 Type Hierarchy A 116
B.2 Type Hierarchy B 117
B.3 Type Hierarchy C 118

Chapter 1

Introduction

Commonsense reasoning embraces the concept to draw conclusions on the basis of partial information and also inconsistent information that we retract in order to obtain a more complete information. It is a shared knowledge that this type of logic by reasoning with defaults is *non-monotonic* (Moore, 1995; Wahllöf, 1996).

Since the late 80's non-monotonic inference has been a prominent theoretical issue (Daelemans, Smedt, & Gazdar, 1992). Especially in the area of computer linguistics the research of default reasoning has a great impact. In parallel, analogous to considerations in software engineering to favor the use of object-orientation in computer science, it becomes desirable to embed linguistic theories in inheritance networks. Linguistic theories such as Systemic Functional Grammar (Halliday & Matthiessen, 2004), Word Grammar (Fraser & Hudson, 1992), or HPSG (Pollard & Sag, 1994) make use of inheritance to describe linguistic structures at the lexical, morphological, syntactic, or semantic (conceptual) levels. A wide spread opinion is that single inheritance network (*orthogonal inheritance*) are not suitable to the description of natural languages. For instance in the context of lexical maintenance, morphological and syntactic properties are frequently disjoint, i. e., the subcategorization class of a verb is not predictable from its conjugation class, and vice versa. *Multiple inheritance* permits the two types of information to be separated by isolating them in distinct subhierarchies. More general preferences concerning the properties of multiple inheritance hierarchies are basically similar to the benefits of object-oriented programming language, including (1) parsimony—inheritance lexicons are smaller in comparison to their full-entry counterparts; (2) maintenance comfort—instead of changing

or correcting numerous individual entries, modifications to a couple of nodes may be sufficient; (3) uniformity—it is possible to refer the same allocation of inference to different levels of linguistic description; (4) modularity—multiple inheritance permits distinct to apply for distinct levels of linguistic description; (5) interaction—it is possible to express the fact if a lexical property at one level of description, e. g., syntactic gender depends on a lexical property at another level of description, e. g., the phonology of a word-final vowel.

The linking of multiple inheritance with default reasoning represents one of the key issues in the knowledge representation literature, where strategies of how to deal with the inheritance of mutually contradictory information from two or more parent nodes have been offered. Subsequently it has made possible a range of applications that can be attributed to two distinct motivations.

One major topic is the attempt to embrace linguistically significant generalizations in an inheritance hierarchy, where default reasoning acts as an extension to the common unification-based hierarchies. For throughout all linguistical disciplines a series of restricted default inheritance languages have been designed. Among the most early striking solutions to encapsulate non-monotonic behaviour in inheritances is the specification in syntactic theory of GPSG (Gazdar, 1987). For the purpose to express exceptional behaviour towards more specific concepts in the hierarchy, GPSG requires to make heavy use of defaults.

Thereinafter (Shieber, 1986) proposes an operation *add*, which adds information of a feature structure A to a feature structure B, as far as this information is not in conflict with information in B.

Bouma (Bouma, 1990, 1992) provides a definition of *default unification*[1] as an operation similar to ordinary unification with the difference that non-default parts of the structure take precedence in case of unification failure.

While the application of Bouma's language *FML* is associated to the syntactical level of linguistic description, ELU in (Russell, Carroll, & Warwick, 1991; Russell, Ballim, Carroll, & Warwick-Armstrong, 1992) represents an underlying formalism dealing with the morphological aspect. A goal in lexicon maintenance is having the capability to make general statements about classes of words and also to adhere exceptions to such statements affecting individual words and subclasses. ELU employs the multiple default inheri-

[1] The term default unification is commonly used and refers to the operation deployed to most of the works treating with default reasoning.

tance mechanism to describe regularity, subregularity and exceptions when classifying the properties of words.

The concern in this work aligns with the second motivation of default reasoning consisting in the use of a variety of general non-monotonic logics for formalizing pragmatic components of NLP and dialog systems.

A uniformity to all approaches in non-monotonic logics is established since (Kaplan, 1987) has suggested to consider the operation qualified to combine default with the non-default information as an asymmetric operation that is not commutative and not transitive. In literature, numerous variants of terms for the purpose to indicate the distinction of what is default and non-default come across.[2] Kaplan's notion of *priority union* provides the technical background to his analysis of gapping constructions.

$$A = \begin{bmatrix} q & r \\ s & t \\ u & v \end{bmatrix} B = \begin{bmatrix} q & m \\ s & t \\ p & l \end{bmatrix} A/B = \begin{bmatrix} q & r \\ s & t \\ u & v \\ p & l \end{bmatrix}$$

Figure 1.1: The slash operator of the priority union identifies which input structure is default or non-default. In the example taken from (Kaplan, 1987, P. 180), the part of non-default structure A associated to q, overwrites the corresponding counterpart in B.

Carpenter (Carpenter, 1993) introduces a new approach to the notion of default unification. He defines two default unification operations called *credulous* and *skeptical* default unification. The idea behind the credulous operation is to generalize the default structure until it unifies with the strict structure. Unlike the previous strategies to replace conflicting parts by the strict information, he suggests in case of clashes to relax the specificity on the defeasible part to achieve consistency with the strict structure. Carpenter states that the definition of the generalized default might be not deterministic, which entails that the credulous variant allows having multiple results. On the other hand the skeptical version would finally collapse diverse results

[2]Defaults are alternatively described by synonyms as *defeasible*, *background*, *old*, *source* and non-default as *strict*, *cover*, *new*, *target*. Within the principal part of this work we stick to the terms strict, cover on one side and defeasible, background on the other side.

into a unique one. Carpenter implicitly assumes an existing inheritance hierarchy that organizes the subsumption relation between the structures. Strict and defeasible information is described by feature structures that are partially ordered according to a subsumption ordering that can be interpreted as an ordering on the amount of conveyed information. Accordingly, Carpenter's definition of default unification heavily motivates the use of a typed feature structure language, henceforth TFS, that is a synthesis of several key concepts stemming from unification-based grammar formalisms, knowledge representation languages and logic programming. It offers declarative framework, with all the advantages of logical formalisms: expressive power, simplicity, and sound formal semantics.

The first detailed description of an algorithm realizing Carpenter's definition with the use of TFS has been provided by (Grover, Brew, Manandhar, & Moens, 1994). Its implementation in ALE (Carpenter & Penn, 1998) uses the specification of priority union in (Grover et al., 1994) to resolve parallelism-dependent anaphora, i. e., verb phrase ellipsis. The resolution mechanism is based on the linguistic discourse model of (Prüst, Scha, & Berg, 1994). The term *most specific common denominator* (MSCD) represents the test for parallelism between two *discourse unit constituents* (DCU). Moreover it refers in the scope of Carpenter's definition to the alleviated version of the default structure that is consistent with the strict information. The actual ellipsis resolution is executed separately, where the structure of MSCD is combined with the target.

The underlying formalism of Grover's priority union suffers from drawbacks regarding both the computational and the theoretical point of view. The algorithm in (Grover et al., 1994) decomposes the source into atomic feature structures (Moshier, 1988) and unifies consecutively each atomic feature structure with the strict structure until the unification fails. The clash during unification depends on which order the atomic feature structures are unified with the strict structure. Thus unification is executed $n!$ times—where n corresponds to the number of atomic structures in the defeasible part of the argument—between background's atomic feature structure and the cover to receive all possible results. Finally, since the algorithm may produce redundant results, we must sort out the most informative feature structures.

The theoretical concern attests that atomic feature structures do not belong to any type or type description. It is questionable to allow the arbitrary

unification of atomic feature structures out of the source with the target, since it is unclear which meaning the set of atomic feature structures has that is unifiable with strict structure. Grover also challenges the fact whether MSCD sufficiently incorporates a *characteristic generalization* of the defeasible source and states that their formalism lacks giving a formal notion of the common ground that is supposed to establish parallelism .

An algorithm linear to the size of the structures involved is introduced by *lenient default unification* in (Ninomiya, Miyao, & Tsujii, 2002). In contrast to Carpenter's credulous default unification, the goal of the algorithm in case of clashes is to maximize the information content of the resulting structure by "pushing" the affected values of the default structure towards the leafs. Though their approach somehow omits the requirement to have well-typed result, since the effect of introducing types to feature structures possibly produces feature structure that are not well-typed.

Whereas in (Lascarides & Copestake, 1999) the realization of an order independent persistent default unification is based on the agreement "*Only parts of the feature structure which are fully type compatible with the non-default structure are split*" as stated in (Copestake, 2002). An interesting novelty based on the marking of default information as it has been previously presented by (Young & Rounds, 1993) is that default unification is described as a binary, order-independent (i.e., commutative and associative) operation. However some of other postulated properties for default unification conflict with Carpenter's definition and further are not applicable to tasks of the kind as in (Grover et al., 1994). According to their formalism, default unification may fail if strict information itself already incorporates inconsistency. Carpenter does not take into account the potential that strict input information contains already conflicts and so default unification always succeeds. Further property of the order independent persistent default unification is that it returns always a single result, which is also not an indispensable desideratum according to Carpenter, see credulous default unification.

The *overlay* operation in (Alexandersson & Becker, 2007, 2004; Alexandersson, Becker, & Pfleger, 2004; Alexandersson & Becker, 2003) as (Grover et al., 1994) contains a precise formalization of Carpenter's credulous default unification. Their approach to heavily utilize type hierarchies during default unification brings together favorable computational properties and also declarative semantics attributed to the resulting structures. Further, the analysis is realized in an appealing constraint-based architecture for an object-oriented language based on multiple inheritance, TFSs, and unifica-

tion. A slim version of the overlay specification considering single inheritance hierarchies has been implemented in the SmartKom system (Wahlster, 2003; Wahlster & Wahlster, 2006), a multi-modal dialog system with a frame-based meaning representation. SmartKom models the user's intention in a dialog system where conclusions referring to the last user's utterance are based on previous knowledge collected during a dialog. The defeasible part is related to the last instance of the knowledge frame that might be overwritten by new information. Within a new event during the dialog, the locutor possibly introduces additional information to the context that conflicts with the given context, i. e., change of topic, previous information is revised, some parts in the context are no longer valid thus redundant. To deal with the latter case, parts of the discourse context are to be inherited such that interpretation of the actual instance in the dialog is complete and consistent with the new information.

A simplified excerpt of the ontology describing the frames discussed above is shown in figure 1.2. Consider the following example of a dialog[3] between

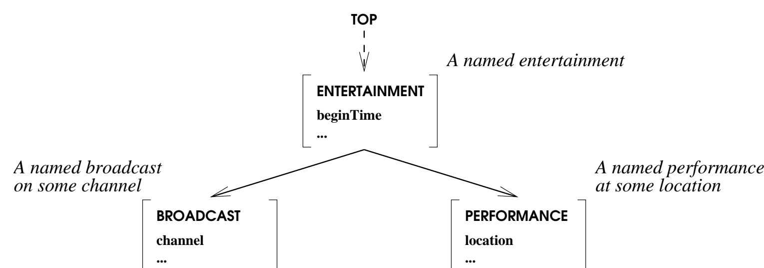

Figure 1.2: An excerpt from the SmartKom ontology showing the more general *entertainment* frame and the two specialized frames *performance* and *broadcast*.

a user who is seeking information about the movie and TV program and the SmartKom system:

(1) **User**: I'd like to go to the movies tonight.

(2) **SmartKom**: Here (↗) are the films showing in Heidelberg.

[3]The only multi-modal aspect of this dialog is the presentation of the actual film titles on the display and a pointing gesture by the presentation agent, marked by ↗.

(3) **User**: No, there is nothing interesting there,

(4) **User**: what is showing on TV?

After the system's reply (2), the context contains information about the topic *going to the movies* and some details like *in Heidelberg* and *tonight*. Note, that in the SmartKom system, certain defaults, e. g., the location Heidelberg in this example, are incorporated into the representation of the user utterances. The next user utterance (3) could be seen as meta-talk, signaling a change of topic which is then made explicit in the final utterance (4). Since the two topics *movies* and *TV* are clearly related, some of the information from the old context should be retained, i. e., *in Heidelberg* and *tonight*.

Thus the default unification formalization in (Alexandersson & Becker, 2007) is used in the SmartKom system as a general purpose mechanism for enriching the interpretation of some communicative action with consistent information from some defeasible structure. In addition they provide a *scoring function* that is a precise specification of how a distinction based on informational content with respect to the default and strict structure among the multiple results can be realized.

This work performs an extension to the default unification presented in (Alexandersson & Becker, 2007) concerning the theoretical and the practical aspect without taking into account an evaluation-based mechanism like the scoring function. Firstly, the algorithm of the default unification is adapted to multiple inheritance type hierarchies omitting the requirement to have a unique greatest lower bound. This implies to deal with a finite poset permitting two or more types to feature joins as a set of disjunctive and disjoint types. Consequently, appropriate observations are needed to light up the great impact this property has on the default unification formalism in (Alexandersson & Becker, 2007). Moreover the constraint-based architecture of the type feature language has been enhanced approving the well-formedness condition on TFSs discussed by (Copestake, 1992). Analogously we provide an adequate and efficient implementation of the default unification operation featuring the properties introduced in the theoretical analysis.

1.1 Outline

Chapter 2 : Prerequisites - Theory

The next chapter serves with necessary theoretical foundations, containing essential formal specifications of the type hierarchy, type feature structure and the unification operation on TFSs.

Chapter 3 : A Prescription of Well-formed Default Unification

The main contribution in the thesis is to give finalized version to the already detailed specification of the algorithm respecting Carpenter's credulous default unification, based on extensive study of all sorts of eventualities encountered in multiple inheritance *bounded* hierarchies that are not *complete*.

Chapter 4 : Prerequisites - Practice

A suitable type hierarchy for the purpose of experimenting the default unification procedure is given by $\mathcal{TDL}$ (Krieger, 1995) that offers sufficient capability to provide the basis of this work. However essential adjustments for the purpose of implementing the default unifier were necessary concerning the open world type system of $\mathcal{TDL}$ (Gerdemann, 1995a). Those are in particular modifications applied to the type hierarchy by flop preprocessor to obtain the *BCPO* condition.

Chapter 5 : An Implementation of Well-formed Default Unification

This chapter describes the impact that the framework of $\mathcal{TDL}$and flop has on the implementation. Eventually it comes up with additional proposals to optimize the default unification procedure. In the following commented pseudo codes corresponding not only to the modification but also to entirely implemented modules illustrate the course of action in the described procedure respectively. Finally we give an excerpt of a more complex example with a detailed demonstration of the single steps during the default unification procedure.

Chapter 6 : Conclusions and Future Work

At last an overview of the main contributions and some anticipations of future works are presented.

Chapter 2

Prerequisites—Theory

This chapter will give the required knowledge and definitions in order to understand and to be able to give a precise specification of the default unification procedure.

As in (Alexandersson & Becker, 2007) the specification of the default unification algorithm is to be interpreted as an operationalization of the formal characterization of the credulous default unification in (Carpenter, 1993). Therefore many definitions are related or are taken from the latter work introducing the concepts of types, feature structures, henceforth FS, and lattices.

A crucial difference to the common analysis of lattice theory is that we allow multiple inheritance, where different types may inherit from an equal set of types. Such a property gives on one side more expressiveness to some hierarchy but implies also on the other side complex impacts on the required operations referred to the hierarchy.

Further we agree that a TFS that is included in the set of the maximal extension of a type definition is well-formed. Moreover we consider it to be an essential issue to maintain the well-formed condition on the structures, if we want to apply the required operations for the purpose of the default unification calculation. In particular we relate the unification on TFSs to the specification of well-formed unification presented in (Copestake, 1992). Finally this chapter seclude with Carpenter's essential definition of the credulous default unification.

2.1 Type Hierarchy

First of all, we have to specify our inheritance hierarchy as a type hierarchy. For this purpose, we need a set Type of types ordered according to their specificity, i. e., given $t, t' \in \mathsf{Type}$, if t' inherits information from t, then t' is said to be more specific. At this point, we also say that t *subsumes* t', and write it $t \sqsubseteq t'$.[1]

Definition 1 Partial Order

A relation $\sqsubseteq$ on a set of types Type is a *partial order* in case it is:

- *reflexive*, i. e., $\forall t_1 \in \mathsf{Type}$ we have $t_1 \sqsubseteq t_1$
- *antisymmetric*, i. e., $\forall t_1, t_2 \in \mathsf{Type}$ if $t_2 \sqsubseteq t_2 \wedge t_2 \sqsubseteq t_1$ then $t_1 = t_2$
- *transitive*, i. e., $\forall t_1, t_2, t_3 \in \mathsf{Type}$ if $t_1 \sqsubseteq t_2 \wedge t_2 \sqsubseteq t_3$ then $t_1 \sqsubseteq t_3$

□

The *bounded complete partial order* condition (BCPO, (Carpenter, 1992)) on lattices assumed in the definition of the inheritance hierarchy given in (Alexandersson & Becker, 2007) says that for any subset T of Type there is a *greatest lower bound* and a *least upper bound*.[2] Saying that for each $t, t' \in \mathsf{Type}$ there is a unique upper bound t_u with $t_u \geq t, t_u \geq t'$ and also unique lower bound t_l with $t_l \leq t, t_l \leq t'$. In our analysis the strict bounded completeness conditions on the inheritance hierarchy has been relaxed to allow the existence of *minimal upper bounds* and *maximal lower bounds*[3] (B. Davey, 1990).

Definition 2 Maximal/Minimal Lower/Upper Bounds

Given $T \subset \mathsf{Type}$ and $t_a, t_b \in \mathsf{Type}$,

- if $t_a \sqcup_t t_b = T$ and T is the set of maximal lower bounds, then for each $t' \in T$, we have

[1] In terms of a types as sets denotational semantics, increasing information content decreases the denotation, i. e., if $t \sqsubseteq t'$ then $[\![\text{t}]\!] \supseteq [\![\text{t'}]\!]$.

[2] We refer later on to *glb* and *lub* respectively.

[3] We refer later on to *mub* and *mlb* respectively.

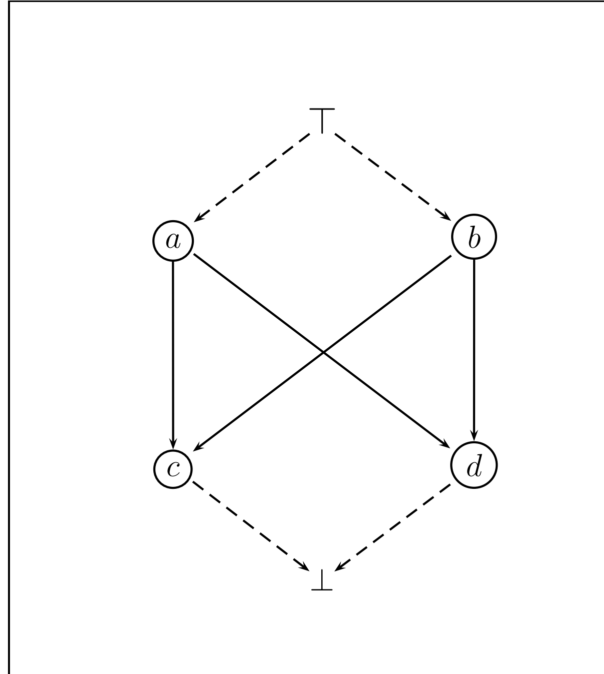

Figure 2.1: A prototypical hierarchy featuring maximal upper bounds and minimal lower bounds.

  - $t_a \sqsubseteq t',\ t_b \sqsubseteq t'$
  - If $t' \sqsubseteq t''$ or $t'' \sqsubseteq t'$, and $t'' \in T$ then $t' = t''$

- if $t_a \sqcap_t t_b = T$ and T is the set of minimal upper bounds, then for each $t' \in T$, we have

  - $t_a \sqsupseteq t',\ t_b \sqsupseteq t'$
  - If $t' \sqsubseteq t''$ or $t'' \sqsubseteq t'$, and $t'' \in T$ then $t' = t''$

□

The example in the figure 2.1 is a partial order $\langle \mathsf{Type}, \sqsubseteq \rangle$, where the join $\sqcup(a, b)$ would have to be defined non-deterministically to generate c and d as possible results. Dually we have as meet $\sqcap(c, d)$ the results a and b. In this work we assume to consider the partial orders to be bounded, that is all hierarchies have a greatest element *top* and a least element *bottom*. We

suggest here that more specific types are placed towards the *bottom* and dually more general types towards the *top*.

Such hierarchies have been, for instance, installed to SFP domains (Plotkin, 1976) or to partial orders (Vickers, 1989). Further similar concepts of non-determinism arise in the context of universal unification (Siekmann, 1984) and some notions of set-valued FSs (Pollard & Moshier, 1990).

2.2 TFS

The definition of the TFS is based on the concept of labelled finite-state automaton (Carpenter, 1993) with arcs connecting nodes to features, with the difference that nodes are exclusively labelled by types. A node is labelled to either a *complex* structure, that is a type with outgoing arcs or to a *atomic* structure a type without any features[4].

Definition 3 TFS

A TFS is a tuple $F = \langle Q, \overline{q}, \theta, \delta \rangle$ where:

- Q is a finite set of nodes with the root $\overline{q}$
- $\overline{q}$ is the unique root node
- $\theta : Q \rightarrow \mathsf{Type}$ is a total node typing function
- $\delta : \mathsf{Feat} \times Q \rightarrow Q$ is a partial feature value function

□

In order to map the monotonicity condition on the type hierarchy to the TFS, we must establish a definition that associates features with types. Given a set of features Feat and a set of types Type:

Definition 4 Appropriateness

Let $\langle \mathsf{Type}, \sqsubseteq \rangle$ be an inheritance hierarchy and $f \in \mathsf{Feat}$. Then we define a partial function $\mathsf{Approp} : \mathsf{Feat} \times \mathsf{Type} \rightarrow \mathsf{Type}$ where

[4] Structures performing such a property are also called *atomic type*.

Feature introduction For every feature $f \in \mathsf{Feat}$ there is a most general type $\mathsf{Intro}(f) \in \mathsf{Type}$ such that $\mathsf{Approp}(f, \mathsf{Intro}(f))$ is defined

Downward closure/Right monotonicity If $\mathsf{Approp}(f, \tau)$ is defined and $\tau \sqsubseteq \sigma$, then $\mathsf{Approp}(f, \sigma)$ is also defined and $\mathsf{Approp}(f, \tau) \sqsubseteq \mathsf{Approp}(f, \sigma)$

□

Translating the definition informally, the first condition says that for each feature f there is a type $t \in \mathsf{Type}$, where f is introduced and it does not exist some other type $t' \in \mathsf{Type}$, such that f occurs in t' and $t' \sqsubseteq t$. The second condition says that for a feature f, if it is the case that f is defined for a type t, then f will be propagated to all subtypes of t and for any t' and $t \sqsubseteq t'$ we have $\mathsf{Approp}(f, t) \sqsubseteq \mathsf{Approp}(f, t')$.

There are some versions of TFS languages (also including (Carpenter, 1992); (Gerdemann & King, 1994); (Krieger, 1994a); (Smolka, 1992), (Emele & Zajac, 1990)) that does not take into account the *constraint specification* given by (Copestake, 1992). Adopting the notion of constraint specification as a template description (Shieber, 1986), we can construct powerful typed inheritance hierarchies while defining only a fraction of all possible extended TFSs (Copestake, 1996). Inheritance hierarchies in this way are relevant from a computational viewpoint and as well as theoretical one. We give now a definition of the *most general satisfier*, that is adapted from the concept of the constraint specification in (Copestake, 1992).

Definition 5 Most General Satisfier

Given an inheritance hierarchy $\langle \mathsf{Type}, \sqsubseteq \rangle$, a type $t \in \mathsf{Type}$ and a set of TFS, FS and for each $F' \in FS$, F' meets the appropriateness condition we have a partial function $MGsat :\subseteq \mathsf{Type} \rightarrow FS$ such that
$MGsat(t) \rightarrow F = \langle Q, \overline{q}, \theta, \delta \rangle$, where:

1. $\theta(\overline{q}) = t$.
2. For all $F' = \langle Q', \overline{q}', \theta', \delta' \rangle$ and $\theta(\overline{q'}) = t$, then $F \sqsubseteq F'$.
3. For all $q \in Q$ we have the condition $\theta(\overline{q} \not\sqsupseteq \theta(q)$.)

□

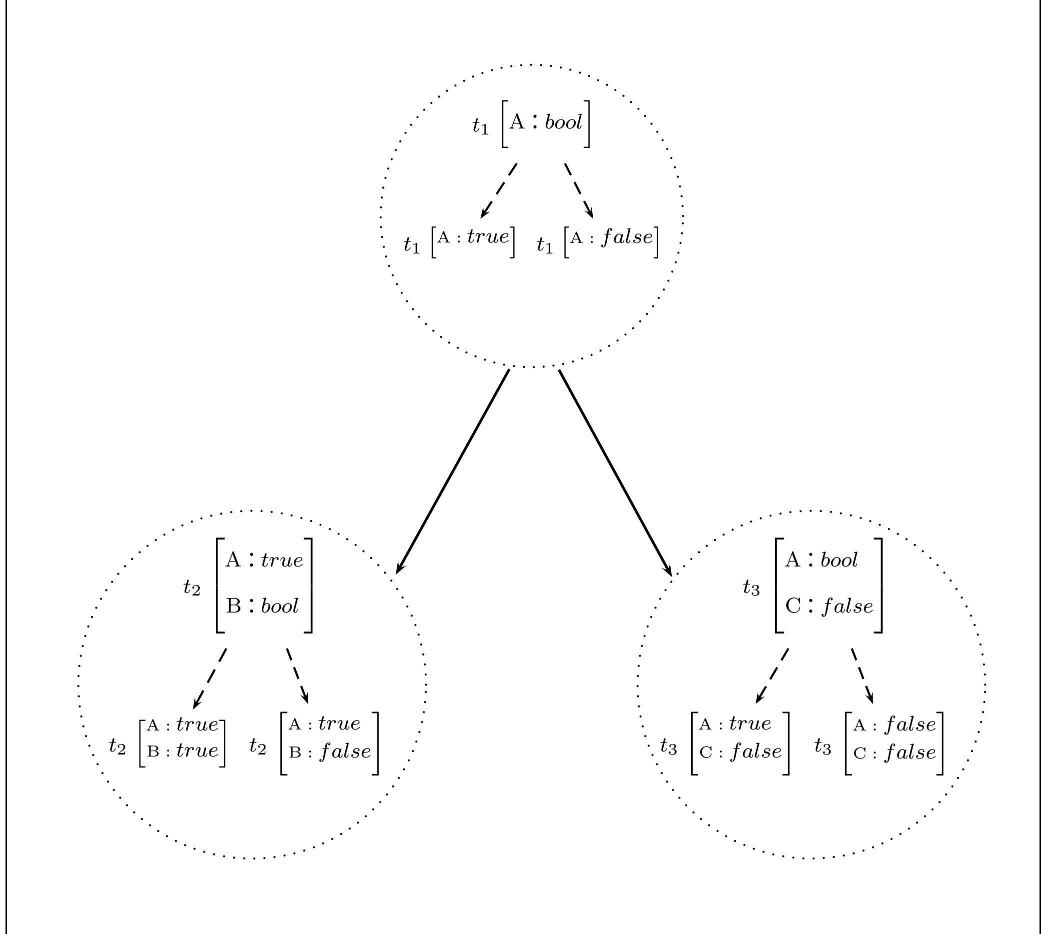

Figure 2.2: Within the dotted circles we have the type definitions together with their appropriate maximal extensions. We say that all structures that are subsumed by the type definitions are well-formed.

The definition of the most general satisfier implies also the notion of *well-formedness* of a TFS given an inheritance hierarchy.

Definition 6 Well-Formed TFS

Some TFS $F' = \langle Q', \overline{q'}, \theta', \delta' \rangle$ is well-formed if $MGsat(\theta'(\overline{q'})) \sqsubseteq F'$. $\square$

Figure 2.2 illustrates an example of a type hierarchy, that would correspond to the definition of the language of TFS we use here. Subsequently we will switch between the terms most general satisfier[5], *the prototype structure* of a type t and *type definitions* in general.

We further adopt the notion, that two type definitions are not equal, if they can be extended to distinct collections of maximal FSs (Carpenter, 1992). Thus this implicates that we accept the assumption in (Alexandersson & Becker, 2007), that for each type $t \in \mathsf{Type}$ there is a possibly empty set of features introduced by type t. Copestake in contrast suggests for their TFS language, that given a set $\mathsf{Feat}' \subset \mathsf{Feat}$ where $Appfeat(t) = \mathsf{Feat}'$ and further $Appfeat(t') = \mathsf{Feat}'$ it follows that $t = t'$. Appfeat is a partial function taken from (Copestake, 1992) that is analogous to the Approp, but instead Appfeat assigns a type to an appropriate set of features.

This is a relevant change to (Alexandersson & Becker, 2007), where the notion of MGSat does not explicitly satisfy the well-formedness condition above. For instance the figure in 2.2 satisfies the appropriateness condition. If we leave out the well-formedness condition it would be feasible to replace the value *true* at feature A with *bool* for the type definition t_2, while still maintaining the appropriateness condition. So in addition to the type definition t_2 in the figure 2.2 the modified version of the TFS would be also qualified to be a MGSat for t_2 in the hierarchy. In principle when omitting the well-formed condition we can have many different MGSats that all satisfy the appropriateness condition. Hence a hierarchy that respects the well-formedness condition is one concrete hierarchy among many hierarchies following the appropriateness condition. We will see that it is crucial to have a concrete specification of the type definitions when integrating them into the default unification processing.

The last condition defining MGSat inhibits the generation of TFSs containing cycles, that is referred to the *compatability condition* in (Copestake, 1992). Strictly speaking the compatability condition entails the prevention to assign the *top* type to a feature, regardless what type definition we want to determine. Though, if we look at the illustration in the figure 2.3 adopted from (Copestake, 1992), we note that the compatability condition is violated. The design of the type hierarchy viewable in the appendix B, that is integrated into the built up system does not follow the latter constraint as well. The treatment of cyclic structures during the processing is not re-

[5]Henceforth we use sporadically also the abbreviation MGSat.

quired. Cycles would only be generated if we extended type definitions to all TFSs satisfying the type definition. For the purpose of experimenting with the default unification processing in a realistic environment, it is sufficient given the type definitions to check the well-formedness of the conveyed arguments. So in practice, instead of considering the compatability condition in the broadest sense, it is satisfactory to shift the task of excluding cyclic structures to the construction of the type hierarchy, where the type definition itself may not contain cycles.

2.3 Well-Formed Unification

Given the TFS language we want to specify the *unification* between two TFSs, that is an essential operation during the default unification process. Therefore a definition of the *subsumption relation* between two type feature structures, that is derived from the very elegant definition in (Moshier, 1988), we give first below:

Definition 7 Subsumption

$F = \langle Q, \overline{q}, \theta, \delta \rangle$ is said to subsume $F' = \langle Q', \overline{q}', \theta', \delta' \rangle$, $F' \sqsubseteq F$, iff there is a total function $h : Q \rightarrow Q'$ such that:

- $h(\overline{q}) = \overline{q}'$
- $\theta(q) \sqsupseteq \theta'(h(q))$ for every $q \in Q$
- $h(\delta(f, q)) = \delta'(f, h(q))$ for every $q \in Q$ and feature f such that $\delta(f, q)$ is defined

□

The definition says that a TFS F subsumes another TFS F' if and only if all features in F are also present in F' and the type assigned by F to a feature subsumes the type assigned by F' to the same feature in the type ordering. We can now define unification as a binary operation over two type feature structures bringing together their informational content.

Definition 8 Unification

The unification $F \sqcup F'$ of two TFSs F and F' is taken to be maximal lower bounds of F and F' in the collection of well-formed TFSs ordered by subsumption. □

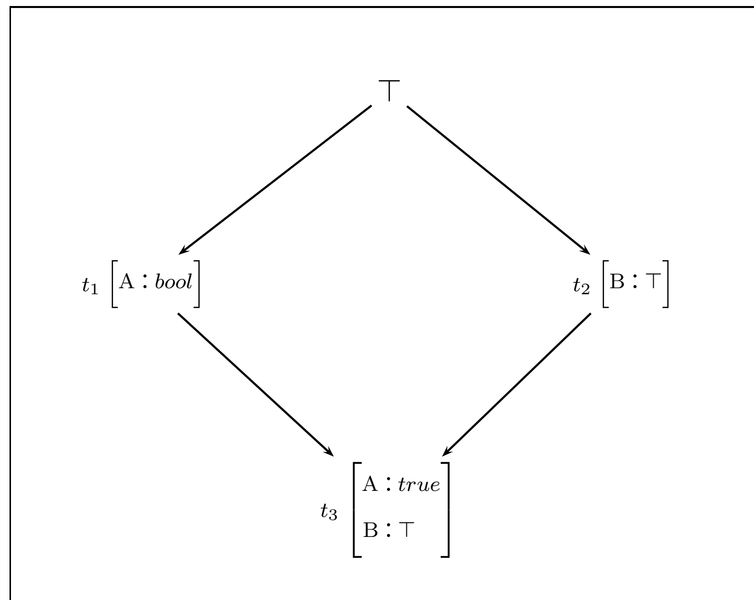

Figure 2.3: A type hierarchy sample that motivates the well-formed unification in (Copestake,1992).

Even if in the introduced definition of the unification we assume implicitly as maximal lower bounds have to be well-formed TFSs, we must explicitely extend the unification from a binary to a ternary operation. For the unification process in order to yield a well-formed result it is required to unify additionally with the type's MGSat of the resulting structure. An illustration based on the hierarchy in figure 2.3 exemplifies the inconsistency, that may arise when leaving out the additional step. Note that we still consider the unification operation in a non-deterministic environment such as in figure 2.1. Therefore the resulting structure of a unification is supposed to be a set of well-formed TFSs. Thus two TFSs are unifiable if their unification does not result in a set containing one single element that is equal with *bottom*.

Given the type hierarchy in the figure 2.3 we have $t_3 = t_1 \sqcup_t t_2$ and

$$MGsat(t_1) \sqcup MGsat(t_2) = \{ \; {}_{t_3}\begin{bmatrix} \text{A} : \textit{bool} \\ \text{B} : \top \end{bmatrix} \; \}, \text{ but } MGsat(t_3) \not\sqsubseteq \; {}_{t_3}\begin{bmatrix} \text{A} : \textit{bool} \\ \text{B} : \top \end{bmatrix}$$

Now consider the well-formed TFSs F_1, F_2, F_3

$$F_1 = \; {}_{t_2}\begin{bmatrix} \text{B} : \top \end{bmatrix} \qquad F_2 = \; {}_{t_1}\begin{bmatrix} \text{A} : \textit{bool} \end{bmatrix} \qquad F_3 = \; {}_{t_1}\begin{bmatrix} \text{A} : \textit{false} \end{bmatrix}$$

with the following unifications:

$$1.) \; F_1 \sqcup F_2 = \{ \; {}_{t_3}\begin{bmatrix} \text{A} : \textit{bool} \\ \text{B} : \top \end{bmatrix} \; \} \qquad 2.) \; F_1 \sqcup F_3 = \{ \; {}_{t_3}\begin{bmatrix} \text{A} : \textit{false} \\ \text{B} : \top \end{bmatrix} \; \}$$

If we omit to unify with the appropriate MGSat we run the risk as in the first case of accepting a TFS that does not satisfy the type definition, though it is in subsumption relation with the MGSat. In the second case the unification would result in the production of a TFS, that is even not consistent with the type definition.

In (Copestake, 1992), the unification procedure comprising the generation of only well-formed results is referred to as *well-formed unification*. Equally we also state the definition of the unification with respect to the conservation of the well-formedness of TFSs.

Definition 9 Unification revised

The unification $F \sqcup F'$ of two TFSs $F = \langle Q, \overline{q}, \theta, \delta \rangle$ and $F' = \langle Q', \overline{q'}, \theta', \delta' \rangle$ is given by $\{F \sqcup F' \sqcup MGsat(\theta(\overline{q}) \sqcup_t \theta(\overline{q'}))\}$. The result, if unification does not fail is a non-empty set containing well-formed TFSs only. $\square$

2.4 Credulous Default Unification

The general idea of default unification is to maintain as much of the default information as possible, as long as it does not conflict with the strict information. This notion is totally incorporated by the definition of *credulous default unification* in (Carpenter, 1993), that embodies the full power of default unification in a simple and clear manner. Although Carpenter uses FSs in his description, we easily can apply the definition of credulous default unification to TFSs. All we have to do, is to consider the subsumption ordering on types or type definitions instead of FSs. Given that the strict information is conveyed by type feature structure F we have:

Definition 10 Credulous Default Unification

$$F \overset{<}{\sqcup}_c G = \{F \sqcup G' \mid G' \sqsubseteq G \text{ is maximal such that } F \sqcup G' \text{ is defined}\} \quad \Box$$

The credulous default unification of the defeasible information in F and the strict information in G is defined as the set of unifications between a TFS F and all TFSs G' subsuming G with a maximal amount of information compatible with F. In the defeasible TFS there is always a set of information compatible with the information in the strict TFS, since the case is included if we have only an empty set of TFSs (G' corresponds to *top*) that is consistent with the strict TFS. This implies that two TFSs can always be default unified.

We can state the following concerning the properties of the default unification operation:

Proposition 1

- It is always defined as already mentioned. The set $F \overset{<}{\sqcup}_c G$ is never empty.
- It is potentially ambiguous as we have a set containing TFSs.
- It is not commutative, we have therefore $\overset{<}{\sqcup}_c(F, G) \neq \overset{<}{\sqcup}_c(G, F)$. Though if $\sqcup(F, G)$ is defined then $\overset{<}{\sqcup}_c(F, G) = \overset{<}{\sqcup}_c(G, F)$. So the default unification operation in case strict information and defeasible information are consistent is reducible to the unification in the definition 9.

- If $H \in F \overset{<}{\sqcup}_c G$ then $F \sqsubseteq H$.
  All strict information is preserved as every result is expressed as the unification of the strict information and the compatible subset of information in the defeasible source.

- It is always finite, since the defeasible TFS has a finite number of atomic FS (Moshier, 1988) to be combined with the strict TFS.

- It is well-formed as we will see in the next section.

The alternative proposition of Carpenter to do skeptical default unification, where we get always a result as well though exactly one result is not an adequate option to realize for instance discourse processing in (Grover et al., 1994). Moreover possibly valuable default information that is consistent with the strict information will be thrown away.
In the working out of the algorithm we will stick to Carpenter's definition and refer simply to default unification.

Chapter 3

A Prescription of Well-formed Default Unification

The novelty in the algorithm firstly presented in (Alexandersson & Becker, 2003) is the approach, that most of the computation of the default unification can be covered by taking into acount types only. The suggestion is to break down the default unification process into a two stage operation. Referring to the most recent work (Alexandersson & Becker, 2007), we adopt the conception of two operations described as the *assimilation* process and the *overlay* process. Also we will meet the convention to use the terms *cover* representing the strict information and *background* referring to the defeasible information.

Assimilation is a preprocessing step that first computes the target types for the cover and background. After, the cover and the background will be translated to their appropriate target types.

Overlay performs type assignment by combining the information from the two FSs returned by the assimilation.

The latter operation should not be confused with the Overlay engine in (Alexandersson & Becker, 2007) representing the default unification operation together with the scoring function. In this work we will not refer to the latter functionality.

3.1 The Assimilation Process

The assimilation is the part of the algorithm that incorporates the main contribution to the computation of the default unification. A great part in the computation implies to operate on types, which at the same time motivates having a type hierarchy as a backing component in the default unification. The *type preprocessing* occupies the main part of the assimilation process. However the assimilation does not consider exclusively lattice operations. After having identified the appropriate target types for the cover and the background we need adequate operations that take over the translation: we call the operation performing the translation from the cover to the appropriate target type *specialization* and analogously the translation concerning the background *generalization*.

So given the type of the cover t_{co} and the type of the background t_{bg} the assimilation process comprises three operative components:

Type Preprocessing are lattice operations required for identifying $t'_{bg} \in \mathsf{Type}$ with the property $t'_{bg} \sqsubseteq t_{bg}$ and $t'_{bg} \sqcup_t t_{co} \neq \{\bot\}$ and $t'_{co} \in \mathsf{Type}$ where $t_{co} \sqsubseteq t'_{co}$ and $t'_{bg} \sqsubseteq t'_{co}$.

Specialization of cover to the MGSat of t'_{co}.

Generalization of background to t'_{bg}.

On the basis of a type hierarchy we can calculate directly the desired type that subsumes the background's type and that is consistent with the type of the cover. Since type preprocessing does not have to take into account the *internal structure*[1] of cover and background, the well-formedness issue during this operation is not relevant. A separate and profound analysis of the first component in the assimilation process, studying intricate eventualities appearing in multiple inheritance hierarchies is discussed in the next section.

Given that both specialization and generalization transforms the internal structure of a TFS, we explicitely examine as next in which manner the well-formedness condition is affected during the translation. We put forward that if the well-formedness condition is maintained in each step concerning a single operation during the assimilation, then it is also the case that the entire assimilation procedure respects well-formedness.

[1] We consider the internal structure of a TFS to be all substructures that are reachable from the root node.

3.1.1 Specialization

The underlying operation in the specialization is equal to the unification in the definition 9. The unification has two functions. Firstly, to check if the cover will satisfy the well-formedness of the target type and secondly, to execute the translation to the target type. Given the hierarchy in figure 2.3, suppose we want to specialize the well-formed TFS $t_2[A : false]$ to the type t_3, we would get a type clash at the feature A. This means that if unification fails, the well-formedness condition on TFS in the following operations is no longer valid. If this is the case then another target type for the cover must be identified. This would imply the detection of another target type for the background as well. So well-formedness during specialization is maintained, since the translation to the target type happens first if unification succeeds. During the whole default unification process we perform the unification always between the cover and the MGSat of a target type. Under these circumstances it is not necessary to apply the additional step in the definition of the unification on the root level of the arguments.

As already mentioned, the explicit definition of the MGSat is missing in (Alexandersson & Becker, 2007). We know that the appropriateness condition alone on TFSs induces many type hierarchies to be permissible. However the use of simple unification presupposes the condition to allow only one concrete version of the type hierarchy. In particular the additional step in the revised unification to unify also with the result's MGSat is redundant in type hierarchies where for all $t, t' \in \mathsf{Type}$, $f \in \mathsf{Feat}$ and $t \sqsubseteq t'$ we have $\mathsf{Approp}(f, t) = \mathsf{Approp}(f, t')$. Actually this constraint does not appear to be useful to be imposed on TFSs. So in order to maintain the condition on TFSs to satisfy their MGSat in type hierarchies entailed by the appropriateness condition, it is inevitable to apply the revised unification procedure. Hence it is not correct to factor out this consideration and to use the simple unification as in (Alexandersson & Becker, 2007).

3.1.2 Generalization

The definition of the generalization step is similar to the one in (Alexandersson & Becker, 2007). The following definition slightly differs, since we take also the MGSat of the target type into consideration. We assume additionaly a function, given a node q $\mathsf{Closure}(q)$ that computes the subset of nodes reachable from q including q. Given two types t, t_g, where $t_g \sqsubseteq t$ and $f \in \mathsf{Feat}$:

Definition 11 Generalization

Let $F = \langle Q, \overline{q}, \theta, \delta \rangle$ with $\theta(\overline{q}) = t$ and $MGsat(t_g) = \langle Q_g, \overline{q}_g, \theta_g, \delta_g \rangle$, then $\mathcal{G}(F, t_g)$ is the generalization of F to a type t_g is a TFS $\langle Q', \overline{q}', \theta', \delta' \rangle$, where

- $\overline{q}' = \overline{q}$
- $Q' = \{\overline{q}'\} \cup \{\mathsf{Closure}(q_i) | \delta_g(f, \overline{q}_g) =\downarrow \text{ and } \delta(f, \overline{q}) = q_i, q_i \in Q\}$
- $\theta' = \{\theta(q) = t_i | q \in Q' - \{\overline{q}'\}\} \cup \{\theta'(\overline{q}') = t_g\}$
- $\delta' = \{\delta(f_k, q_l) = q_m | q_l \in Q'\}$

□

During the generalization it is assured, that the well-formedness condition on TFSs is maintained. Due to subsumption ordering on types, the internal structure of some TFS t' is also appropriate to the MGSat of a type that subsumes the type t'.

Proof: Given that the background bg is a well-formed TFS $\langle Q_{bg}, \overline{q}_{bg}, \theta_{bg}, \delta_{bg} \rangle$, the appropriate prototype $MGsat(\theta(\overline{q}_{bg}))$ and the type definition of the target type $MGsat(t'_{bg})$ we know that:

- $t'_{bg} \sqsubseteq \theta(\overline{q}_{bg})$ and further $MGsat(t'_{bg}) \sqsubseteq MGsat(\theta(\overline{q}_{bg}))$
- Aducting the law of *transitivity* on subsumption ordering we can state that $MGsat(t'_{bg}) \sqsubseteq MGsat(\theta(\overline{q}_{bg})) \sqsubseteq bg$ □

3.2 Type Preprocessing

In this section we will emphasize the role of the preprocessing on types in the assimilation process during the computation of the default unification. The general notion of the preprocessing is to look for a type more general than the type of the background structure and a type more specific than the type of the cover, where both types are in subsumption relation.

In the following we formulate specifications for types referred in the last section to as target types. Let us assume, that t_{bg} is the type of the background and t_{co} is the type of the cover, then we have:

Definition 12 Abstract Background Type

The *abstract background type* t'_{bg} of t_{bg} is defined, if $\sqcup_t(t'_{bg}, t_{co}) \neq \emptyset$, where $t'_{bg} \sqsubseteq t_{bg}$. □

In other words an abstract background type is the type of a candidate among the supertypes of the original background structure, that has a *mlb* with the type of the cover. Its definition refers to the target type of the background. Analogously the abstract cover type represents the target type of the cover:

Definition 13 Abstract Cover Type

The *abstract cover type* t'_{co} of t_{co} is defined, if $T'_{co} = \sqcup_t(t'_{bg}, t_{co}) \neq \emptyset$ and $t'_{co} \in T'_{co}$. □

Resuming the terms yet defined together, we introduce the notion of the type configuration.

Definition 14 Type Configuration

A *type configuration* $\langle t'_{bg}, t'_{co} \rangle$ is a pair of type identifiers consisting of the abstract background type and the abstract cover type . □

Further we specify the resulting TFSs after the translation of the cover structure to the abstract cover type and the translation of the background structure respectively.

Definition 15 Abstract Cover Structure

Let co be a TFS of the cover argument, then $co' = co \sqcup t'_{co} =\downarrow$ is defined as the *abstract cover structure*. □

Definition 16 Abstract Background Structure

Let bg be a TFS of the background argument, then $bg' = \mathcal{G}(bg, t'_{bg})$ is defined as the *abstract background structure*. □

3.2.1 Account for Multiple Results during Type Preprocessing

What makes default unification in our account more complicated is the possibility, that due to the multiple inheritance hierarchy more than one abstract background type proves to be a candidate for a successful default unification. In the following we will see the most perspicuous constellations in the multiple inheritance hierarchies provoking the emergence of multiple results when preprocessing on types. Since we are considering the bounded partial orders (BPO)[2] ignoring the completeness property, the presented scenarios producing ambiguous results also imply the situations in BCPO hierarchies generating multiple results.

In the first introduced case multiple results is caused by having two or more type configurations, where the respective abstract background types have no subsumption relation among each other. The image in the figure 3.1 represents a typical occurrence of type relations in multiple inheritance hierarchies. Note that the example is not a BCPO, since $\sqcup_t(t_1, t_2)$ is $\{t_{bg}, t_3\}$. Given the abstract background type t_1 and t_2, we get the abstract cover types $\sqcup_t(t_1, t_{co}) = \{t_3\}$ and $\sqcup_t(t_2, t_{co}) = \{t_3\}$ [3]. We obtain as type configurations $\langle t_1, t_3 \rangle$ and $\langle t_2, t_3 \rangle$. We assume that the specialization of the cover t_{co} to t_3 is successful and yields co'_{t_3}, then we would generalize the background to t_1 and also to t_2 yielding in bg'_{t_1} and bg'_{t_2} respectively. At this point we can already verify, that the default unification may differ in the outcome when referring

[2] The depicted example hierarchies represent only the relevant fragment of some bigger structure containing a *top* and a *bottom*.

[3] Concerning the presented scenario, a similar source for multiple results would be the case if the abstract cover types subsume each other. That case though certainly does not imply the to have potentially two equal results at the end

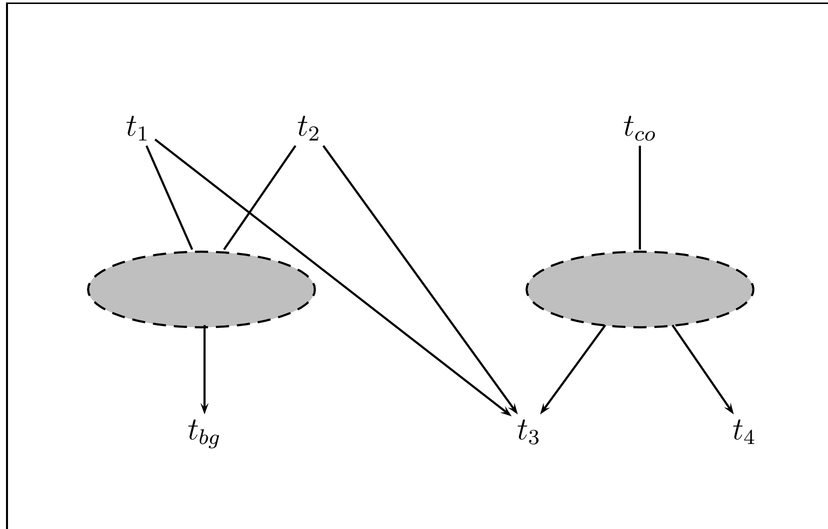

Figure 3.1: Multiple results due to two abstract background types that have no subsumption relation. The grey cloud indicates an area within all possible types except *top* and *bottom* may occur in a partial order, where the subsumption relation between the types below and above the cloud still holds.

to $\langle t_1, t_3 \rangle$ or to $\langle t_2, t_3 \rangle$. After the generalization, the two distinct abstract background structures contain different set of features that modify different parts within the abstract cover structure. But this constellation in the hierarchy may also result in equal TFSs. Such an outcome is unpredictable neither during the type preprocessing nor on the level when combining the abstract cover structure with the abstract background structure[4].

If we want to capture all relevant type configurations we have to keep track of every path leading from the background to its supertypes, even if on some path an abstract background type has already been detected and specialization of cover to the abstract cover type succeeds. It is not possible to rule out that a more general abstract background type under the same path may yield also a successful specialization that would lead to a different relevant result during the default unification. Figure 3.2 displays that the two

[4]The composition of the abstract background structure and the abstract cover structure is described by the overlay operation we specify later on.

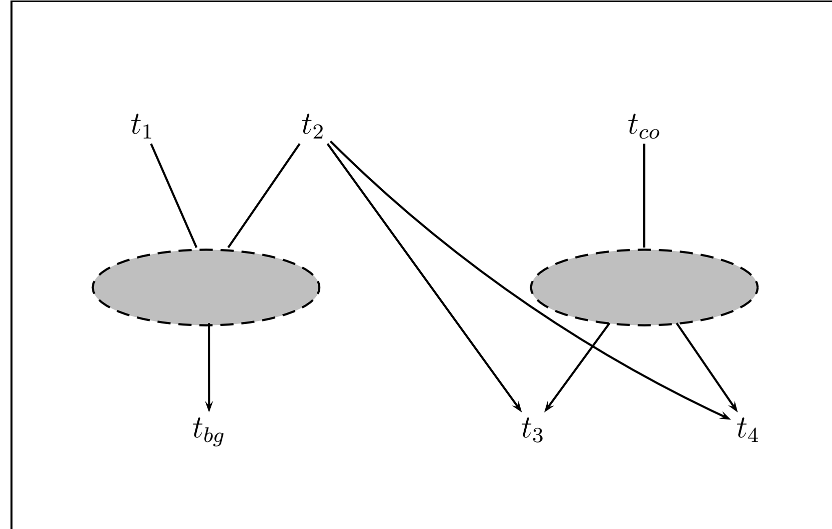

Figure 3.2: Multiple results due to two different abstract cover types, that are not under subsumption relation.

type configurations have different abstract cover types that do not subsume each other. We identify the valid type configurations $\langle t_2, t_3 \rangle$ and $\langle t_2, t_4 \rangle$, since t_2 is the only abstract background calculating abstract cover types $\sqcup_t(t_2, t_{co}) = \{t_3, t_4\}$. The sample hierarchy implicates also the case with two type configurations containing two different abstract background types subsuming each other. Consequently as far as the search concerns, it is not feasible to stop querying the next supertype even if the currently visited type turns out to be a valid abstract background type.

This particular behaviour we attribute to the allowance of the non-determinism type hierarchies, see definition 2 on page 12. A crucial point that now arises is that Carpenter's definition presumes to not concur in this observation. According to Carpenter we consider structures subsuming the background with an maximal amount of information to be compatible with the cover. It appears that there is a contradiction to our observation that an abstract background type candidate detected on the path where some more specific abstract background type has been found already, would not represent a potential abstract background structure. We postpone this issue and come back to it later on.

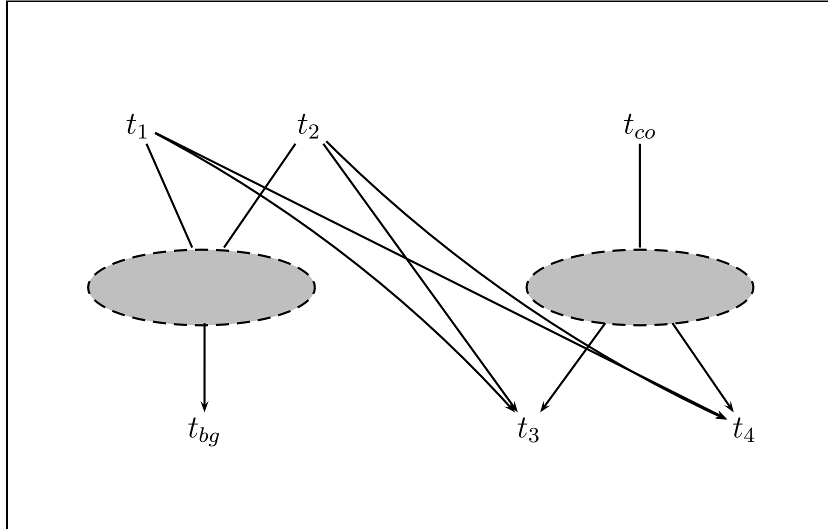

Figure 3.3: The number of results corresponds to the cartesian product of the number of "different" abstract backgrounds and abstract covers.

The image in figure 3.3 comprises the latter situation together with the displayed scenario in the figure 3.1. We get now two different abstract background types and two different abstract cover types where each are pairs that do not subsume each other. In this case we have four valid type configurations for the overlay process, that are $\langle t_1, t_3\rangle$, $\langle t_1, t_4\rangle$, $\langle t_2, t_3\rangle$ and $\langle t_2, t_4\rangle$. Generally, the maximal number of possible results due to the preprocessing is calculated by the following proposition:

Proposition 2

The number of multiple type configurations is equal to the cartesian product between the set of abstract background types T_{bg} and the set of abstract cover types T_{co}, if the following conditions hold:

- $\forall t, t' \in T_{bg}$ it is the case that $t \not\sqsubseteq t'$ and $t \neq t'$.
- $\forall t, t' \in T_{co}$ it is the case that $t \not\sqsubseteq t'$ and $t \neq t'$.
- $\forall t \in T_{bg}$ and $\forall t' \in T_{co}$ it is the case that $t \sqsubseteq t'$.

3.2.2 Termination of the Search for valid Type Configurations

The detection of valid type configurations is driven by the identification of abstract background types. An important issue that has not been specified yet is the break condition, i. e., when the search of the abstract background types terminates.

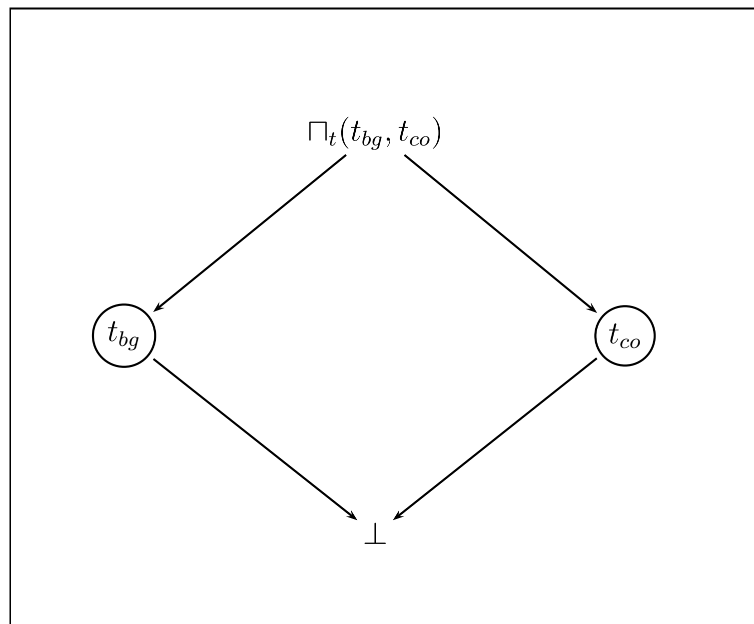

Figure 3.4: The break condition during the search for abstract background types in BCPO hierarchy holds if the supertype of the background type is not subsumed by the least upper bound between the background type and the cover type. In this case the type of the cover is at the same time the abstract cover type.

In the given analysis and algorithm in (Alexandersson & Becker, 2007) the search for abstract background types is narrowed down to the candidates that are situated exclusively on the path between the type of background and the *lub* of cover and background. During the type preprocessing we would impose a constraint on the background supertypes to be subsumable by the *lub* type. The exploration of the abstract background types would

stop for those paths when the next abstract background type in question is not anymore subsumed by the type of *lub*. The *lub* node represents the most specific TFS with the internal structure, i. e., the intersection of features that occurs in the background as well as in the cover. Hence if it is the case, that the search for some potential abstract background type arrives at the *lub* type, the "default" resulting structure of a default unification is the cover argument itself. It is evident when taking a look at figure 3.4 that the type of the abstract cover type is equal to the type of the cover.

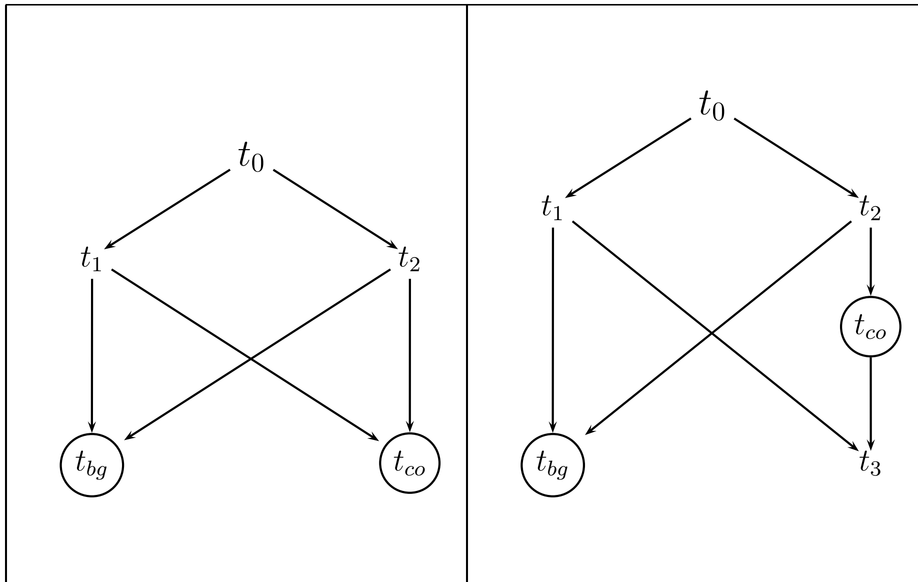

Figure 3.5: On the left the abstract background type t_1 is subsumed by the *mub*, $t_1 \in \sqcap_t(t_{co}, t_{bg})$, while on the right it is not subsumed by a *mub* of cover type and background type.

In the next step we want to clarify if this break condition is still valid in multiple inheritance hierarchies performing the non-determinism aspect. Instead of a constraint on abstract background types to be a subtype of the *lub* type, we apply a constraint on abstract background types to be subsumed by the type of *mub*. On the left side of figure 3.5, we have as *mub* between the types of cover and background t_1 and t_2, that are at the same time also two valid abstract background types. The adoption of the constraint imposed on abstract background types to be subsumable, correctly identifies $\langle t_1, t_{co} \rangle$ and

$\langle t_2, t_{co} \rangle$ as type configurations in the given situation. Though if we replaced the node of the cover's type definition by some other type definition and shifting the former one level higher in the hierarchy, we would modify the *mub* between cover and background while keeping the subsumption relations concerning the other types. The right side of the image demonstrates that the *mub* between the types of cover and background is t_2 and further t_2 is a valid abstract background type. However it is justified to state that in addition to t_2 also the type t_1 is still an abstract background type, hence a source for an additional result during the default unification. A modification of the constraint by shifting the layers of supertypes to be explored higher than in the version of (Alexandersson & Becker, 2007) would detect the type t_1 as an abstract background type. A suitable option for the search termination is to stop the expansion to the next supertypes for those paths, when the currently visited type subsumes the *mub* type. Applying the modified constraint to the scenario on the right side, we would correctly capture the type configurations $\langle t_2, t_{co} \rangle$ and $\langle t_1, t_3 \rangle$.

The just formulated constraint would still limit the search space to some extent and cope with the last described scenario provoked by the non-determinism in the type hierarchy to deliver all possibly results during the default unification. In the following we revise the definition of the abstract background by taking into account the last insight into a feasible constraint imposed on abstract backgrounds.

Definition 17 Abstract Background Type Revised

An abstract background type is the type of a candidate among the supertypes of the original background, that has a *mlb* with the type of the cover structure, and does not subsume the *mub* between type of cover and type of background. □

Now the question arises, if even a scenario exists, where the last stated constraint does not hold anymore? The concrete example displayed in figure 3.6, suggests that the type configuration with t_0 as abstract background and t_7 as abstract cover type is plausible in addition to $\langle t_4, t_6 \rangle$. After all, t_7 contains the information in the cover structure and valid information communicated by the background structure added to the properties inherited

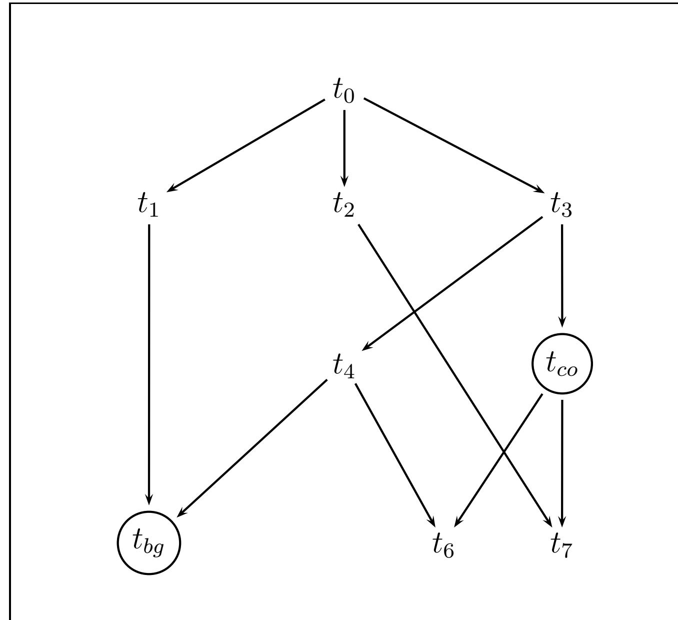

Figure 3.6: Is t_1 a valid abstract background type, though it subsumes the *mub* t_3 between t_{co} and t_{bg} ?

by t_2. Here again allowing t_0 to be an abstract background type, we abandon a partition of Carpenter's definition of credulous default unification. The background's supertype t_0 that is supposed to be a valid abstract background type subsumes the already identified abstract background type t_4. In general accepting abstract background types subsuming the *mub*, here t_3, infers to accept also abstract background types subsuming other candidates. Those are the detected abstract background types subsumed by the type of *mub* and especially the type of *mub* itself. At this point it is questionable to allow abstract background types subsuming the type of cover that are not at the same the *mub* between cover and background. We assume that *mub* of the cover and background is a valid abstract background type. Given by the definition that the *mub* subsumes the cover type we have always the constellation that the cover type is equal to the abstract cover type, since the *mlb* between the *mub* (abstract background type) and the cover type is again the cover type. This is analogous for all background's supertypes subsuming the *mub* like t_0 in figure 3.6. Consequently t_7 is not an option for being an

abstract cover type, since the *mlb* of the questionable abstract background type t_0 and t_{co} is only t_{co}. Given that t_7 is excluded from being an abstract cover type we are able to exclude t_0 from being an abstract background type as well. Hence the revised notion of the abstract background type still holds.

3.2.3 Reformulation of Carpenter's Definition for BPO Hierarchies

Now we want to clarify the issue if we are still able to stick to Carpenter's definition, although our analysis reveals that valid abstract background types do not need to be the most specific ones in the subsumption ordering. Our claim is that allowing more general abstract background type to be a source of multiple results does not necessarily conflict with Carpenter's core notion of credulous default unification.

Carpenter suggests to consider all possible subsets of the shared structures in background and cover to be the most specific structures in the subsumption ordering that hold the condition to unify with the cover. From this it follows that the unifications of the cover and the most specific generalized backgrounds are most specific as well. We actually can circumscribe the default unification of two TFS to be FSs subsumed by valid type definitions, i. e., the MGSat of the abstract cover type, that are closest in the hierarchy to the structure of the "non-existing" unifier. Hence those are in particular well-formed TFSs that are most specific and subsume the "non-existing" unifier. We keep on mind that Carpenter's definition of default unification suffices for taking into account all relevant results that arise in a hierarchy following the BCPO condition while in a hierarchy that is only a bounded partial order, the definition would ignore possible results.

We reformulate Carpenter's fundamental credulous default unification as follows:

Definition 18 Credulous Default Unification Revised

$$F \overset{<}{\sqcup}_c G = \{F \sqcup G' \mid G' \sqsubseteq G \text{ such that } F \sqcup G' \text{ is defined and maximal}\}$$

□

The definition says that the result of credulous default unification between the cover F and the background G are represented by unifications between F and G', where G' subsumes the background G, that are defined and most specific.

In the analysis of the constellations in the BPO hierarchies up to this point we have identified an upper limit of the search space for abstract background types generating a valid result during default unification. Though it is not clarified if all results contributed by those abstract background types also produce most specific results.

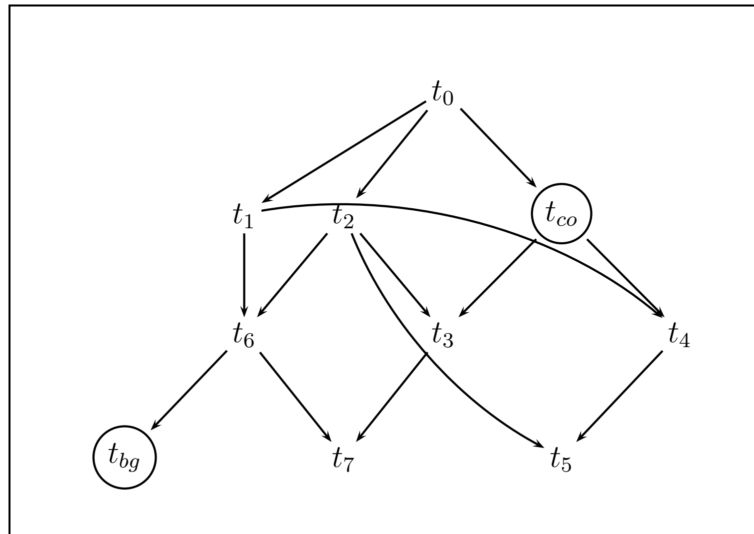

Figure 3.7: This sample hierarchy features type configurations that produce also results during default unification that are not most specific. The observation reveals that the members of the type configurations $\langle t_0, t_{co} \rangle$, $\langle t_2, t_3 \rangle$ and $\langle t_6, t_7 \rangle$ are related in parallel by the subsumption ordering. We argue that given the latter type configuration, the two other options are not relevant anymore.

On the basis of the sample hierarchy in the figure 3.7 we seek to give a specification of type configurations, that are relevant for further processing to identify a most specific result during default unification. Our fundamental account to sources of multiple results during default unification enables to

identify the appropriate type configurations linked to the scenario with two or more abstract background types not under subsumption relation and the constellation with two or more abstract cover types not under subsumption relation. We assume that all type configurations available from the hierarchy subsequently produce results, i. e., specialization does not fail.

Given the hierarchy in figure 3.7 we identify the following type configurations:

1.) $\left.\begin{array}{l}\langle t_1, t_4\rangle\\ \langle t_2, t_5\rangle\end{array}\right\} : \textit{where } t_1 \not\sqsubseteq t_2, t_1 \not\sqsupseteq t_2$

2.) $\left.\begin{array}{l}\langle t_2, t_5\rangle\\ \langle t_2, t_3\rangle\end{array}\right\} : \textit{where } t_5 \not\sqsubseteq t_3, t_5 \not\sqsupseteq t_3$

3.) $\langle t_6, t_7\rangle$

4.) $\langle t_0, t_{co}\rangle$

We have altogether five distinct type configurations, where three of them included in point 1 and 2 are evoked by the scenarios featuring the non-determinism. The type configurations under point 3 and 4 are due to ordinary multiple inheritance and have no involvement to non-determinism. If we compare $\langle t_0, t_{co}\rangle$ and $\langle t_2, t_3\rangle$ we can discover that $t_0 \sqsubseteq t_2$ and $t_{co} \sqsubseteq t_3$. It is evident that the abstract cover structure and the abstract background structure translated by $\langle t_0, t_{co}\rangle$ subsume respectively the abstract cover structure and the abstract background structure generated by $\langle t_2, t_3\rangle$. Hence the type configuration $\langle t_0, t_{co}\rangle$ during default unification results in a TFS that subsumes the result contributed by $\langle t_2, t_3\rangle$. Since $\langle t_0, t_{co}\rangle$ does not yield a most specific result during the default unification, we want the algorithm to exclude such TFSs. Analogously $\langle t_2, t_3\rangle$ is related to $\langle t_6, t_7\rangle$, so we do not want the type configuration $\langle t_2, t_3\rangle$ as well to be included for further aspects in the processing.

In order to exclude results that are not most specific, we need to specify a subsumption ordering on type configurations.

Proposition 3

Given two type configurations with $t_{conf} = \langle t_{conf_1}, t_{conf_2} \rangle$ and $t'_{conf} = \langle t'_{conf_1}, t'_{conf_2} \rangle$, then we say that t_{conf} is more specific than t'_{conf} , if $t'_{conf_1} \sqsubseteq t_{conf_1}$ and $t'_{conf_2} \sqsubseteq t_{conf_2}$. We adequately write $t'_{conf} \sqsubseteq t_{conf}$ to express the subsumption ordering on type configurations.

Additionally we propose a mapping from tuples consisting of translated TFS, i. e., the abstract background structure and the abstract cover structure, to the ordinary type configuration for the purpose to ease the specification of redundant type configurations in the algorithm of well-formed default unification.

Proposition 4

Given a tuple $T_{conf} = \langle bg', co' \rangle$ with $bg' = \langle Q_{bg'}, \overline{q}_{bg'}, \theta_{bg'}, \delta_{bg'} \rangle$ and $co' = \langle Q_{co'}, \overline{q}_{co'}, \theta_{co'}, \delta_{co'} \rangle$, we assume $\zeta(T_{conf}) = \langle t'_{bg}, t'_{co} \rangle$ where $\theta(\overline{q}_{bg'}) = t'_{bg}$, $\theta(\overline{q}_{co'}) = t'_{co}$.

Now we are able on the level of type preprocessing to make statements over the background's supertypes that do not subsume the *mub* of cover and background, if they yield to most specific results in the default unification. Though as mentioned already given the scenario in figure 3.1, where we have the same abstract cover type in two distinct type configurations type, we have still to consider the possibility to obtain equal results.

3.3 The Algorithm

In this section we point out the algorithm of the default unification. First we put the introduced processings concerning the precomputational issues into appropriate sequential steps in order to give a complete formalization of the assimilation process. The course of action of the assimilation algorithm slightly differs from (Alexandersson & Becker, 2007). Firstly generalization is not executed until specialization does not fail. Consequently there is at most one generalization involved per default unification procedure. Further we do not unify iteratively with all type definitions on the way towards the target type. Given the target types, we consider it to be more sound to try to unify first with the prototype structure of the target type, whose unifier would be in principle also the optimal and desired solution. If unification

fails at this point then the next appropriate target type of the background would give us the next target type for the cover. This approach would rather start at the expected result and then gradually move away from the more specific to the more general target type of the cover. A dual approach to the assimilation is to reverse the type preprocessing. Instead of qualifying the abstract cover type by the detected abstract background type we could do it also the other way around, that is, checking the subtypes of the cover type if they have a *mub* with the type of the background. The further processing is then analogue to the algorithm presented in the following.

Algorithm 1 Assimilation

Let *co* and *bg* be two TFS(covering and background) such that $co = \langle Q_{co}, \overline{q}_{co}, \theta_{co}, \delta_{co} \rangle$ and $bg = \langle Q_{bg}, \overline{q}_{bg}, \theta_{bg}, \delta_{bg} \rangle$. Further we have $t_{bg} := \theta_{bg}(\bar{q}_{bg})$ and $t_{co} := \theta_{co}(\bar{q}_{co})$. The assimilation of *co* and *bg*, $\mathcal{A}(bg, co) = \Omega$, where Ω is a set of pairs of bg' and co' such that:

	CONTROL FLOW	COMMENTS
(1) if		
	$t_{bg} \sqsubseteq t_{co}$	
then		
	$bg' = bg,\ co' = co$ $\langle bg', co' \rangle \ \in \Omega$	algorithm terminates, assimilation results into a singleton
(2) if		
	$t_{co} \sqsubseteq t_{bg}$	
	if	
	$\sqcup(co, MGsat(t_{bg})) \neq \emptyset$	*specialization* has a valid result
	then	
	$CO' = \sqcup(co, MGsat(t_{bg}))$, $bg' = bg$ $\{bg'\} \times CO' \subseteq \Omega$	algorithm terminates and results into the cartesian product of the *bg* singleton and the result of the *specialization*
	else	
	go to (3.1) with $\mathcal{T}(t_{bg}), t_{mub} = t_{co}$	unification has failed during *specialization*

(3)	else	
	$T_{mubs} := \sqcap_t(t_{bg}, t_{co})$	find minimal upper bounds of t_{co} and t_{bg}
	for each $t_{mub} \in t_{mubs}$ go to (3.1) with $\mathcal{T}(t_{bg})$	
(3.1)	$\mathcal{T}(t) := \{t'_{bg} \| t'_{bg} \sqsubseteq t$ with t'_{bg} is maximal and $\sqcup_t(t'_{bg}, t_{co}) \neq \emptyset\}$	find appropriate supertypes of t_{bg}
	for each $t_{bg'} \in \mathcal{T}(t_{bg})$ go to (3.2)	
(3.2)	if	
	$t'_{bg} \sqsubseteq t_{mub}$	break condition
	then	
	HALT	expansion of node stopped at t'_{bg}
	else	
	for each $t_{mlb} \in \sqcup_t(t'_{bg}, t_{co})$ go to (3.3)	
(3.3)	if	
	$\langle t'_{bg}, t_{mlb}\rangle \not\sqsubseteq t'_{conf}, \forall\, t'_{conf}$ where $t'_{conf} = \zeta(T'_{conf}), T'_{conf} \in \Omega$	block $t_{bg'}$ if result would not be most speficic
	then	
	go to (3.4)	
	else	
	go to (3.1) with $\mathcal{T}(t'_{bg})$	
(3.4.)	if	
	$\sqcup(co, MGsat(t_{mlb})) \neq \emptyset$	*specialization* has a valid result
	then	

$CO' = \sqcup(co, MGsat(t_{mlb}))$ $bg' = \mathcal{G}(bg, t'_{bg})$ $\{bg'\} \times CO' \subseteq \Omega$ go to (3.1) with $\mathcal{T}(t'_{bg})$	algorithm continues to scan bg's supertypes - the intermediate result, the cartesian product of the *bg* singleton and the result of the *specialization* represents a subset of the final result
else go to (3.1) with $\mathcal{T}(t'_{bg})$	unification has failed during *specialization*

(1.) represents the trivial case concerning the type preprocessing. In the case of the scenario that the background type subsumes the cover type, the type configuration is given without computing new target types. In particular it is not required to navigate through the hierarchy, since both cover and background have the target types already represented by their own types. It follows that the assimilation for those kind of argument will not produce multiple results during the default unification.

Whereas in (2.), if specialization is possible then the result of assimilation corresponds to the cartesian product of the type feature structures resulting from unification between cover and background. Up to this point we have in principle an equivalence between the effect of the default unification process and the unification process between cover and background. However if specialization to the cover's target type fail, we have to shift the target type for the background towards the *mub* between t_{bg} and t_{co}, that is eventually handled by the last and main case of the algorithm.

(3.) says that there is no subsumption relation between cover and background, hence it requires to search after appropriate background supertypes. During the allocation of possible abstract background types represented by the function $\mathcal{T} : \mathsf{Type} \rightarrow \mathcal{P}(\mathsf{Type})$, we assume that $\mathcal{T}$ returns all next supertypes that have a *mlb* with the type of the cover that is not equal to the empty set. The tracking of a path is pursued as long as the visited nodes are not subsumed by the *mub*, otherwise the generation of potential abstract background types is stopped.

The production of only most specific results is given by comparing the abstract background types t'_{bg} and the associated abstract cover types t_{mlb} to the already used type configurations that are recorded along the assimilation process. Therefore we scan Ω containing the results up to that point processing and tranform the tuple of translated structures to the representation of type configuration with the function given in the proposition 4. Consequently, if the current type configuration is not rejected, that is it does not subsume any of the already employed type configurations, we do the translations to the appropriate target types. We take as intermediate result referring to some abstract background type as the cartesian product of the specialization's outcome, a set of abstract cover structures, and the result of the generalization that has always a unique result. At the end Ω represents a collection of tuples containing an abstract background structure and an abstract cover structure, that are most specific in the analog sense we specified subsumption ordering on type configurations.

3.3.1 Algorithm of the Default Unification

Before defining the default unification procedure including the overlay functionality, we need a function given a type feature structure TFS, that points from some appropriate feature of TFS to the internal structure, again a TFS.

Definition 19 Feature Value

Let $F = \langle Q, \overline{q}, \theta, \delta \rangle$ be a well-formed TFS and $f \in \mathsf{Feat}$, such that f is defined for $\delta(\overline{q})$. Then the Feature Value of F with respect to f, $\mathcal{F}(F, f)$ is a TFS $F' = \langle Q', \overline{q}', \theta', \delta' \rangle$, where

- $\overline{q}' = \delta(f, \overline{q})$
- $Q' = \mathsf{Closure}(\overline{q}')$
- $\theta' = \{\theta(q_i) = t_i | q_i \in Q'\}$
- $\delta' = \{\delta(f_k, q_l) = q_m | q_l \in Q'\}$

□

Finally we give the specification of the default unification procedure made up of the specified assimilation algorithm and the now introduced overlay operation, that determines how the pairs of abstract background structures and abstract cover structures are to be combined.

Algorithm 2 Default Unification

Let co and bg be two TFSs. The default unification $defaultunify(bg, co)$ is determined by the following:

Let $\mathcal{A}(bg, co) = \Omega$ then for each $\langle bg', co' \rangle \in \Omega$ and

$$bg' := \langle Q_{bg'}, \overline{q}_{bg'}, \theta_{bg'}, \delta_{bg'} \rangle$$
$$co' := \langle Q_{co'}, \overline{q}_{co'}, \theta_{co'}, \delta_{co'} \rangle$$

do $overlay(bg', co') = \langle Q_o, \overline{q}_o, \theta_o, \delta_o \rangle$, where

- $\overline{q}_o := \overline{q}_{co'}$
- $\theta_o(\overline{q}_o) := \theta_{co'}(\overline{q}_{co'}) \cup \theta_{bg'}(\overline{q}_{bg'})$
- $\delta_o(f, \overline{q}_o) := f \in \mathsf{Feat}$:
  (1.1) $\delta_{co'}(f, \overline{q}_{co'})$ if $f \notin \mathsf{Appfeat}(bg')$ and $f \in \mathsf{Appfeat}(co')$,
  (1.2) $defaultunify(\mathcal{F}(bg', f), \mathcal{F}(co', f))$ if $f \in \mathsf{Appfeat}(bg')$ and $f \in \mathsf{Appfeat}(co')$

After assimilation the combining rule of overlay basically looks for shared features in the abstract cover structure and in the abstract background structure, where default unification applies on the arguments given by the definition 19. The recursion in the specification causes that the default unification algorithm gets executed successively each time a stage deeper in the intermediate structures of the arguments. In particular substructures of cover and background are modified after each call of the default unifier, i. e., each time overlay combines the previously assimilated parts stemming from background and cover respectively and determines the resulting types of the structures on the current level. Thus instead of attaching single values to the shared features we have to assign a set of TFSs due to possible multiple results resulting from the overlay of more than one pair of assimilated backgrounds and covers. In this theoretical analysis we suppress a representational formalization concerning this issue. However the subsequent chapter that describes the implementation based on this algorithm will explicitely propose a solution to treat

more complex TFS that allows a transition function $\delta : \mathsf{Feat} \times Q \rightarrow P(Q)$ with features pointing to a set of TFSs.

Finally, as for the case that we have a feature only in the cover we just need to absorb the feature value pair to the result. In addition we never have the constellation with the assimilated background consisting features that are not in cover, because a valid type configuration by definition entails that the abstract background type subsumes the abstract cover type. It follows that at least all features in the abstract background type are also in the abstract cover type.

Chapter 4

Prerequisites—Practice

This chapter introduces the underlying system for the implementation of the default unifier consisting of the type description language—$\mathcal{TDL}$ by Ulrich Krieger and Ulrich Schäfer (Krieger, 1995) and the flop preprocessor by Ulrich Callmeier (Callmeier, 2001). Further we go into modifications concerning the requirements the default unification sets to the two components.

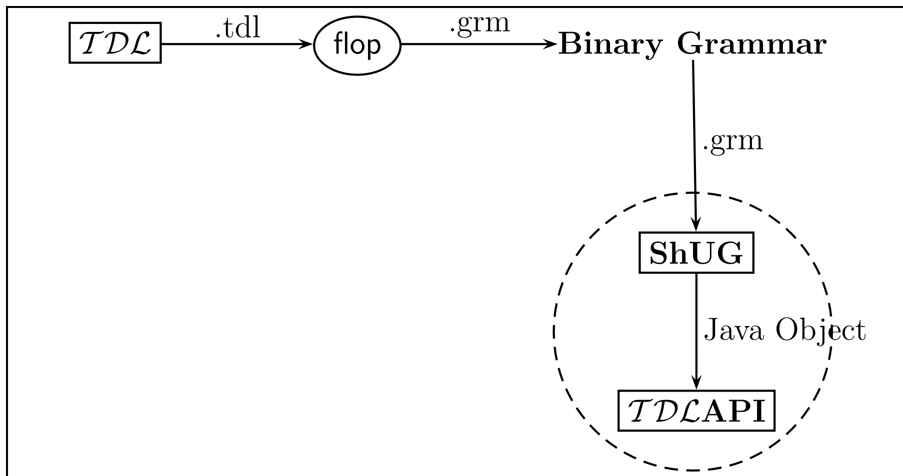

Figure 4.1: With $\mathcal{TDL}$ as starting point the system passes the type hierarchy in an intermediate format generated by the flop preprocessor to a wrapper class called ShUG ("Shallow Unification Grammars", see 4.2.2) that stores the type hierarchy in a java object.

4.1 Type Description Language—$\mathcal{TDL}$

The main contribution of $\mathcal{TDL}$ to the thesis is that $\mathcal{TDL}$ provides the implementation of a type system. $\mathcal{TDL}$ is a unification based grammar development environment and realizing a grammar formalism for TFSs (Krieger, 1995) supporting HPSG and also other feature-based grammars. Within the DISCO project at the DFKI a HPSG grammar for the German-language was developed. Together with $\mathcal{UDINE}$, an untyped unification engine allowing the use of other logical connectives, they build the main modules in DISCO. Besides providing the environment for developing grammars, the main task of $\mathcal{TDL}$ in DISCO is taking over the unification of two TFSs, deciding by referring to the information provided by the type hierarchy, whether they are by type definition compatible or not.

The Expressivity of $\mathcal{TDL}$ sticks with the set-theoretical semantics worked out in (Smolka, 1988). In particular $\mathcal{TDL}$ represents a subset of first order logic. As for TFSs specified in 2, $\mathcal{TDL}$ is an adequate and sufficiently powerful tool for the purpose of our TFS language. A type definition $s :=< t, \phi >$ in $\mathcal{TDL}$ is a compact representation of a complex expression, where t is a type constraint concerning super- and subtype relationships and the feature constraints (attribute-value pairs) ϕ states the necessary features and their values that are in our case again type definitions.

Formally

$$s :=< t, \phi >, \qquad \phi := [\ ATTR1 \quad q, ATTR2 \quad p\ ]$$

corresponds to the following first order formula, where attributes/features are interpreted as binary relations and types as unary predicates,

$$\forall.s(x) \rightarrow t(x) \wedge ATTR1(x, q) \wedge ATTR2(x, p)$$

and below we have the adequate type definition realized in the $\mathcal{TDL}$ format.

```
s := t & [ATTR1 q , ATTR2 p]
```

4.1.1 The Structure of $\mathcal{TDL}$ Grammars

The definition of TFS in 6 does not exhaust the full power of $\mathcal{TDL}$. Type definitions in $\mathcal{TDL}$ allows also to define other constructs like instances or templates. Since the main focus we have here is on type definitions, we will not go into the whole syntax of the $\mathcal{TDL}$ grammar. For more details of the full language in $\mathcal{TDL}$ one might find answers in the $\mathcal{TDL}$ reference manual (Krieger, 1994b).

The relevant BNF forms formulating type definitions in $\mathcal{TDL}$ is given in the following specification of the input syntax in BNF form, adapted from the definitions in the $\mathcal{TDL}$ reference manual concerning type definition.

- `type-def` → `type avm-def '.'`
- `type` → `IDENTIFIER`
- `avm-def` → `':=' conjunction`
- `conjunction` → `term | term '&' conjunction`
- `term` → `type | feature-term`
- `feature-term` → `['['attr-val-list']']`
- `attr-val-list` → `attr-val [',' attr-val-list]`
- `attr-val` → `attr-list conjunction`
- `attr-list` → `attribute ['.' attr-list]`
- `attribute` → `IDENTIFIER`

A very convenient property in the specification is defining a type by the direct supertype relationships the type performs. Besides stating type subsumption relation, there is an effect to save writing each time the feature constraints that are already defined among the supertypes. If we want to narrow down inherited feature constraints, we have to overwrite the corresponding feature constraints while respecting the subsumption ordering in the hierarchy.

Below is a sample of type definitions viz type hierarchy in $\mathcal{TDL}$ format, that is also part of the built type hierarchy for testing the default unification operation. It is easy to recognize that all types inherit from one single type *top* and the types t_{32} and t_{34} respectively inherit from 2 different supertypes, thus realizing multiple inheritance. The corresponding directed acyclic graph to this type hierarchy is displayed in the appendix by the figure B.2.

```
:begin :type.

t26 := *top* & [FT26 *top*].

t27 := t26 & [FT27 t1].

t28 := t27 & [FT26 t1].

t29 := t27 & [FT26 t1, FT27 t5].

t30 := t27 & [FT26 t7, FT27 t2].

t31 := t28 & [FT28 bool].

t32 := t28 & t29 & [FT26 t9, FT27 t8].

t33 := t29 & [FT26 t11, FT27 t9].

t34 := t30 & t32 & [FT26 t12, FT27 t8].

:end :type.
```

Figure 4.2: A sample of type definitions in $\mathcal{TDL}$

4.1.2 Open World vs. Closed World

$\mathcal{TDL}$ provides a tool to write grammar in interactive mode. Unlike LOGIN (Aït-Kaci & Nasr, 1986), ALE (Carpenter & Penn, 1998) or LIFE (Aït-

Kaci, 1993), $\mathcal{TDL}$ does not enforce the grammar writer to specify the type subsumption a priori. $\mathcal{TDL}$ allows the grammar writer to choose between a type hierarchy with an open world or with a closed world concept. $\mathcal{TDL}$ applies *sort types* to represent the close world semantics. They correspond to the partial order we have on types. The so-called *avm types* in $\mathcal{TDL}$ would correspond to type definitions. We see in the figure 4.3 the different behaviour of *avm types* in the closed world and open world assumption respectively.

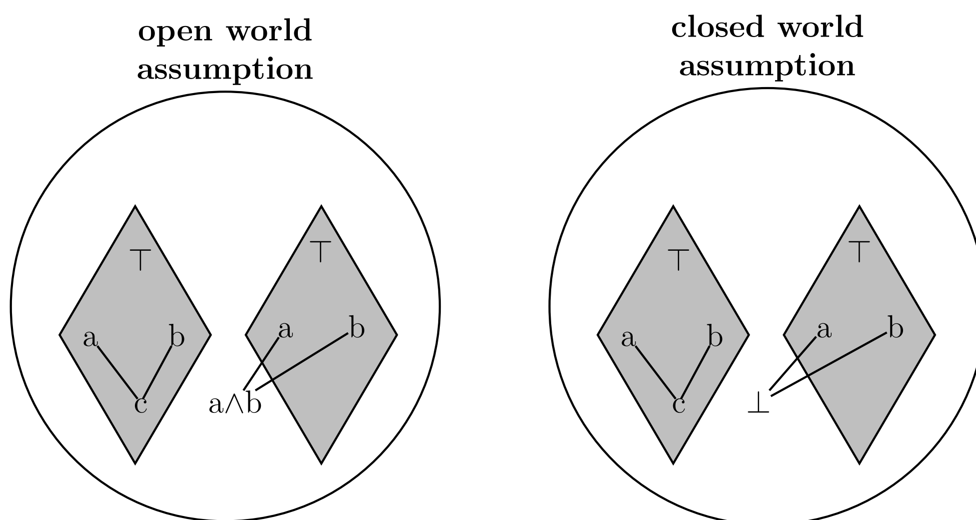

Figure 4.3: The open world case to the left incorporates $\mathcal{TDL}$ to integrate a newly generated type definition into the type hierarchy if we have no explicit join of the *avm types* a and b. The same case would produce the *bottom* type as output when accepting the closed world assumption.The grey area represents the domain of the type definitions in our type hierarchy.

In the *glb* computation, $\mathcal{TDL}$ differs between the external *glb* that deals with avm types and the internal *glb* concerned with type subsumption relation on sort types. Both approaches work with the type encoding method presented in (Aït-Kaci, Boyer, Lincoln, & Nasr, 1989), that enables the execution of lattice computation in $\mathcal{O}(n)$. The bit-encoding method of the types will be discussed in detail later on.

One crucial point that arises here is the fact that $\mathcal{TDL}$ offers the possibility to modify the type hierarchy afterwards by redefinition of types or the extension of the type hierarchy when creating new type definitions during unification, reflecting the option having a type hierarchy with an open world nature. Whereas the latter choice represents a reasonable device of the incremental grammar development, concerning the needs of a type hierarchy as being a "playground" in order to test the default unification operation, we accept the closed world assumption in the type hierarchy. Given the type definitions, the type hierarchy as such is initiated once at the beginning and will be not modified in the course of action. The initialization of the type hierarchy is carried out by reading the $\mathcal{TDL}$ grammar in the preprocessor flop.

4.2 The flop Preprocessor

The flop preprocessor generates an binary intermediate form of the $\mathcal{TDL}$ grammar form. The binary file is then read in by ShUG at runtime. It contains the information conveyed by the grammar sources , i. e., type hierarchy, type constraints. A point to mention here is that also flop is not designed to accept the full language of $\mathcal{TDL}$. The flop preprocessor defines a conjunctive subset of $\mathcal{TDL}$ that omits various forms of disjunctions and negations.

The preprocessor was implemented within Callmeier's master thesis as a part of a system that is called PET, see the figure 4.4. The aim in his work was to set up a modular environment for experimenting with different and the most influential approaches concerning algorithms for graph unification. The set of building blocks includes objects like *chart*, *agenda*, *grammar*, *type hierarchy* and *type feature structure*. Using alternative implementations of one object allows to have controlled experiments when comparing different approaches. While the evaluation refers to one aspect of processing, PET alters between the different unification algorithms.

4.2.1 The Functions of flop

The flop preprocessor covers an important range of tasks. The motivation of flop is having a conversion of the grammar source into a representation

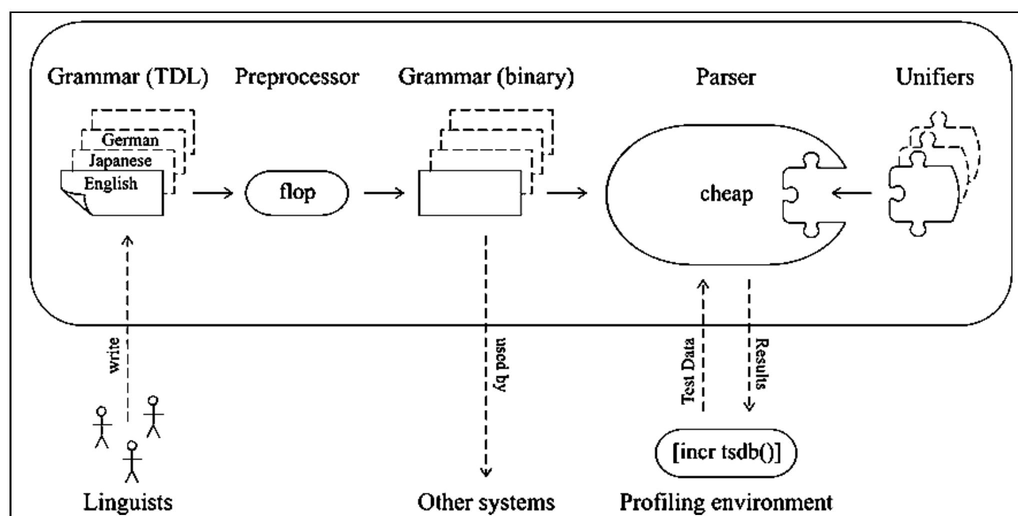

Figure 4.4: PET—System Overview and Experimental Setup

suitable for efficient processing. The features of flop are the following:

Type Expansions In the $\mathcal{TDL}$ grammar we allow the use of symbol names to represent types referring to complex constraints (type definitions). The configurable expansion technique taken from (Krieger & Schäfer,) reconstructs the idiosyncratic constraints of a type and also the constraints inherited from the supertypes.

Unfilling of type definitions After expansion, structures and substructures in type definitions that fulfil the condition of its maximal appropriate type are removed except for the root level ((Götz, 1993), (Gerdemann, 1995b)).

Inferring appropriateness conditions In the grammar, some feature f is introduced exactly once at some type definition. In the following, subtypes of that type definition inherit the feature f. The reuse of the same feature name in some other type definition will cause the generation of the binary output to fail. Moreover flop checks also, if the monotonocity condition at the feature values is respected. If we do not consider the subsumption ordering on the feature values, flop will send out an adequate warning. That finally means, that flop presupposes

the grammar to meet the appropriateness condition. Therefore it is the task of the grammar writer to meet the appropriateness condition in the built up type hierarchy.

Decomposition of the type hierarchy for FS encoding purposes. A set of feature configurations that each is assigned by unique identifier is defined to ease the unification processing. At runtime, a mapping from types to feature configurations allows an efficient check if two types have the same feature configuration. Given the feature configurations, for two types with different feature configuration we have an additional mapping from feature to the feature position and its inverse. That facilitates the unification process by iterating over the positions in one single TFS.

Constructing lower semi-lattice The mechanisms in PET requires that the type hierarchy is a BCPO, but it does not assume that the condition is considered by the grammar writer. During the semi-lattice computation flop possibly modifies the type hierarchy to accomplish the BCPO condition. The type hierarchy as it is specified by the grammar writer is represented in flop as a directed graph and is computed to an encoded image using the bit-coding technique in (Aït-Kaci et al., 1989).

For the efficiency purpose, it is reasonable to save recurring tasks such as syntax checking, uncovering consistency and the expansion of constraints. As well for this work, since there is no need for a dynamic type hierarchy it is convenient to factor out those tasks. Some of the features in flop during the construction of the type hierarchy are adopted in the implementation of the default unifier. The technique to normalize feature constraints used during unification is related to the type expansion and the unfilling issue. The partial expansion of TFSs allows to save unification processes beyond the root level of two argument structures, if both fulfil the requirements of the prototype. Therefore the resulting TFS still remains unexpanded and unfilled for those internal structures equal to the most general satisfiers respectively. Further a modified approach of the iteration over features of a feature configuration is also realized in the implementation of the unification processes. Considering the last point among the functionalities of flop we will go more in detail in the next section.

4.2.2 ShUG

The output of flop consists of several sections in fixed order. Some of them, like *rules* section containing grammar rules or also the *lexicon* section containing the full form lexicon are not relevant for the default unification operation and so are left out in the discussion. In this context the important sections are the *toc*, *symbol table*, *hierarchy*, *constraints* and the *supertypes* sections. The section *toc* stores information of other section's offsets, indicating the location of stored informations from the grammar file. The wrapper class ShUG reads in the desired sections and by configuring the *tocReader* method it is easy to determine which sections to be visited or not. Further, it was mandatory for the following computation to extend ShUG by a method that reads in the direct subsumption relation among types. We have those information in the supertypes section of the binary file and the final supertype calculation is a simple look-up of the map between types and their supertypes. From there we can also easily infer a mapping from types to subtypes. The *symbol table* section delivers information about the mapping between an internal type identifier, an *int* and its symbol name in the grammar source, a *string*. The most essential section parts are the *hierarchy*—and the *constraints* section. Former section occupies the type encoding for efficient *glb* processing and the latter section instantiates the TFSs in their unexpanded form. The output file format is specified more in detail in A.1. The illustration of the methods in ShUG to translate the binary output in objects of Java is given in A.2.

4.3 Implementing the Lattice Operations

The structure and the information adhered to the image of the type hierarchy returned by the flop preprocessor are for a few reasons not appropriate for our scope. In this subsection, we will give a sketch of the encoding technique employed by flop and its impact on the lattice operations we need to compute and address the issues that are to be modified in terms of an efficient implementation of the default unifier. The type hierarchy produced by flop makes a distinction between three different sort of types, i. e., *proper types*, *leaf types* and *synthetic types*.

4.3.1 Proper Types

Our main interest lies on proper types that are located in areas of the type hierarchy featuring multiple inheritance property. A crucial difference concerning the implementational aspect between leaf types and proper types is that leaf types have the property not to be assigned to a bit-code while proper types features an encoded image. The applied encoding method is the transitive closure taken from (Aït-Kaci et al., 1989). Let $x_1, x_2, ..., x_n$ be elements of the hierarchy, in this case we consider proper types, then a transitive closure matrix is a two-dimensional array of 0's and 1's whose $(i, j)th$ element is 1 if and only if x_i is an ancestor of x_j. Hence for a proper type t_1 each bit-position is uniquely assigned to a proper type t_i in the hierarchy and embodies further the information if t_1 subsumes t_i.

4.3.2 Synthetic Types

Equally to the proper types, synthetic types have also bit-code map. Synthetic types are introduced by flop to maintain the BCPO condition that is imposed on the type hierarchy. A synthetic type is added if two different types have more than one unifier and they are not equal. That corresponds to the non-determinism issue and further the definitions for maximal lower bound and minimal upper bound in 2.

The BCPO condition requires for each pair of types to have by definition a unique greatest lower bound and a unique least upper bound. In the scenario of figure 4.5 the bit-*AND* operation between $\gamma(d)$ and $\gamma(f)$ does exactly yield the code of the synthetic type that will be added to the hierarchy.[1]

[1] $\gamma(t)$ is a function that maps a type identifier to its bit-encoding representation

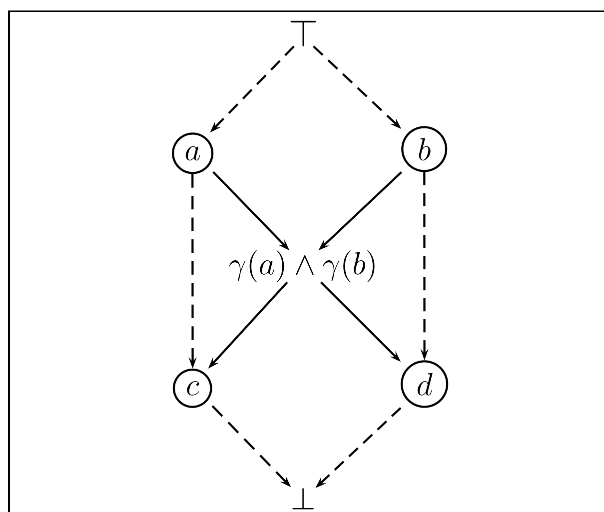

Figure 4.5: The emergence of synthetic types

The BCPO construction algorithm in flop takes as input the type hierarchy as specified by the grammar writer and the output is a graph representation of the embedding BCPO. The procedure is carried out in 3 main steps:

1. Calculate the $\gamma(t)$ for each type t, based on the AssignCode function from Ait-Kaci et al. (Aït-Kaci et al., 1989).

2. For each (ordered) pair of types (t_1, t_2), the outcome of the bit-wise and between t_1 and t_2 must have some correspondency with some existing code in the type hierarchy. If not a new type, a synthetic type is added. The check is to be repeated until no more new types have to be inserted.

3. The last step consists of the reordering of the subsumption relationships including the new synthetic types. For that purpose, it suffices to modify the subsumption relations of the parent nodes and the child nodes of the synthetic types.

4.3.3 Leaf Types

Callmeier's thesis defines leaf types as types that have no descendants and only a single immediate ancestor. However, the outcome of the different type hierarchies fed in flop reveals that also types with descendants are assigned to be leaf types. Actually flop referres leaf types to types with the property to have exactly one single ancestor and for all its subtypes the condition holds, that they have at most one supertype. To put it in another way, if a type is a leaf type then its substructure must be a tree and the parent node is the root node of the substructure. Omitting leaf types from encoding during the preprocessing of flop has a justification based on the motivation of the task to offer a unification based grammar development environment, where the needed lattice operation considers solely the *glb* calculation. In order to acquire gain in terms of efficiency, flop identifies leaf types given their property not to be necessarily encoded for efficient *glb* processing. Callmeier compares the processing time—provoked by the most time consuming task during preprocessing, the semi-lattice computation—between three different grammar frameworks realized in $\mathcal{TDL}$ a Japanese grammar, a German grammar and the LinGO grammar. As he stated, the significant difference in preprocessing time of the German grammar is attributed to the larger number of non-leaf types. Besides, the used encoding technique employs one bit per type in the hierarchy, when considering large-scale grammars we have a noteable saving in the storage of the hierarchie's encoded image.

In order to have the appropriate implementation of lattice operations introduced in the previous chapter, we have first to annul the embedding of the type hierarchy into the least structure following the BCPO condition that contains it. In particular we have to exploit the information delivered by the synthetic types and enable the lattice computations to generate multiple results. Synthetic types as such cannot make part of any lattice operations outcome. The *mlb* processing considers the calculation between two proper types, two leaf types and between proper types and leaf types. As for the calculation of *mub* we compute an additional encoding for all types.

4.3.4 *mlb* Calculation between Proper Types

As commented in the appendix A.2, the bit-code part in the flop output is represented as *Integers* determined in length by ending zeros. For the pur-

pose of the *glb* processing ShUG stores the bit-codes of each proper type as arrays of *ints*, where the *ints* with a higher position in the array are appended at the beginning of the code (little-endian order). The length of the array is therefore determined by the quotient of the number of proper types in the hierarchy and the bit-size of an *int* (32). The identification of the bit-code of each type is organized by a mapping between an identifier of a proper type or a synthetic type (an *int*) on the corresponding bit-code and vice versa. The efficient *glb* processing basically happens on the singular bit-wise *AND* operation between the int arrays of the type arguments. The resulting code maps uniquely to one proper type or one synthetic type. In the following the *glb* procedure of pairs of arguments, where bit-calculation was performed already, is reduced to a simple lookup in the hashtable, consisting of a mapping between an array-like encoding of pair of types ($type\ identifier(t_1) + number\ of\ types * type\ identifier(t_2)$) and the saved result, an *int Array*. This technique is adapted from the indexed memoization in (Michie, 1968) that basically incorporates the function to tabulate results of executed applications. Same approach we apply also later during the default unification process to save computations.

The occurrence of synthetic types in the type hierarchy benefits the conversion of the *glb* operation present in ShUG into a *mlb* operation. The detection of multiple results is supported by simply verifying if the resulting type is a synthetic type or not. If the resulting code after the bit-*AND* operation corresponds to a synthetic type, then the final result must be its descendants. Further it is relevant to check if we have among the direct descendants of a synthetic type other synthetic types located. In this case, the subtype query must be done iteratively until all synthetic types are replaced by their descendant proper types. Note that, if we are visiting subtypes or supertypes successively in a multiple inheritance environment, we have the case to probably visit some type more than once. Thus in this case we must sort out duplicate types and types in subsumption relation, whereas in the latter case the more general type is redundant. A point to remark is that only the case when we compute *mlb* between proper types, it is possible to have multiple results, since as mentioned already only types with a bit-code image are located in multiple inheritance areas of the hierarchy.

The two sample hierarchies given in the figure 4.6 show first the modifications executed by the preprocessing on the hierarchy developed by the grammar writer and secondly, the resulting consequences that must be con-

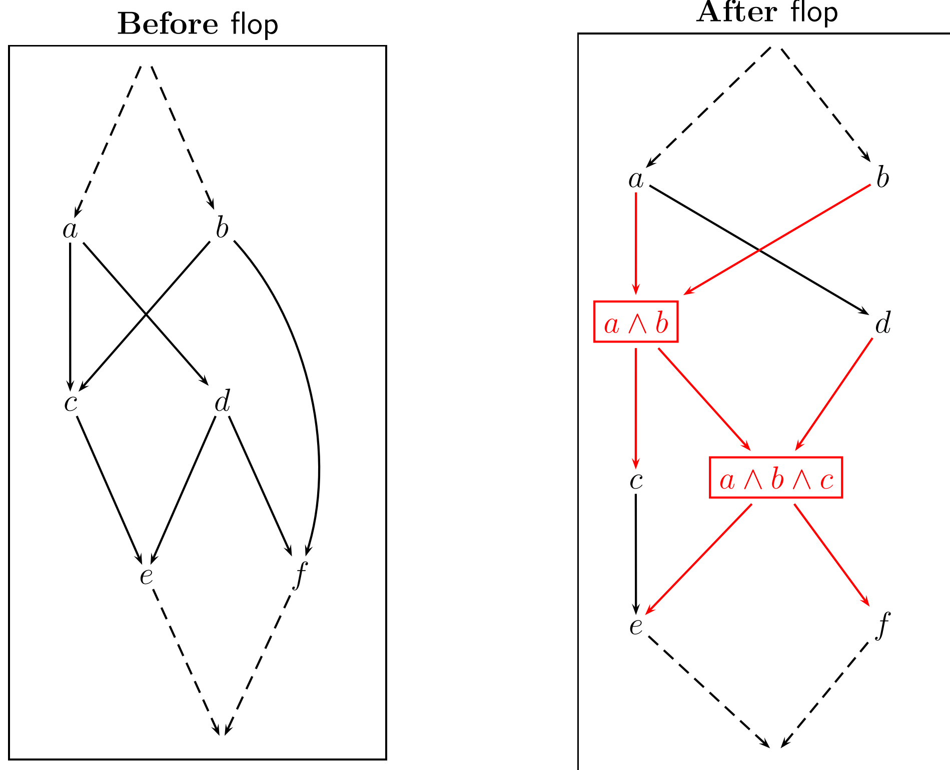

Figure 4.6: Handling multiple results in *mlb* processing

sidered when calculating the *mlb*. The original hierarchy is exposed on the left side, while the hierarchy with two added synthetic types is visualized on the right side. The red lines represents the modulated subsumption relations. The reason why two synthetic types are added is due to multiple joins between the types a and b, and between the types b and d. Having a look at the original hierarchy, it is not difficult to spot c and f as the *mlb* between a and b, while it appears to be more tricky to make out e and f to be the *mlb* of b and d. As for the *mlb* calculation between a and b referring to the hierarchy produced by flop, we must recursively identify the subtypes of the synthetic type $a \wedge b$, that corresponds to the inverse image of the resulting code of $\gamma(a)$ AND $\gamma(b)$ and we get after all the set $\{c, e, f\}$ as result. Since c subsumes e, e is not a relevant result and we have finally as result $\{c, f\}$.

4.3.5 *mlb* Calculation between Proper Types and Leaf Types

Given the property of leaf types, we observe two optional constellations between a proper type t_p and a leaf type t_l. Or t_p located in the hierarchy with multiple inheritance structures subsumes t_l or they are not comparable and thus incompatible. So for the *mlb* calculation between a leaf type and a proper type it is necessary to know if the proper type is among the types under the path that leads from the leaf type to the root node. Given the set of proper types and leaf types $T_p, T_l \in \mathsf{Type}$ we define $t \in T_p$ as:

Definition 20 Most specific proper type (MSPT)
For each leaf type $t_l \in T_l$, there is t the *most specific proper type*, $\Theta(t_l)$ such that $t \sqsubseteq t_l$, and for all proper types $t' \in T_p$ it applies $t \not\sqsubseteq t'$ with $t \neq t'$. □

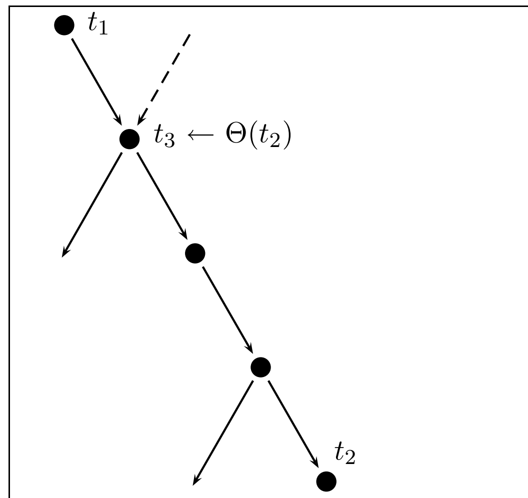

Figure 4.7: Finding *mlb* between a leaf type and a proper type

In the following we will use also the *minimal proper type* to denote the same definition. In practice, given a proper type t_p and leaf type t_l, and $\Theta(t_l)$ one must check if $t_p \sqsubseteq \Theta(t_l)$. If yes, the *mlb* is the leaf type itself, if not, there is no *mlb* viz. the result is *bottom* . Since the subsumption check is now to be executed between types with a bit-encoding, we can use the bit-*AND* operation on $\gamma(t_p)$ and $\gamma(\Theta(t_l))$. In particular the leaf type argument is not

subsumed by the proper type argument if the resulting code is not equal with the code of the minimal proper type of the leaf type.

The frequent use of the recursive search starting from the leaf type after the next proper type among the parents node is expensive. Hence, it makes sense during the construction of the type hierarchy to identify all most specific proper types and set a link between all the appropriate children (leaf types) and the corresponding most specific proper type. The look-up functionality from leaf types to their most specific proper types is an added feature to the system to have a more efficient *mlb* computation between proper types and leaf types. The initialization of the look-up table is done once when the hierarchy is built up in ShUG. In the figure 4.7, t_1 is the proper type argument and t_2 the leaf type argument. The MSPT of the leaf type t_2 is defined by the look-up function $\Theta(t_2)$ pointing to t_3. Since t_1 subsumes t_3 entailing t_1 subsumes t_2, we have at last t_2 as *mlb* between t_1 and t_2.

4.3.6 *mlb* Calculation between Leaf Types

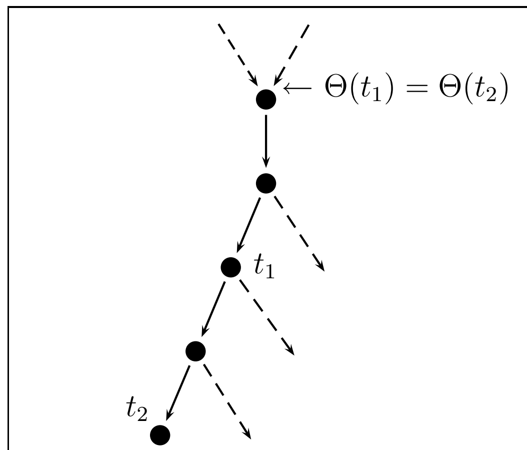

Figure 4.8: Finding *mlb* between a leaf type and a leaf type

Concerning the *mlb* processing between two leaf types, we differentiate considering the scenario if both types are located in the same tree structure and the scenario when they are not inside the same tree structure. The former issue implicates that both arguments have the same MSPT whereas

in the latter case the MSPT differs thus we can already conclude that the *mlb* processing calculates the *bottom* type as result. In case there are two leaf types as arguments located in a big tree, remember that all types in the tree are not assigned to a bit-code, we must collect for both arguments all their supertypes up to the MSPT. Further we would have to check for both if one argument is contained by the set of supertypes of the other argument and vice versa. So if t_2 is detected among the supertypes of t_1 then the *mlb* between t_1 and t_2 results in t_1. If there is no type intersection between the supertypes of both arguments, the leaf type arguments are not under the same path. Hence we have again *bottom* as the result. Accordingly in the figure 4.8 the *mlb* between t_1 and t_2 is determined by the fact that t_1 resides among the supertypes of t_2. Hence t_2 is the *mlb* for t_1 and t_2. This rather laborious *mlb* procedure we replace by a more efficient method that is introduced later on.

Next we establish a new bit-encoding for all types due to the necessity of a *mub* calculation and we will argue why the existing information in the encodings is not desirable.

4.3.7 *mub* Calculation

For the default unification it is relevant for the purpose of the assimilation process to feature either an efficient method that calculates the minimal upper bound of two types. Since the given framework was not aimed to support *lub* operation, this issue is a matter that is necessary to be wholly integrated to the system.

First I want to line out why the existing encoding is not an adequate foundation for a *mub* calculator. The figure in 4.9 shows a hierarchy that would demonstrates the insufficiency of the original encoding. The desired scenario is marked by the framed encoded types. By the way the type in the middle with the code 111 is a synthetic type, therefore it does not occupy a bit-position representing the own type in the hierarchy. Now if we want to compute $mub(g, f)$ one possibly refers to the duality between the bit-*AND* and the bit-*OR* operator and applies the latter to achieve as intend, the reversed effect of the *mlb* operation. However if we perform the bit-*OR* between code of the arguments $\gamma(g)$ and $\gamma(f)$, the resulting code would not

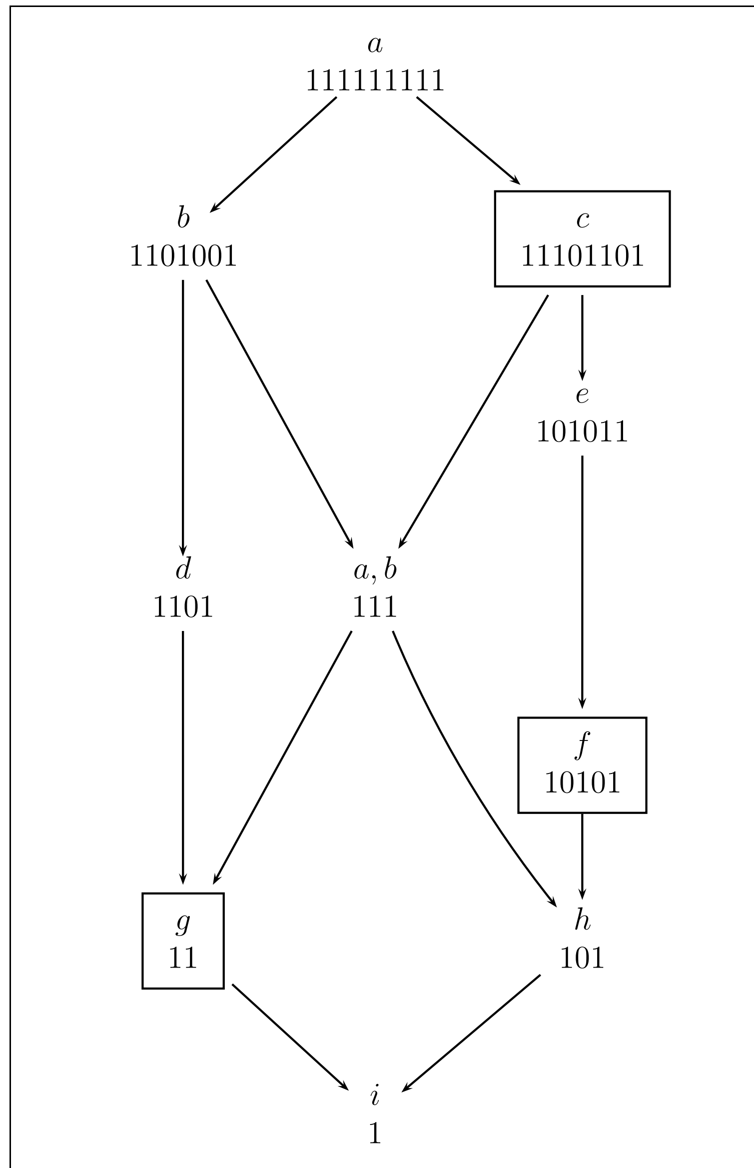

Figure 4.9: Failure of the original encoding when calculating *mub*. Note that missing bit-positions are equal to 0.

correspond to the code of the desired result. [2]

[2] γ^{-1} represents the mapping from a bit-code to the appropriate type identifier

$$11\ OR\ 10101 \ = \ 11101, \quad \gamma^{-1}(11101) \ = \ ?$$

The point is that all information—bit-positions from other types, that are inherited along the path from the type f with the code 10101 to the code of the desired result c are not incorporated in the resulting code. In order to get the type c that is represented by the code 11101101, it is inevitable to search for the most specific type with the encoding that subsumes the code of the calculated result. Technically, it means to request for all types the subsumption relation with the resulting code and to filter out among the types subsuming the result code, the most specific ones. Avoiding this laborious correction, it is reasonable to implement new type encodings designed for the *mub* operation.

Fundamentally the type encoding for the *mub* operation is also based upon the algorithm of Ait-Kaci. Hence the underlying algorithm that has been performed by the flop preprocessor is also valid here with the difference, that the order of the types to be encoded ought to be carry out reversely and the resulting code of some type is determined by the type encoding of its parents and not of its children.

Therefore an important matter to be investigated is to find out in which order the types are to be encoded. It must be guaranteed, if some type t is to be encoded, that there is no other type t' that subsumes t and t' has no code yet. Due to the hierarchy allowing multiple inheritance, determining the order by simply performing a depth first search over the children is not satisfying. For instance referring to the given hierarchy in the figure 4.9 the type h would be collected before its supertype f. This is resolved by the check if the currently visited type has already been visited. Hence we have to look if the collection of visited types in for the encoding appropriate order already contains it. If so it gets sorted out and added at the end of the collection.

The progress of the correct encoding order for the hierarchy in the figure 4.10 would look like this:

```
1.  (a)
2.  (a,b,c)
3.  (a,b,c,d,(a,b))
4.  (a,b,c,d,(a,b),g)
5.  (a,b,c,d,(a,b),g,i)
```

```
 6.  (a,b,c,d,(a,b),i,g,h)
 7.  (a,b,c,d,(a,b),g,h,i)
 8.  (a,b,c,d,g,h,i,(a,b),e)
 9.  (a,b,c,d,i,(a,b),e,g,h)
10.  (a,b,c,d,(a,b),e,g,h,i)
11.  (a,b,c,d,(a,b),e,g,h,i)
12.  (a,b,c,d,(a,b),e,g,h,i,f)
13.  (a,b,c,d,(a,b),e,g,i,f,h)
14.  (a,b,c,d,(a,b),e,g,f,h,i)
```

The first correction in the order happens in step 6, when the type g gets visited for the second time across the synthetic type as the parent node, where the first time it was visited over the type d as the parent node. Thus the type g gets shifted to the end of the order. Type h is either a subtype of the synthetic type, since it has not been already visited before it gets added to the very end of the order. Also in the next step a correction is executed, when type i is visited twice. This continues for every path that leads from each type to the *bottom* and at the end we have the order of types required for their correct encoding in step 14.

In the original version of the encoding the first type assigned to a code is the *bottom*. Now we start the encoding at the *top* type. According to Ait-Kaci's AssignCode, each type is a result of:

$$\gamma(t) \;=\; 2^P \vee \bigvee_{i=1}^{n} \gamma(x_i) \quad , \qquad parents(t) = x_i, ..., x_n$$

Since the first type to be encoded the *top* has no parents, its type is simply assigned to 2^P, where P is zero. The number P gets raised by one after each type encoding. After assigning *top* to a bit-code, we have when encoding the next type, $P = 1$. Hence 2^P generates the bit-position that is reserved to the encoded type and with each encoded type excluding synthetic types the bit-length increases.

The procedure of the encoding of types given the just determined order is as follows:

$$
\begin{aligned}
\gamma(a) &:= 2^0 && = 1\\
\gamma(b) &:= 2^1 \vee \gamma(a) && = 01 \vee 1 = 11\\
\gamma(c) &:= 2^2 \vee \gamma(a) && = 001 \vee 1 = 101\\
\gamma(d) &:= 2^3 \vee \gamma(b) && = 0001 \vee 11 = 1101\\
\gamma(a,b) &:= \gamma(b) \vee \gamma(c) && = 11 \vee 101 = 111\\
\gamma(e) &:= 2^4 \vee \gamma(c) && = 00001 \vee 101 = 10101\\
\gamma(g) &:= 2^5 \vee \gamma(d) \vee \gamma(a,b) && = 000001 \vee 1101 \vee 111 = 111101\\
\gamma(f) &:= 2^6 \vee \gamma(e) && = 0000001 \vee 10101 = 1010101\\
\gamma(h) &:= 2^7 \vee \gamma(a,b) \vee \gamma(f) && = 00000001 \vee 111 \vee 1010101 = 11101011\\
\gamma(i) &:= 2^8 \vee \gamma(g) \vee \gamma(h) && = 000000001 \vee 111101 \vee 11101011 = 111111111
\end{aligned}
$$

The desired encoding to calculate *mub* concerning the sample hierarchy in the figure 4.9 is showed in the figure 4.10, where the bit-*AND* operator provides the unique code of c taking the codes of g and f as arguments. The treatment of synthetic types happens in a dual manner to the one during the *mlb* operation.

4.3.8 *mlb* Calculation between Leaf Types Revised

A consequence having also a mapping of the leaf types to a bit-code is that we can replace the method in ShUG that employs a search over the supertypes for both leaf types arguments. Instead the bit-*AND* operator, the bit-*OR* operator applied to the bit-encodings generated for *mub* processing provides the correct result when identifying the *mlb* between two leaf types. Due to the tree structure where leaf types are located, no inconsistency occurs when taking the same bit-codes that are generated for the *mub* operation. As we have seen the *mlb* calculation of two leaf types is reduced to the subsumption check between the types. The subsumption relation is confirmed if the resulting code of the bit-*AND* operation is equal to one of the arguments. In the type encoding for *mub*, the information of the subsumption relation to all respective supertypes is adhered to each type. Hence in case subsumption

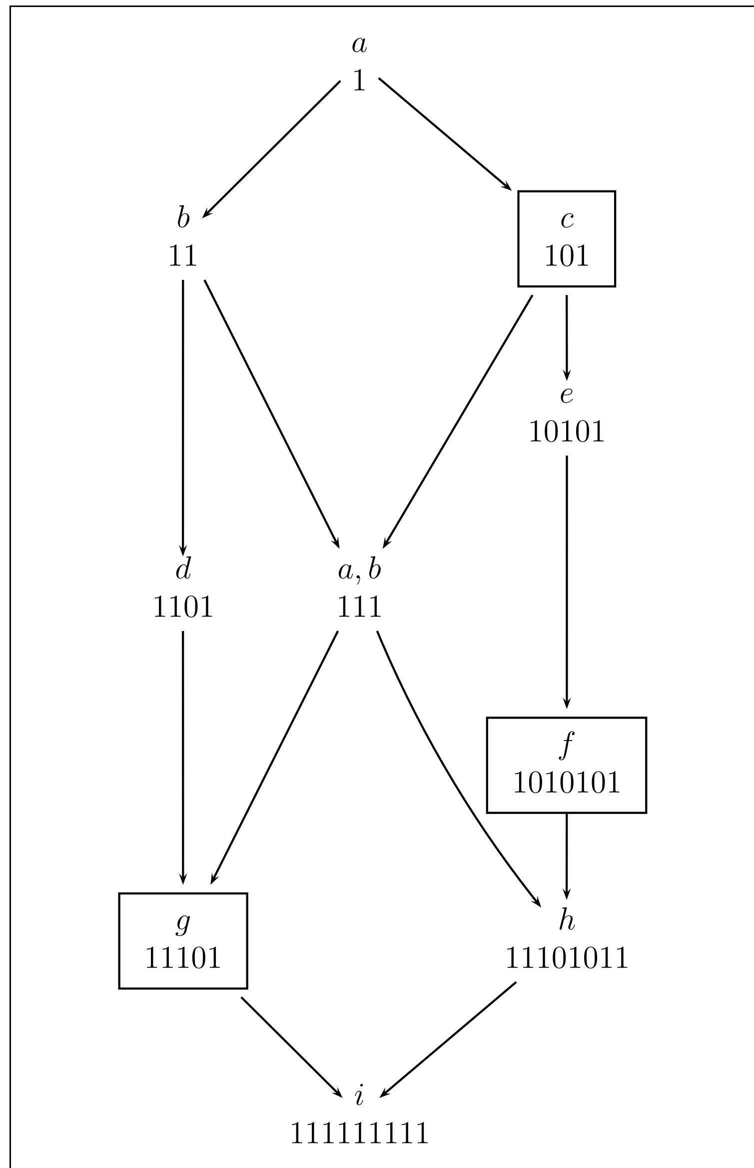

Figure 4.10: Type encoding for *mub* processing.

check succeeds one type incorporates all the supertype relation featured by the other type.

4.3.9 Grouping the Hierarchy

We realize that there are compression methods to lower the increase in the bit-length during the bit-encoding of the types. To get around occupying one digit for each type, different approaches have been suggested, one of them is dividing the hierarchy into code blocks.

In this modified realization of the modulation technique in (Aït-Kaci et al., 1989), rather than to encode all types top-down and thereby to increase with each encoded type the bit-length, the goal is to divide the hierarchy into subgroups and for each subgroup the encoding of types starts from the most specific type in the subgroup. Each subgroup is uniquely identified by a number that is embodied by a defined part of the bit-code. Therefore the

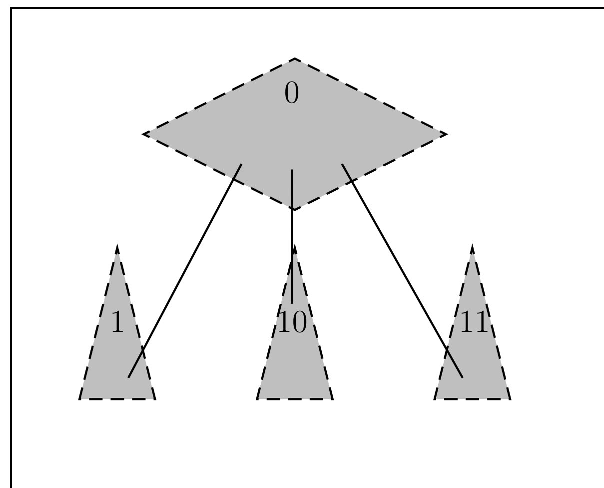

Figure 4.11: The encoding into groups

bit-code consists of a part that identifies the subgroup and a remaining part that is reserved for the encodings of types in the respective subgroup. The proposal here regarding the definition of subgroups in the hierarchy is given by the distinction between leaf types and proper types. In general the output of flop breaks down some type hierarchy into a structure similar to the sample exemplified in the figure 4.11. Some graph structure containing the proper types is located at the *top* of the hierarchy with some tree structures containing leaf types that are attached somewhere at the fringe of the graph. Those types that connects the graph with the tree structures are minimal

proper types.
So in practice the number of minimal proper types in a hierarchy determines the length of the bit-code that is to be added to the codes of all types in the hierarchy. By enumerating over the minimal proper types we obtain an identifier that is to be converted to the corresponding bit-representation. This part of the code identifying to which subgroup the type is assigned to, occupies the first bit-positions of the full bit-code. Applied to the schematic hierarchy in the figure 4.11, we have four subgroups with the additional code in the respective structure. At the end we have reduced the bit-length of the bit-code to the number of types in the subgroup containing the most types plus the bit-length of the bit-representation of the number of minimal proper types in the type hierarchy.

Instead of *int* Arrays, we use objects of the BitSet class in Java in the implementation of the type encoding. Accordingly we can skip to manage the arrays length linked to the number of types to be encoded.

Chapter 5

An Implementation of Well-formed Default Unification

The following chapter discusses in detail the setup of the different implements in order to run and experiment with the default unifier. The first section describes the realization of the type preprocessing and points out the motivation of the extensive realization of the lattice operations.

The realization of the approach in the theoretical analysis regarding the preprocessing on types in the previous chapter is embodied by the notion of the Delta Iterator. In the following we give two different options of the search of abstract background types. The first option sticks to the constraints of the theoretical analysis. As for the second alternative we return to put the focus on synthetic types and show how they can contribute to identify directly type configurations involved in an area of non-determinism. We mention that the implemented type hierarchy viewable in the appendix B is designed to experiment with the observations done considering the multiple inheritance property. In this context the behaviour of the assimilation operating with leaf types is not relevant. Leaf types are involved in a hierarchy structure that corresponds to trees. The assimilation procedure in trees is a trivial variant of the required treatment in multiple inheritance hierarchies, wherein ambiguity would not arise during the default unification.

The second section gives insight into the structural design of the overall implementation. The main functionalities of components such as the unifier for the specialization and the default unifier are illustrated by means of

pseudo code excerpts. Lastly an entry for a concrete complex scenario of the default unification referring to the hierarchy in B is given with the detailed demonstration of the single procedural steps.

5.1 Precomputation on Types

Carpenter's definition to consider all possible subsets of the shared structures in background and cover that have most specific unifications with the cover has been analogously implemented in ALE (Grover et al., 1994). The effect of the assimilation process however is exactly the same in a sense that all valid type definitions unifiable with the cover, that are closest in the hierarchy to the structure of the "non-existing" unifier are to be identified. This happens recursively from the top most level of the cover up to the point when a feature points to an atomic type value. Exploiting the expressiveness of the type hierarchy it is possible to classify intermediate structures and results to types and type definitions throughout the computation.

In the developed system the task of the assimilation is as well divided by one module that undertakes the task in the assimilation dealing only with types, the types preprocessing themselves and one other module that monitors the types preprocessing and takes over the *specialization* and *generalization*. A description of the realization of the former module is given in this section. The types preprocessing in the assimilation process represent the kernel in the implementation of the default unification algorithm. In the previous chapter we have made relevant observations regarding the intricacy of multiple inheritance hierarchies enabling the translation of the assimilation algorithm into a real application. We know the types preprocessing during the default unification process takes over the work that can be mastered without expanding types to TFSs. Therefore it stands for the component where the efficiency of the default unification algorithm rests upon, since "weighty" processings with large memory consumption are avoided. The precomputation with types requires to browse abundantly through the type hierarchy. For each recursive call of the default unification it is necessary to start one assimilation process. The assimilation itselves implies one *mub* query and theoretically a number of *mlb* queries concordant to the number of all supertypes of the background that do not subsume the *mub*. Hence a valuable prerequisite is to resort to an efficient implementation of the needed lattice operations via bit-encoding.

5.1.1 Delta Iterator

```
PROCEDURE update()
 1  WHILE supertypes.isNotEmpty and deltas.isEmpty
 2    supertypes' = supertypes.copy
 3    supertypes.clear()
 4    FOR all supertype in supertypes'
 5      FOR all supertype' in supertype.getsupertypes()
 6          and mub in mub(covertype, backgroundtype)
 7        IF supertype'.isSynthetic nor supertype'.isVisited
 8           nor supertype'.subsumes(mub) THEN
 9          supertypes.add(supertype')
10        ENDIF
11      ENDFOR
12    ENDFOR
13    ...
14    FOR all supertype in supertypes
15      IF mlb(covertype, supertype).isNotEmpty THEN
16        FOR all mlb in mlb(covertype, supertype)
17          deltas.add(cover, supertype, mlb)
18    ...
19  ENDWHILE
```

Figure 5.1: Given the Delta queue is empty the expansion to the next higher layers in the hierarchy happens as long expansion is possible, see line 7, 8 and none among background supertypes has a *mlb* with the cover type, see line 15.

The current implementation of the type preprocessing is designed to examine each level in the hierarchy starting at the level where the background is situated and then proceed checking succesively the higher levels. Rather the approach that would collect all the type configurations in question by doing the search in one go, we suggest to implement an iterator that enables to explore the next upper level in the hierarchy on demand. Basically the search for the abstract background types is supervised by identifying the next layer consisting of background's supertypes that fulfil some defined constraint. In particular the layer by layer exploration is stopped if an appropriate supertype on any of the paths emanate from the background type has been detected, that features a *mlb* with the type of cover. As a result it

would be easy to impose additional conditions on the types and to discharge priorly default unification process for certain type configurations. Further we have a clear separation within the default unification process between the computation dealing with types only and the calculation involving the internal structure.

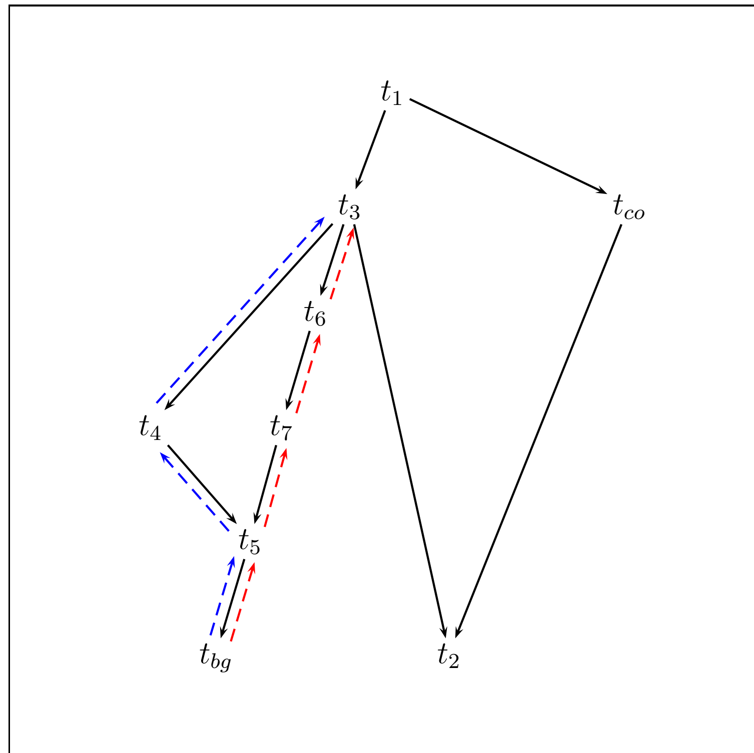

Figure 5.2: Sorting out already visited background candidates. The blue lines show the path $\langle t_{bg}/t_5/t_4/t_3\rangle$ and the red lines display the movement on the path $\langle t_{bg}/t_5/t_7/t_6/t_3\rangle$.

We call the component in the implementation representing the type preprocessing the *Delta Iterator*. A *Delta* corresponds to the concept of type configurations with the trivial difference, that Deltas are triple containing also the type of the cover. The queue containing Deltas is updated (see figure 5.4) each time after the query if there is a next Delta in the queue. Though as long as the queue with Deltas is not empty, the iterator just outputs on

demand the next Delta in the queue. During the update the expansion to the next layer of valid supertypes is not executed until all Deltas obtained by some layer have already been supplied.

Given some hierarchy, the Delta Iterator would not hand over anymore Deltas, when the expansion to a higher level does not return any valid supertype. We have three conditions to sort out supertypes, declaring that an appropriate supertype may not be represented by:

1. A type subsuming any of the *mub* between cover and background.

2. Synthetic types.

3. A type that is already visited in a previous stage of type preprocessing.

We haven't seen the condition under point 3 yet, that is actually a useful constraint we can impose on a early level of the default unification in order to save waste processings. It is a simple and effective constraint to rule out supertypes of the background type that have already been visited. In the figure 5.2, when we proceed to collect the background supertype layer by layer the type t_3 would be visited twice. During the search t_3 is an appropriate supertype of two different paths in the fourth and fifth layer. Note that the search considers also the level of the background in the search space, since the background can be in itself an abstract background structure.

5.1.2 Detection of Deltas due to Non-Determinism

Depending on the hierarchy theoretically some path including possible abstract background types could converge arbitrarily near the *top* before some abstract background type candidate gets located. Hence the tracking of some path all the way up may turn out to be futile, after having checked innumerous types. In particular, if we are pursuing a path that has no *mub* with the cover which per default will come across the *top* then in the worst case we may reach the *top* without crossing any type having an *mlb* with the cover type.

According to the analysis how multiple results arise from type preprocessing there are two alternatives caused by non-determinism in the hierarchy structure:

Option 1 Given a valid type configurations $t_{conf} = \langle t_{bg'}, t_{co'} \rangle$ and there is $t_{mub} \in \sqcap_t(t_{bg}, t_{co})$, $t_{mub} \neq \top$ such that $t_{mub} \sqsubseteq t_{bg'}$. For each type configuration $t'_{conf} = \langle t_{bg''}, t_{co''} \rangle$ such that $t_{bg''} \sqsubseteq t_{bg'}$ and $t_{co'} \not\sqsubseteq t_{co''}$, $t_{co'} \not\sqsupseteq t_{co''}$ we know that t'_{conf} yields a result that is forwarded by non-determinism.

Option 2 Given a valid type configurations $t_{conf} = \langle t_{bg'}, t_{co'} \rangle$ and there is no $t_{mub} \in \sqcap_t(t_{bg}, t_{co})$ such that $t_{mub} \sqsubseteq t_{bg'}$. For each type configuration $t'_{conf} = \langle t_{bg''}, t_{co''} \rangle$ such that $t_{bg''} \sqsubseteq t_{bg'}$ and $t_{co'} \not\sqsubseteq t_{co''}$, $t_{co'} \sqsupseteq t_{co''}$ we know that t'_{conf} and t_{conf} yield a result that is forwarded by non-determinism.

Since we have a conversion of the type hierarchy into a BCPO, we propose a strategy to locate directly valid type configurations that are evoked by the non-determinism. The synthetic types produced by flop are embedded into the hierarchy in order to eliminate the non-determinism concerning the joins and the meets of two types. Hence it is feasible to entail non-determinism if a synthetic type has been added to the hierarchy. Further all added synthetic types can be easily identified and isolated during the construction of the type hierarchy.

Now we want to define the property of those synthetic types that play a role in the two options when multiple results during default unification arise because of the non-determinism. In the figure 5.3 on the left the type configurations $\langle t_6, t_3 \rangle$ and $\langle t_2, t_4 \rangle$ yield both valid results. The subsumption ordering between abstract background type and abstract cover type of the latter has been modified by flop. Since t_2 and t_{co} have no unique lower bound a synthetic type has been inserted between t_2, t_{co} and their *mlb*, $\{t_3, t_4\}$. Hence the synthetic type has the property to be in direct subsumption relation with abstract background type and the abstract cover type. The situation on the right side concerning the second option, where the corresponding abstract background type is not subsumed by the *mub* between the types of cover and background can be described analogously.

Instead of identifying valid type configurations by climbing up the hierarchy and to check for the background's supertypes if they are abstract background types, we now need to look after the synthetic types that fulfil the following specification of type configurations that are generated by the non-determinism:

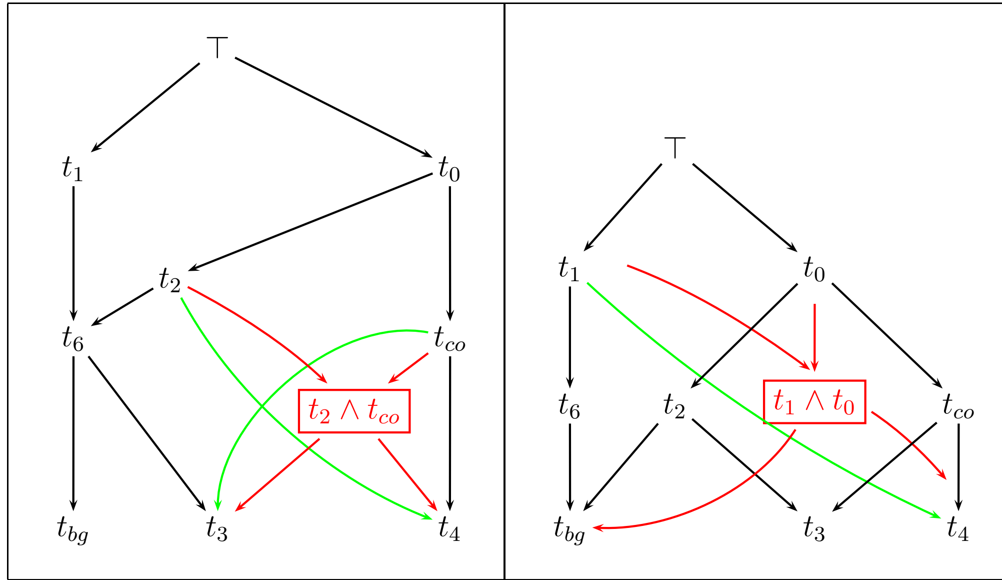

Figure 5.3: On the right side the synthetic type is placed so that we have a unique lower bound between t_2 and the type of the cover. The red arcs substitute the original direct subsumption relation represented by the green arc between t_{co}, t_3 and between t_2 and t_4 (Option 1). On the left side the synthetic type is placed so that we have a unique lower bound between t_0 and t_1. The red arcs substitute the original direct subsumption relation represented by the green arc between t_1 and t_4 (Option 2).

Proposition 5

Given some $t_{bg'} \sqsubseteq t_{bg}$ and some $t_{co} \sqsubseteq t_{co'}$, for each $t'_{syn} \in T_{syn}$ and $t_{bg'} \sqsubseteq t'_{syn}, t'_{syn} \sqsubseteq t'_{co'}$ it follows that $\langle t_{bg'}, t_{co'} \rangle$ is a valid type configuration.

The adequate update procedure likely needs to iterate less times to deliver all valid Deltas in the hierarchy in comparison to the firstly introduced function. Note that the type configurations, that are not involved in non-determinism we still identify by the search after abstract background types. Though it is not necessary after having found an abstract background type to keep on tracking the associated path. Further paths that are not under a *mub* can be omitted as well, since all type configurations assigned from detected abstract background types located on such paths are subject to non-determinism. Actually the first step during the search in the alternative

```
PROCEDURE update()
 1  WHILE supertypes.isNotEmpty and deltas.isEmpty
 2    supertypes' = supertypes.copy
 3    supertypes.clear()
 4    FOR all supertype in supertypes'
 5      FOR all supertype' in supertype.getSupertypes()
 6          and mub in mub(covertype, backgroundtype)
 7        IF supertype'.isSynthetic nor supertype'.isVisited
 8           and mub.subsumes(supertype') THEN
 9          supertypes.add(supertype')
10        ENDIF
11      ENDFOR
12    ENDFOR
13  ...
14    FOR all supertype in supertypes
15      IF mlb(covertype, supertype).isNotEmpty THEN
16        FOR all mlb in mlb(covertype, supertype)
17          deltas.add(new Delta(cover, supertype, mlb))
18        ENDFOR
19  // corresponding Delta results in a successful specialization...
20        supertypes.remove(supertype)
21       ...
22  ENDWHILE
23  ...
24  WHILE deltas.isEmpty
25    synthetictype =  synthetictypes.next()
26    supertype' = synthetictype.getSupertype().getElement(0)
27    subtype' = synthetictype.getSubtype().getElement(0)
28    IF supertype'.subsumes(backgroundtype)
29       and subtype'.subsumes(covertype) THEN
30      deltas.add(cover, supertype', subtype')
31  ...
32  ENDWHILE
```

Figure 5.4: The expansion of layers depends on the fact if the supertype is still on the path to the *mub*, see line 8. If it is the case that the supertype yields a Delta, the background supertype gets removed in line 20. Though before discarding a path it is crucial to check if the specialization succeeds. Other Deltas are identified by the iteration over the synthetic types in 24-30.

update function corresponds to the constraint on background types when dealing with BCPO hierarchies as described in (Alexandersson & Becker, 2007).

When comparing the two introduced update procedures, we establish that the effect of both approaches results into the same set of type configurations. The benefit of one or the other procedure depends on the structure of the hierarchy. Certainly it is reasonable to use the second version, if the hierarchy performs sparsely non-determinism. Even more redundant lattice operations would be executed by the first version in a BCPO hierarchy. Though, given a shallow hierarchy that is highly non-deterministic, then probably the first version needs less iteration to return the next Delta. Some prior knowledge about the structure of the hierarchy therefore can help to make decision on the proper constraints for the background supertypes.

Whatever option we want to consider, both approaches embody the notion to have an ongoing interaction between the actual default unification with the type preprocessing. The implementation permits to extend, and to modify easily the conditions on the types to be default unified. In particular the constraints imposed on the supertypes of the background to be abstract background types are considered to be an exchangeable part in the implementation. The technique to process the next Delta only by the explicit instruction is also a good starting position to alternate between constraints even during the processing. Further, it is possible to monitor the Delta iterator by some extensions to the system, where constraints for the search after type configurations may be imposed from outside the Delta iterator. Besides the adhered stop mechanism of the expansion to the next layer of supertypes, the termination of the type preprocessing can be terminated by an external device as well.

5.2 AVM related Functions

In the previous section the fundamental device as a precomputational step in the default unification has been discussed in great detail. The realization of the type preprocessing as an iterator to calculate the next type configurations on demand turns out to be useful as an interactive tool allowing to communicate with other engines. The part of the implementation that monitors the entire default unification process including the precomputation on types represents the matter in this section.

5.2.1 The Design of TFS

One challenging task was to find a solution to cope with all the generated structures due to the potential multiple results emerging from the type preprocessing and also from the specialization process. A reasonable measure in order to master the possibly rampant production of results during the default unification is specifying a compact format of TFS object that is able to express ambiguity. In the presented algorithm, features are assigned to the overlay of possibly multiple outcome of the assimilation procedure. In order to realize the option that features are allowed to point in addition to a single structure, to a disjunction of TFSs as well, we extend the realization of the TFSs given by the $\mathcal{TDL}$ API.

$$t_1 \begin{bmatrix} A : \left\{ \begin{array}{l} t_2 \begin{bmatrix} \text{C} : t \end{bmatrix} \\ t_3 \begin{bmatrix} \text{D} : f \end{bmatrix} \end{array} \right\} \\ B : \left\{ \begin{array}{l} t_4 \begin{bmatrix} \text{E} : \mathit{bool} \end{bmatrix} \\ t_5 \begin{bmatrix} \text{F} : \mathit{bool} \end{bmatrix} \end{array} \right\} \\ C : t_6 \begin{bmatrix} \text{G} : f \end{bmatrix} \end{bmatrix} = \left\{ \begin{array}{l} t_1 \begin{bmatrix} \text{A} : t_2 \begin{bmatrix} \text{C} : t \end{bmatrix} \\ \text{B} : t_4 \begin{bmatrix} \text{E} : \mathit{bool} \end{bmatrix} \\ \text{C} : t_6 \begin{bmatrix} \text{G} : f \end{bmatrix} \end{bmatrix}, t_1 \begin{bmatrix} \text{A} : t_3 \begin{bmatrix} \text{D} : f \end{bmatrix} \\ \text{B} : t_4 \begin{bmatrix} \text{E} : \mathit{bool} \end{bmatrix} \\ \text{C} : t_6 \begin{bmatrix} \text{G} : f \end{bmatrix} \end{bmatrix}, \\ t_1 \begin{bmatrix} \text{A} : t_2 \begin{bmatrix} \text{C} : t \end{bmatrix} \\ \text{B} : t_5 \begin{bmatrix} \text{F} : \mathit{bool} \end{bmatrix} \\ \text{C} : t_6 \begin{bmatrix} \text{G} : f \end{bmatrix} \end{bmatrix}, t_1 \begin{bmatrix} \text{A} : t_3 \begin{bmatrix} \text{D} : f \end{bmatrix} \\ \text{B} : t_5 \begin{bmatrix} \text{F} : \mathit{bool} \end{bmatrix} \\ \text{C} : t_6 \begin{bmatrix} \text{G} : f \end{bmatrix} \end{bmatrix} \end{array} \right\}$$

Figure 5.5: Equivalence relation between an ambiguous TFS and the resolution to all its interpretations.

```
public class TFS extends FeatureStructure {
   protected ArrayList<FeatureValueList> units;
   protected ArrayList<FeatureValuePair> elements;
   protected Typehierarchy typeHierarchy =
                           (Typehierarchy)this.grammar;
   ... }

public class FeatureValuePair {
   private short feature;
   private TFS value;
   ... }


public class FeatureValueList  {
   private short feature;
   private ArrayList<TFS> value;
   ... }
```

Figure 5.6: A TFS Object consists of a list containing Feature Value Pair Objects and a list containing Feature Value List Objects

A representation of an ambiguous TFS we interpret as an attribute value matrix (AVM), with a finite set of features that are assigned to either a TFS or a set of TFSs. Considering the atomic structures (Carpenter, 1993) of a TFS, the equation in the figure 5.5 reveals that the number of atomic structures including all resolved structures may grow exponentially as to the number of atomic structures in the ambiguous case. The implementation of the default unification is designed to be able to deal with ambiguous structure, i. e., a resulting ambiguous TFS developed from specialization may get an argument for a recursive call of the default unifier. For representational concerns the implementation allows however the conversion of an ambiguous structure to all resolved structures as well. In the simplified fragment of code 5.6 the TFS object differs between attribute value pairs, where in one case the value represents a single TFS object and in the other case we have the value as a list of TFS object. A TFS object is linked to a type hierarchy object that contains crucial information like a fixed finite set of features and a function that maps from a type identifier to the corresponding prototype structure. Type definitions are TFS objects, where embedded TFS objects

assigned by some feature f are expanded in case:

- f is firstly introduced and points to a TFS object, that is not equal with its type definition.
- f is inherited by some more general type definition and points to a TFS object more specific than the adequate TFS object of the direct supertype.

Unlike $\mathcal{TDL}$, the methods responsible for the construction of a TFS object do always check the well-formedness condition referring to the given type hierarchy. An appropriate warning will be send out if the structure object does not meet the requirements of its type definition. Analogously an ambiguous TFS object is required to fulfil the well-formedness conditions for all its interpretations.

According to the expressiveness of ambiguous TFS structures, it is required to adapt basic functions operating on TFSs such as copying methods, the subsumption check on two TFS objects or the test if two TFS objects are equally structured. Moreover the required extensions to the implementation we apply to the unification procedure.

5.2.2 The Unifier Procedures

A new implementation of the common unification operation is necessary, since the unifier provided by $\mathcal{TDL}$ would not consider having multiple results during the unification. The unification in $\mathcal{TDL}$ is based upon the calculation of the greatest lower bound, where synthetic types due to the open world assumption belong to the domain of feasible unifiers. Other consequence of the open world assumption is that the resulting TFS must not satisfy its most general satisfier implying that well-formedness condition on TFS objects is not assured. In the theoretical analysis the definition of well-formed unification says that unification $\mathcal{A}$ must not only succeed between the arguments. The unification is not successful before $\mathcal{A}$ does not fail to unify as well with the type definition of the type of $\mathcal{A}$. In the following we present extractions out of the implementation in a pseudo code manner realizing the well-formed unifier on TFS objects. In the header of the procedure we have the listing of employed acronyms and overall variables.

Line 4 checks if the *mlb* between the types of the arguments yields the *bottom*. If yes the empty list will be returned. The overall unification procedure will throw an exception if the list will remain empty for each unification

```
fvp := FeatureValuePair
fvl := FeatureValueList
th = TypeHierarchy
PROCEDURE unify1(tfs2)
 1   unifications = new Collection<TFS>
 2   mlbs = th.mlb(this.getType(), tfs2.getType())
 3   IF mlbs.size() is 1 and mlbs.get() is bottom THEN
 4     return unifications //empty list cause unification to fail
 5   ENDIF
 6   IF this.getFvps().isEmpty() and this.getFvls().isEmpty()
 7      and tfs.getFvps().isEmpty() and tfs.getFvls().isEmpty()
 8     FOR all mlb in mlbs
 9       unifications.add(new TFS(mlb,th))
10     ENDFOR
11     return unifications
12   ENDIF
13   FOR all mlb in mlbs
14     tfs = unify2(mlb,tfs)
15     IF tfs is not NULL
16       unifications.add(tfs);
17     ENDIF
18   ENDFOR
19   return unifications
```

assigned to any feature. If both arguments do not have any structure inside their body it is the case that either the structures are atomic or they are equal to their type definitions. For both cases it is sufficient to instantiate TFS objects embodying the types of *mlb* while referring to the appropriate type hierarchy and return those structure as result of the unification. Thus the recursive procedure of unification will end up to meet the condition in line 6 and 7. Further the interception of computing unification between TFS objects performing the latter property corresponds to the partial expansion technique applied by $\mathcal{TDL}$ and flop. Besides the reduction at runtime, the size of the type hierarchy including type definitions is reduced as well. On the expanded level of a TFS object unifications are absorb by line 14, whose arguments are propagated to a helper procedure `unify2`.

Similar to the minimal fixed arity in (Callmeier, 2001) when two TFS objects tfs_1 and tfs_2 with different feature sets are to be unified, the features get categorized into features shared between both arguments and features

```
PROCEDURE unify2(mlb,tfs2)
 1   elements = new Collection<FeatureValuePair>
 2   units =  new Collection<FeatureValueList>
 3   proto_mlb = th.getMGSat(mlb)
 4   feat0 = //features that occurred only in the mgsat of mlb
 5   feat1 = //features that occurred in this and not in tfs2
 6   feat2 = //features that occurred in this and not in tfs2
 7   feat1_2 = //features that occurred in this and in arg2
 8   FOR all feat in feat0
 9     elements.add(proto_mlb.getFvp(feat).copy())
10   ENDFOR
11   FOR all feat in feat1 //...compare to line 13
12   FOR all feat in feat2 //...compare to line 13
13   FOR all feat in feat1_2
14     unifications' = new Collection<TFS>
15     proto_feat_ = proto_mlb.getFeatureValue(feat)
```

occurring only in one of the argument. Before the recursive call on the arguments of the internal structure, if we unify tfs_1 and tfs_2 with feature sets $feats_1$ and $feats_2$, the iteration basically happens over all positions valid in tfs_1. The corresponding feature in tfs_2 is identified by a table lookup within the internal structure, the list of Feature Value Pair and Feature Value List objects. Note that the well-formed unification must also consider features that occur only in the prototype structure of the resulting type. The specification of the proceeding with features that does not occur in both arguments in line 11 and 12 can be inferred from the shared case in line 13.

In the next step we make a distinction of every feature if it points to a single TFS or to an ambiguous TFS object. Given some shared feature f in tfs_1 and tfs_2, we have to consider the following four options:

1. f points to a single TFS object in tfs_1 and tfs_2.

2. f points to a single TFS in tfs_1 and ambiguous TFS object in tfs_2.

3. f points to a ambiguous TFS in tfs_1 and single TFS object in tfs_2.

4. f points to a ambiguous TFS object in tfs_1 and tfs_2

The treatment of the cases indicated by option under one and three we show in the piece of pseudo code above. The proceedings of the other cases

```
16     IF this.getFvps().contains(feat) and tfs2.getFvps().contains(feat)
17       this_feat_ = this.getFeatureValue(feat)
18       tfs2_feat_ = tfs2.getFeatureValue(feat)
19       tfs' = this_feat_.unify1(tfs2_feat_)
20       IF tfsList'.isEmpty() THEN
21         return NULL
22       ENDIF
23       FOR all tfs' in tfsList'
24        unifications'.addAll(proto_feat_.unify1(tfs'))
25       ENDFOR
26       IF unifications'.isEmpty() THEN
27         return NULL
28       ENDIF
29     ENDIF //...increment the internal structure
30     IF this.getFvps().contains(feat)
31        and tfs2.getFvls().contains(feat) THEN
32       this_feat_ = this.getFeatureValue(feat)
33       tfs2_feat_ = tfs2.getFeatureValues(feat)
34       FOR all tfs2_feat_' in tfs2_feat_
35         tfsList' = this_feat_.unify1(tfs_feat_')
36         FOR all tfs' in tfsList'
37           unifications'.addAll(proto_feat_.unify1(tfs'))
38         ENDFOR
39       ENDFOR
40       IF unifications'.isEmpty() THEN
41         return NULL
42       ENDIF
43     ENDIF //...increment the internal structure
```

can be analogously inferred. If any of the internal unifications at some feature yield an empty list, the computation is interrupted in line 20-22 and causes an empty list of TFS objects to be returned. The same behaviour we have concerning the unifications with the structure of the MGSat as well in line 26-28. Whereas when dealing with features associated to multiple TFS, we consider a unification to be still successful even if not all unifications between the respective arguments have succeeded. We assume that it is sufficient to meet the condition in line 40 saying that at least one unification did not fail. When unification at some feature is successful then the outcome will be assigned to the feature.

Given that unification has not been interrupted, the internal structure

```
44 //incrementation of the internal structure
45    IF unifications.size() is 1 THEN
46      elements.add(new FeatureValuePair(feat,
47                                    unifications'.get()))
48    ELSE units.add(new FeatureValueList(feat, valNew))
49    ENDIF
50 //...after iteration over all features
51    return new TFS(mlb,elements,units,th);
```

for the resulting TFS gets incremented by the successful unification at the given feature. The unifications are employed to update either the list of feature value pair implying that unification yield a single TFS object or the list of feature value list with the appropriate feature pointing to a non-deterministic unification. Finally after the iteration over all features, the completed internal structure, the type of *mlb* and the type hierarchy produces the new TFS object, that belongs to the collection of TFS object introduced in `unify1`.

The Default Unifier

The design of the default unifier implementation resembles in some part the unifier processing. Though the values of the internal structure during unification are solely defined by the *mlb* processing whereas in the default unification two components, the assimilation process and the overlay operation determine the values of the features in the resulting structure. Due to the assimilation procedure unlike unification the default unification represented by the function `defaultunify1` will never throw an overall failure exception.

```
abt := abstract background type
act := abstract cover type
th = TypeHierarchy
co, bg, defaultunifications, validdeltas
PROCEDURE defaultunify1(tfs2)
 1   defaultunifications =  new Collection<TFS>
 2   bg = this //default information
 3   co = tfs2 //strict information
```

```
 4   //...check for shortcuts
 5   deltait = new DeltaIterator(co.getType,bg.getType(),th)
 6   WHILE deltait.hasNext()
 7     delta = deltait.next()
 8     IF delta.isMostSpecific() THEN
 9       abt = delta.getAbt()
10       act = delta.getAct()
11       IF defaultunify2(abt,act).isNotEmpty() THEN
12         validdeltas.add(delta) //specialization was successful
13       ENDIF
14     ENDIF
15   ENDWHILE
16   return defaultunifications
```

If the check for shortcuts that we present later on has no effect, the type preprocessing gets initialized in line 5. The calculated Deltas are succesively examined by their information specificity referring to the given type hierarchy. Only in case they are not more general than some already processed Deltas, the associated abstract background type and abstract cover type will be send to the helper procedure `defaultunify2`. In line 12 the processed Delta will be stored, if it is has yielded a valid result.

```
abs := abstract background structure
acs := abstract cover structures
PROCEDURE defaultunify2(abt,act)
 1   //specialization
 2   TRY acss = co.unify1(act.getMGSat());
 3   CATCH unificationFailureException
 4     return defaultunifications
 5   //generalization
 6   elements = new Collection<FeatureValuePair>
 7   proto_abt = th.getMGSat(abt)
 8   FOR all feat in proto_abt.getFeatures()
 9     elements.add(bg.getFvp(feat).copy())
10   ENDFOR
11   abs = new TFS(abt,elements,th);
```

A Delta that contributes to a further result during default unifying, sticks to the with the successful specialization of the abstract cover type. If uni-

fication fails in line 2, the empty list default unifications will be returned and the next Delta gets calculated. Otherwise the computation continues to generalize the abstract background type. At this stage of the computation the assimilation process is complete.

```
12   FOR all acs in acss
13     elements = new Collection<FeatureValuePair>
14     units =  new Collection<FeatureValueList>
15     feat_acs = // features occurring in acs
16     feat_abs_acs = // features occurring in abs and acs
17   //overlay
18     FOR all feat in feat_acs
19        IF acs.getFvps().contains(feat) THEN
20          elements.add(acs.getFvp(feat))
21        ENDIF
22        IF acs.getFvls().contains(feat) THEN
23          units.add(acs.getFvl(feat))
24        ENDIF
25     ENDFOR //...
```

The default unifier before its recursive application sorts the features located in the translated target structures as well. However in this case it is irrelevant to consider the MGSat of the resulting structure, since well-formedness as we have seen in the analysis is assured by the succeeded specialization. The lines 18-25 describe the overlay operation for the features that occur only in the abstract cover structure. The feature value pair associated to those features gets added to the internal structure of the resulting value.

```
26     FOR all feat in feat_abs_acs
27       abs_feat_ = abs.getFeatureValue(feat)
28       IF acs.getFvps().contains(feat) THEN
29         acs_feat_ = acs.getFeatureValue(feat)
30         defaultunifications' =
31                   abs_feat_.defaultunify1(acs_feat_)
32       ENDIF
33       IF acs.getFvls().contains(feat) THEN
34         acs_feat_ = acs.getFeatureValues(feat)
```

```
35          FOR all acs_feat_' in acs_feat_
36             defaultunifications' =
37                   abs_feat_.defaultunify1(acs_feat_')
38          ENDFOR
39        ENDIF
```

The overlay concerning shared features differentiate as in the unification processing between features pointing to single and assigned disjunctive TFS objects. However latter option may only be embodied by abstract cover structures. Therefore the computation needs to consider only 2 instead of 4 options. Those are shown respectively in the lines 28-32 and the lines 33-39.

```
40        IF defaultunifications'.size() is 1 THEN
41          elements.add(new FeatureValuePair(feat,
42                             defaultunifications'.get()))
43        ELSE units.add(new FeatureValueList(feat,
44                                   defaultunifications'))
45        ENDIF
46        return defaultunifications.add(
47                          new TFS(act,elements,units,th))
48      ENDFOR
49    ENDFOR
```

Finally analogous to the construction of the TFS object during the unification, the default unifier has to check if the default unification for some feature resulted in a single TFS object or in a disjunctive TFS object. The resulting TFS will then be added to the collection of TFS objects that is the return value of the default unification computation.

In the figure 5.7 the illustration shows the application of the depth-first method for the construction of the TFS object during the default unification. The root node of the structure is represented by the top-most node and the dashed lines pointing the nodes denote the type values that the nodes can receive. In particular the abstract cover types detected by the Delta Iterator in some layer in the hierarchy are optionally attached to $q \in Q$,

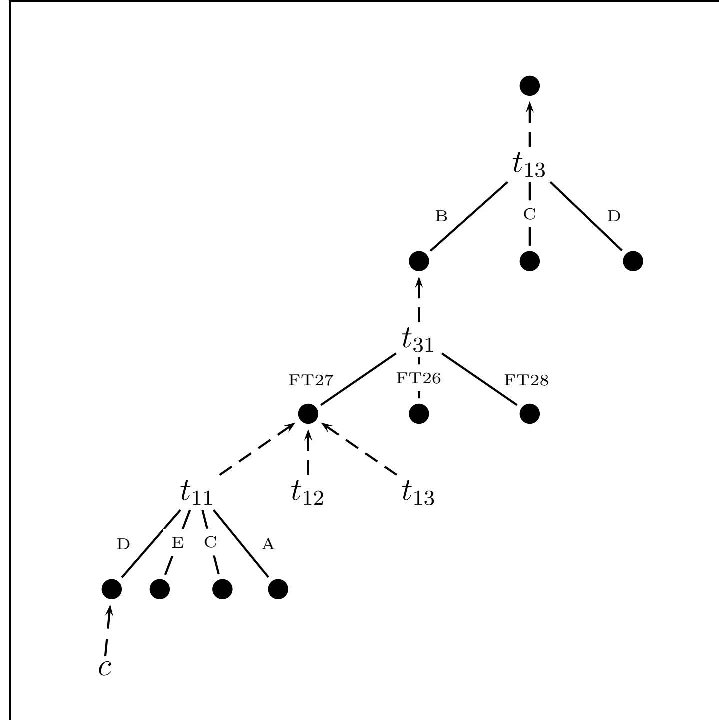

Figure 5.7: The incremental construction of the resulting TFS during the default unification process.

where the left-most will be further processed. The first type provided by the next Delta is t_{13}, which will be attached to the root node. Given that the specialization of the cover to the type definition of t_{13} does not fail the expansion of the internal structure at the first feature continues. For the feature B Delta has detected the abstract cover type t_{31}, whereas at the feature FT_{27} the Delta iteration would capture three different abstract cover types in one layer. Again the structure under t_{31} is processed only if the specialization was successful. The first atomic FS for the return value is finally completed, when the assimilation at feature D yields the atomic type value c. The next recursive call of the default unifier determines the value to be assigned at the feature E. If the internal structure under t_{11} is complete, the processing continues for the already captured Delta in the queue with the abstract cover type t_{12}. Hence if the next specialization does not fail the feature FT_{27} receives a further TFS object with the type t_{12}.

Efficiency Issues

For the unification processing there are conventional methods to abbreviate processings. The following cases can be applied to default unification processing as well. The appropriate result of default unification regarding the single cases below is quiet obvious:

- Cover is equivalent with the *top*.
- Background is equivalent with the *top*.
- Cover and background are equivalent.
- Cover and background has a subsumption relation.
- Cover and background are unifiable.

However default unification offers several more possibilities to save computation. The following pseudo code incorporates in the lines 8-10 the option to replace the subsumption relation on TFS objects by the more economic subsumption test on types, given cover and background are equal with their type definitions. The analogous case is applicable to the unifiability issue visible in the lines 12-17.

```
t_bg := bg.getType()
t_co := co.getType()
cachedefaultunification = Map<Integer,TFS>
 1  proto_bg = th.getMGSat(t_bg)
 2  proto_co = th.getMGSat(t_co)
 3  IF bg.equals(proto_bg) and co.equals(proto_co)
 4    IF th.cachedefaultunification(bg.getType(),co.getType()) THEN
 5      return th.getdefaultunification(bg.getType(),co.getType())
 6    ENDIF //...
 7
 8   IF subsumestype(t_co,t_bg)
 9     return defaultunifications.add(bg)
10   ENDIF //...
11
12  IF th.hasmlb(t_bg,t_co)
13    FOR mlb in mlb(t_bg,t_co)
14       defaultunifications.add(new TFS(mlb,th)
```

```
15    ENDFOR
16    return defaultunifications
17  ENDIF //...
18
cacheunification = Map<Integer,Map<TFS,Collection<TFS>>>
19  IF cacheunification.containsKey(co).containsKey(act)
20    IF cacheunification.get(co).get(act).isEmpty()
21      return defaultunifications
22    ELSE return cacheunification.get(co).get(act)
23  ENDIF //...
```

The indexed memoization of calculations that comply with specific properties allows to reduce one default unification process to a simple look up in a hash table. In particular if both arguments as well the background as the cover are of the same structure of their type definitions, the default unification processing will store the result in a mapping from an array-like encoding of the types of background and cover to the resulting TFS object. The administration of the table is conducted on a superior level of the default unifier, that is in our case the instantiation of the type hierarchy. Recurring default unifications at some later point with the same arguments can be easily sorted out in advance, so that further processings are disabled, as can be taken from the lines 3-6. Similar approach we apply to the specialization, where the abstract cover structure gets associated to its type and the cover. The corresponding look up is shown in line 19 to 23.

```
1  IF co.getFeatures().isEmpty()
2     and abs.getFeatures().isEmpty()
3    validdeltas.add(delta)
4    defaultunifications.add(th.getMGSat(act).copy())
5    continue //...
6  ENDIF
```

The above abbreviation is exceptional in so far as it can only be potentially exploited after the assimilation process. Therefore the computation needs to come across the case when the generalization of the background results in a TFS object that has no internal structure. If additionally the cover is of the same property, then the default unification of the cover and the background matches to the type definition of the abstract cover type.

System Architecture

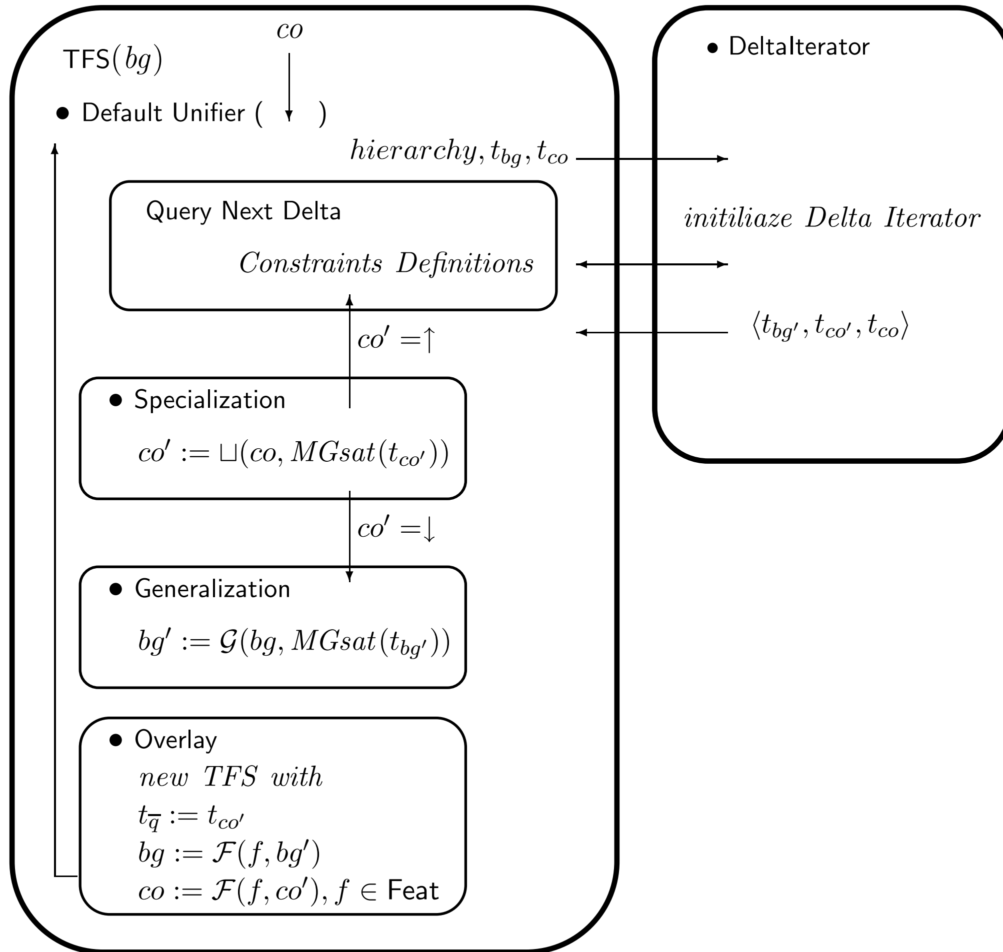

Figure 5.8: Overview of the interactions of the single parts in the implementation. After the cover is passed to the default unifier, the delta iterator gets instantiated. The query of the next Delta follows the specialization that decides if the processing is delayed by the query of the next Delta or if it continues with the generalization and at last the overlay operation. The latter component comprises the recursive application of the entire procedure to the internal structure of the assimilated structures.

5.2.3 An Extract of an Example

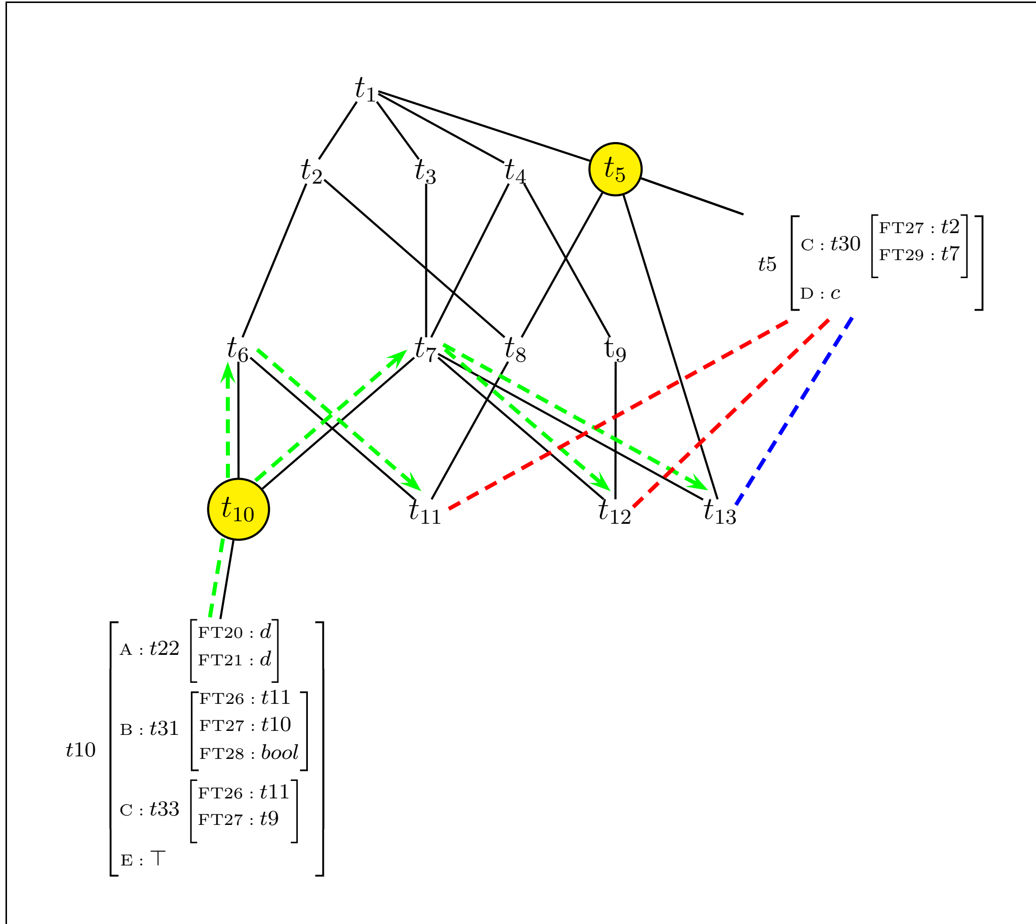

Figure 5.9: This image represents the effect of the first Delta request, entailing the expansion to all supertypes in the first layer atop the background type. The hierarchy refers to a fragment of the implemented type hierarchy in the figure B.1. The green lines indicate the navigated route computed by the Delta iterator. The red lines indicate that the unification between the TFS of the cover argument and the prototype structure of the abstract cover type fails, whereas the blue line says that unification succeeds.

Given the well-formed background and cover as one can learn from the type hierarchy in the figure B.1 within the type definitions are explicitely displayed, the initialized Delta Iterator identifies three different type configuration on the first layer atop the type of the background. The following unifications determine if some Delta is valid.

$$\begin{pmatrix} \Delta_1 := \langle t_7, t_{13} \rangle \\ \Psi_1 := \sqcup(co, MGsat(t_{13})) \end{pmatrix} : co \sqcup t_{13} \begin{bmatrix} \text{D} : f \\ \text{C} : t_{32} \\ \text{B} : t_{31} \begin{bmatrix} \text{FT28} : bool \\ \text{FT27} : t_5 \\ \text{FT26} : t_7 \end{bmatrix} \end{bmatrix} = t_{13} \begin{bmatrix} \text{D} : f \\ \text{C} : t_{34} \\ \text{B} : t_{31} \begin{bmatrix} \text{FT28} : bool \\ \text{FT27} : t_5 \\ \text{FT26} : t_7 \end{bmatrix} \end{bmatrix}$$

$$\begin{pmatrix} \Delta_2 := \langle t_7, t_{12} \rangle \\ \Psi_2 := \sqcup(co, MGsat(t_{12})) \end{pmatrix} : co \sqcup t_{12} \begin{bmatrix} \text{F} : \top \\ \text{C} : t_{29} \\ \text{B} : t_{28} \\ \text{D} : d \end{bmatrix} = \uparrow$$

$$\begin{pmatrix} \Delta_3 := \langle t_6, t_{11} \rangle \\ \Psi_3 := \sqcup(co, MGsat(t_{11})) \end{pmatrix} : co \sqcup t_{11} \begin{bmatrix} \text{E} : \top \\ \text{D} : c \\ \text{C} : t_{31} \\ \text{A} : t_{20} \begin{bmatrix} \text{FT20} : i \end{bmatrix} \end{bmatrix} = \uparrow$$

Only the path leading to t_7 yields on the first layer the type configuration $\langle t_7, t_{13} \rangle$ that also produces an abstract cover structure, whereas other type configurations are to be discharged from computation. It follows that the generalization to the abstract background type propagated by the valid Delta provides the information from the background to be considered during the overlay operation.

$$\Theta := \mathcal{G}(bg, t_7) = t_7 \begin{bmatrix} \text{B} : t_{31} \begin{bmatrix} \text{FT26} : t_{11} \\ \text{FT27} : t_{10} \\ \text{FT28} : bool \end{bmatrix} \\ \text{C} : t_{33} \begin{bmatrix} \text{FT26} : t_{11} \\ \text{FT27} : t_9 \end{bmatrix} \end{bmatrix},$$

$$overlay(\Theta, \Psi_1) = t_{13} \begin{bmatrix} \text{D} : f \\ \text{C} : defaultunify \left(t_{33} \begin{bmatrix} \text{FT26} : t_{11} \\ \text{FT27} : t_9 \end{bmatrix}, t_{34} \right) \\ \text{B} : defaultunify \left(t_{31} \begin{bmatrix} \text{FT26} : t_{11} \\ \text{FT27} : t_{10} \\ \text{FT28} : bool \end{bmatrix}, t_{31} \begin{bmatrix} \text{FT28} : bool \\ \text{FT27} : t_5 \\ \text{FT26} : t_7 \end{bmatrix} \right) \end{bmatrix},$$

...

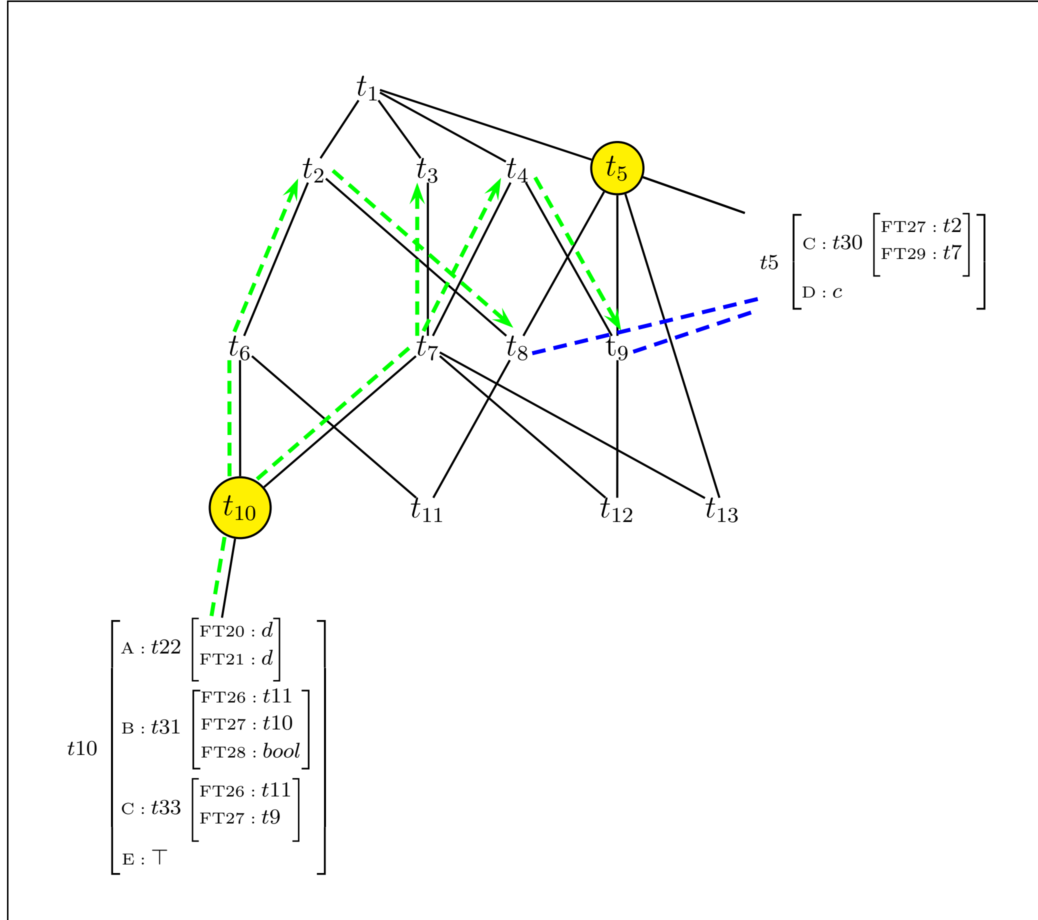

Figure 5.10: The second request of a Delta effectuate the expansion to all supertypes in the second layer atop the background type.

In this exemplification we stick to the examination of TFSs that are calculated in the default unification at the level of the root node. Therefore we keep on tracking each outgoing arc of the visited background supertype so far. The next layer consits of the types t_2, t_3 and t_4. Since t_3 has no *mlb* with t_5, the next type configurations to be delivered by the Delta iterator are $\langle t_2, t_8 \rangle$ and $\langle t_4, t_9 \rangle$. We have the situation that both type configurations entail successful specializations. At this stage of the default unification process, the assimilation would terminate after reaching the next layer, where

all pursuited paths meet at the *mub* node t_1. Lastly the next layer would not bring any more result, since the unique type configuration in the last layer $\langle t_1, t_5 \rangle$ is more general than type configurations, that are computed by now.

- $\begin{pmatrix} \Delta_1 := \langle t_2, t_8 \rangle \\ \Psi_1 := \sqcup(co, MGsat(t_8)) \end{pmatrix} : co \sqcup t_8 \begin{bmatrix} \text{C} : t27 \\ \text{D} : c \\ \text{A} : t_{20} \begin{bmatrix} \text{FT20} : g \end{bmatrix} \end{bmatrix} = t_8 \begin{bmatrix} \text{C} : t30 \\ \text{D} : c \\ \text{A} : t_{20} \begin{bmatrix} \text{FT20} : g \end{bmatrix} \end{bmatrix},$

$$\Theta := \mathcal{G}(bg, t_2) = t_2 \begin{bmatrix} \text{A} : t_{22} \begin{bmatrix} \text{FT20} : d \\ \text{FT21} : d \end{bmatrix} \\ \text{C} : t_{33} \begin{bmatrix} \text{FT26} : t_{11} \\ \text{FT27} : t_9 \end{bmatrix} \end{bmatrix},$$

$$overlay(\Theta, \Psi_1) = t_8 \begin{bmatrix} \text{A} : defaultunify \left(t_{22} \begin{bmatrix} \text{FT20} : d \\ \text{FT21} : d \end{bmatrix}, t_{20} \begin{bmatrix} \text{FT20} : g \end{bmatrix} \right) \\ \text{C} : defaultunify \left(t_{33} \begin{bmatrix} \text{FT26} : t_{11} \\ \text{FT27} : t_9 \end{bmatrix}, t_{30} \right) \end{bmatrix},$$

...

- $\begin{pmatrix} \Delta_2 := \langle t_4, t_9 \rangle \\ \Psi_2 := \sqcup(co, MGsat(t_9)) \end{pmatrix} : co \sqcup t_9 \begin{bmatrix} \text{F} : \top \\ \text{C} : t_{27} \\ \text{D} : b \end{bmatrix} = t_9 \begin{bmatrix} \text{F} : \top \\ \text{C} : t_{30} \\ \text{D} : e \end{bmatrix},$

$$\Theta := \mathcal{G}(bg, t_3) = t_4 \begin{bmatrix} \text{C} : t33 \begin{bmatrix} \text{FT26} : t11 \\ \text{FT27} : t9 \end{bmatrix} \end{bmatrix},$$

$$overlay(\Theta, \Psi_2) = t_9 \begin{bmatrix} \text{F} : \top \\ \text{D} : e \\ \text{C} : defaultunify \left(t_{33} \begin{bmatrix} \text{FT26} : t_{11} \\ \text{FT27} : t_9 \end{bmatrix}, t_{30} \right) \end{bmatrix},$$

...

At this point the processing of the default unification, has already produced at least three TFSs with the types t_{13}, t_8 and t_9 at their root node. The recursion of the default unification applied to the inner structure may produce further results. The complete processing of the default unification with the given background and cover can be retraced in the appendix C. The accomplished result of the entire default unification run resolves into thirteen different TFS objects.

Chapter 6

Conclusions and Future Work

6.1 Summary

Multiple inheritance hierarchies emerge in different fields like knowledge representation and reasoning, database management and query processing, and object-oriented programming. The demand to have a compact representation of the hierarchy with the ability to compute relationships efficiently is shared between these domains. In this work, an efficient implementation of an operation concerning the default reasoning in multiple inheritance hierarchies has been provided. Based on a thorough study of recent research and the profound analysis of the effects including multiple inheritance hierarchies, it has been accomplished to extend achievements—with regard to both the formal theory and implementational issues —in the research of default reasoning that has been performed up to now.

- **Sophisticated extension to the algorithm described in (Alexandersson & Becker, 2007)**

  - We motivate to incorporate the notion of well-formedness into default reasoning. Thus we propose to employ well-formed TFS along the processing of default unification. Consequently we constitute and implement the well-formed unification operation as discussed in (Copestake, 1992) which involves that the default unification operation produces well-formed TFSs only.
  - One of the main contribution is that we have provided a reformulation of Carpenter's definition of credulous default unification in

(Carpenter, 1993) and a precise formalized characterization of an adequate algorithm. In contrast to previous works we have considered hierarchies as bounded partial orders that are not complete. In this context an extensive investigation of the exhaustive covering of all results during the preprocessing on types in multiple inheritance hierarchies has been performed. The non-deterministic behaviour particularly during the lattice operations increases considerably the degree of difficulty to allocate all possible outcomes of the default unification operation.

- Within the implementation we provide a convenient management of valid type configuration detection. The Delta Iterator allows the constraint imposed on the supertypes of the background to be easily exchangeable. This factor facilitates for instance to convert the default unification in BPO hierarchies to the default unification in BCPO hierarchies. Besides, the incremental mode of the search enables to query the needed information on demand. In conjunction with a *sensitive* scoring functionality presented in (Alexandersson et al., 2004) it might be possible to interrupt the calculation on the level of the type preprocessing. Further this enables also to change the constraint during the process of a dialog, which is practical in the context of utterance ambiguity of the user. Those issues need more investigation and we will add them to the agenda of future works.

- **A practical solution to ambiguous TFS**

  One pivotal issue in the course of research is the usage of the TFS logic as the formal language providing the basis for implementing and testing the default unifier. We propose and put into effect a compact representation of TFS that is capable of expressing ambiguity within the structures. In computer science literature some authors have motivated to examine adequate representations of ambiguous TFS as well. A approach considering ambiguous TFS has been carried out in (Duchier, 2003), though referring to the context of constraint programming applicable to syntactical parsing tasks. In (Romanelli, 2005) it is discussed how plurality can be embedded into an ontology-based dialog system like SmartKom. In the outlook of Romanelli's work it is suggested, in order to have a more comfortable background to define sets it would

be reasonable to introduce a TFS language that accepts the transition function $\delta : \mathsf{Feat} \times Q \rightarrow P(Q)$ with features pointing to a set of TFS.

- **Employment of efficient lattice operations**

  Furthermore we have built an interlinking of an implementation of multiple inheritance hierarchies together with the realization of default unification based on the extended algorithm. Thereby we have complemented the established platform regarding the efficient implementation of the required lattice operations. The framework enables to experiment with TFSs in multiple inheritance hierarchies featuring efficient *mlb* and *mub* operations based upon the bit-code image of the type hierarchy.

6.2 Future Work

Basically the analysis of default unification dealing with TFS is adaptable to appropriate frame-based formalisms. Since TFSs are suited for knowledge representation, the concept of type definitions can be mapped to the notion of frame objects.

- **Testing with large real-world hierarchies**

  We suggest that an ontology-based system such as SmartKom is extendable to feature the multiple inheritance property in the context of default reasoning as well. In particular SmartKom would represent a "playground" to test the implementation of the default unification operation on large-scale hierarchies. Further, such a measure entails to enable the continuation of other researches based on the dialog backbone of the SmartKom system, e. g., the purpose of illuminating the integration of multiple inheritance in the ontology considering the plurality issue in (Romanelli, 2005).

- **Employing alternative bit-encoding methods**

  Since the efficiency issue of this implementation heavily depends on lattice operation working on a bit-code image of the hierarchy, we suggest to adopt this technique to adequately encode the ontology in systems such as SmartKom. However, in this work we have assumed that the implemented type hierarchy needs to be preprocessed only once and

is not a subject for further modifications afterwards. Analogously the compiling of the bit-code image is done prior to the desired operations applied to TFSs. Therefore, besides using a more compact bit-encoding technique that enables to save more memory space, it is reasonable to install a code-image of the type hierarchy that allows to be incrementally updated as well (Bommel & Beck, 1999). The support of dynamic ontology development would make the application of type bit-encoding more attractive in areas of ontology maintenance.

- **Account on structure sharing in TFS**

  The production of multiple results in this work is caused by the employment of a type hierarchy that heavily performs multiple inheritance. In contrast, Carpenter's credulous default unification produces ambiguous results that he motivates by given reentrant structures in FSs. Hence the involvement of structure sharing in the TFS language would result into a superior dimension of ambiguity during the default unification:

  - Coreferences in the background trigger additional ambiguity if overlay performs with involved features that stem from the background only—that is the case if the type of the cover subsumes the type of the background. As for instance the reentrant values for some feature f_1 and f_2 is to be default unified at f_1 with a type value stemming from the cover and at f_2 with nothing. A type clash in the former effects to have a version of the result where coreference "survives" and overwrites the background-stemming value for the feature f_2, and additionally a result with a discarded coreference and a kept background-stemming value for the feature f_2.
  - Structure sharing that occurs in the cover is kept in any case during the default unification process. Ambiguity arises if overlay combines between reentrant values from the cover each with some type value from the background, where the background-stemming values are more specific than the corresponding cover value and not consistent among each other. The outcome performs structure sharing with multiple variants, respecting all the clashing type values that stem from the background.

Along the research of default unification in multiple inheritance hierarchies we have gained even more theoretical insights considering coreferences. However, most of the implementational issues in this context we postpone as a goal for future research.

- **Incorporating a score-like functionality**

  A crucial contribution in (Alexandersson & Becker, 2007) is the usage of a scoring function that computes the best hypothesis among the multiple outcome of the default unification process. The notion of *informational distance* appears to be a reasonable device to restrain the production of multiple results in the context of this work as well. An interesting application-oriented concern is to investigate the relation between feature decrease in the background and the feature increase in the cover during assimilation. Possibly, a potential tendency to favor information loss in the background with respect to information gain in the cover or vice versa can be detected. This issue presumes the expansion of the precomputation on types to consider also appropriate features.

Appendix A

Import of the Type Hierarchy

A.1 Binary file generated by flop

A specification of the binary file produced by flop is listed in the following appendix. It is adapted from the definitions given in (Callmeier, 2001). The listing does not show all possible sections generated by flop from the $\mathcal{TDL}$ input, yet the relevant ones that are useful for the implementation of the default unifier. The semi-formal specification of the file format splits up the file into sections. The latter are further recursively decomposed into smaller elements down to atomic elements. Those are *int*, *short* and *char* that respectively consists of four bytes, two bytes and one byte. The multi-digit atomic elements are stored in little endian byte order.

A.1.1 Sections

type	description
header	Header to identify the file
toc	Offsets to sections in file
symbol-tables	Symbols in the grammar
hierarchy	Type hierarchy
constraints	Constraints of types and instances

header (Information identifying the file format and the name of the grammar contained in the file)

type	identifier	description
int	magic	Magic value to identify file format
int	version	Version of file format
string	description	Description (name) of the grammar

string

type	identifier
short	length
length x *char*	text

toc (Offsets to later sections in file, allowing a program to skip over sorted out sections)

type	identifier
int	offset-symboltable
int	offset-hierarchy
int	offset-constraints
int	offset-supertypes

symbol-tables (Names for the objects of the grammar)

type	identifier
int	npropertypes
int	nleaftypes
int	nattrs
ntypes[a] x *string*	type-name
nattrs x *int*	attr-name

[a] ntypes = npropertypes + nleaftypes

hierarchy (Information about the type hierarchy underlying the grammar)

type	identifier
int	nbits
npropertypes x *bitcode*	type-bitcode
nleaftypes x *int*	leaftype-parent

bitcode (The Bitencoding, applied to represent the types and to decode the type hierarchy)

type	description
bitcodepart	Parts of bitcode
...	repeated until end marker
int = 0	End marker
short = 0	End marker

bitcodepart

type	identifier
int	value
value $\neq 0$: *short* $\neq$ 0	repetition

constraints (Feature structure constraints of types)

type	identifier
ntypes x *constraint*	Constraints associated to types

constraint

type	identifier
int	nnodes
int	narcs [b]
nnodes x *node*	node

[b]

$$narcs = \sum_{0<i<nnodes} nattrs_i$$

nattrs value respective to the node visited

node

type	identifier
int	type
short	nattrs
nattrs x *arc*	arc

arc

type	identifier
short	attribute
short	value

supertype

type	identifier
npropertypes x *supertypes*	supertypes

supertypes

type	identifier
short	nsupertypes
nsupertypes x *int*	propertype-parent

A.2 Binary file read in by ShuG

ShUG (Shallow Unification Grammars) is a collection of objects and methods that enables to read in the binary file format of the flop preprocessor. Given the binary output presented in the previous appendix ShUG pipes the binary file into a byte array. In order to manage the desired sections in java, all the methods required to assemble the atomic elements (*int*, *short*, *char*) of the file format must be implemented. The atomic elements encode the location of the desired information and the information itself in the binary output. So the byte array is read out consecutively from the beginning up to the TOC section that contains the pointer to the sections and allows to navigate in the binary file by switching to the focused sections.

A.2.1 undump int

This method assembles *integers* picked out of the byte array. Four bytes are attached successively, whereas the latter ones are appended at the front (little endian order).

A.2.2 undump short

This method assembles *shorts* picked out of the byte array. Two bytes are attached successively, whereas the latter ones are appended at the front (little endian order).

A.2.3 undump string

The readout of a *string* is performed in 2 steps. First a *short* determines the length of the *string*, whereas the last character defined by the binary representation is a null character and is to be ignored. Then the number of bytes corresponding to the length is stored in a byte buffer, that is eventually transformed into a char buffer (one byte maintains one char character).

A.2.4 undump node

This method reads out at first an *int* that is the type identifier of the feature structure linked to the node, then a *short* that stands for the number of arcs

viz. feature value pairs. Finally for each arc the undump arc method is called (See the constraints section).

A.2.5 undump arc

Two *shorts* that indicate first the feature identifier then the index of the array containing the feature structures constructed so far (see constraints section).

A.2.6 undump bitcode

The bit-encoding of the proper types for the *glb* computation are stored in an *int array*. Based on the length of the codesize that is identified in the hierarchy section, a sequence of *ints* are read out and laid down in the array. Long sequences of zeros are given in a compressed mode. Once an *int* with the value zero is read out, a following *short* informs about the number of zeros in the queue. A code snippet ends with an int zero or a short zero.

A.2.7 Header section

The header section consists of two *integers* and one *string*, that represents information concerning the file format, only. This information has no further relevance for the computation.

A.2.8 TOC section

The TOC section stores the offset of the different sections in the byte array as *ints* and enables the access to the individual sections. The offset values are mapped by a so-called section types represented by an *int*, where each preset value stands for some specific section. The output of flop makes ten different sections available. In addition to the header section and the TOC section, we put the focus on other four sections. Those are the symbol-table section that has the section type one, the hierarchy section with the section type three, the constraints section with the section type seven and the supertypes section with the section type ten. Each section begins with the encoding of its section type.

A.2.9 Symbol-table section

The symbol-table section contains important knowledge about the hierarchy that further are also captured in a log file. The number of types, in particular the number of proper types and leaf types and the number of features are stored in this section. Types in the system are associated with so-called status that takes value such as `atom,lex-rule` etc. The number of status and their values are also managed in this section, however this part can be ignored since our implementation has to handle only proper types and leaf types. The creation of a mapping between the type identifier an *int* and the type name a *string* and the inverse mapping is carried out by the iteration over the number of types and the readout of type names. The type identifier is equal with the position of the type in the array and therefore determined by the order of types given in the binary output. Generally all leaf types are stored after the proper types. This happens analogously with features. Altogether, the convenient mappings `type2number`, `feature2number`, `number2types` and `feature2types` are established by the readout of this section.

A.2.10 Hierarchy section

The hierarchy section starts with the section type number three, followed by the information about the number of bits (*int*) that are needed to encode all the proper types in the type hierarchy. Then for each proper type the code represented by vectors of *ints* is read out by the `undump bitcode`. A mapping between proper type identifiers (*ints*) and codes (*int*[ ][ ]) is established, that is called `gamma`. Analogous to the introduced mappings so far, an adequate inverse mapping called `gammaInverse` is installed. Finally, an association between leaf type identifiers (*ints*) and their parent type identifiers (*ints*) is build up, which is necessary for the employed technique of the glb computation in ShUGA.

A.2.11 Constraints section

This section reads out the constraints of the unfilled feature structures [1], both for the proper and the leaf types. For a given type identifier, every feature structure as well as every feature-value pair is assigned to an index

[1] Unfilled feature structures corresponds to the most general satisfier of the feature structure

and can be accessed through the global arrays `this.featureStructures` and `this.featureValuePair`. These arrays are build up for every type that is read in. The procedure is as following; for every type, one iterates over the number of nodes (undump short) of feature structures which are associated with the individual type identifier. For each node a new feature structure with the appropriate type identifier (undump int) is instantiated resulting in an array of feature structures. Finally for each feature structure there are numbers of arcs (undump short) associated, each representing a feature value pair consisting of a feature (undump short) and the array index (undump short) containing the feature structures. The last index of the array returns the complete assembling of the structure for one type identifier. For instance, given the example below we instantiate an array of feature structures with the size of the number of nodes assigned to the type identifier of t_5, that is three. The feature structures are respectively instantiated with the types t_{27}, a and t_5. We recognize that only the latter structure consists of arcs. Given two arcs, the feature value pair with the feature C assigned to the value that is the first position in the array of feature structures and the feature value pair with the feature D assigned to the value that is the second position in the array of feature structures array are created. The last and third position in the array corresponds to the value of the entire structure.

$$\begin{bmatrix} t_5 & \\ \mathrm{C} & t_{27} \\ \mathrm{D} & a \end{bmatrix}$$

A.2.12 Supertypes section

The last discussed section deals with information on supertype relations of the proper types. A *short* declares the number of supertypes the respective proper type has and each subsequent *int* matches the type identifier of one supertype. The implementation of the readout of the supertypes section has been entirely added to ShUG, whereas the readout of the other sections was subject to slight modifications. Further the administration of the data concerning the type hierarchy in ShUG has been adapted to the design of the implementation of the default unifier.

Appendix B

Building the Type Hierarchy

The implementation of the type hierarchy is connected with the challenging task to build a hierarchy, that allows to experiment sufficiently with the introduced scenarios along the analysis. The employed type hierarchy features, in total 33 proper types and 6 leaf types. Additionally 3 synthetic types have been generated and embedded into the hierarchy by the processing of flop. We omit to display explicitely the synthetic types, instead we give the explanation of when and where they get inserted. The entire hierarchy consists of five substructures that are attached to the *top*. For the purpose of clarity we divide the visual image into 3 subparts demonstrating the type hierarchy A, B and C respectively.

The first part of the hierarchy consists of only proper types, where two supplemental synthetic types have to be additionally considered during the default unification process. The *mlb* between t_4 and t_5 equals $\{t_9, t_{13}\}$, that implicates the bridging of the subsumption relations between the pairs by a synthetic type syn_1. Since the *mlb* between t_5 and t_7 in the original version of the hierarchy produces a set of results with more than one element as well, there is still one more synthetic type involved. In the now modified hierarchy the non-determinism is forwarded to the *mlb* of syn_1 and t_7. Hence by flop a further synthetic type syn_2 is introduced as the new *glb* between the types t_5 and t_7. The background in the concrete example of default unification in the next appendix C is a well-formed TFS of the type t_{10} and the cover is a well-formed TFS of the type t_5.

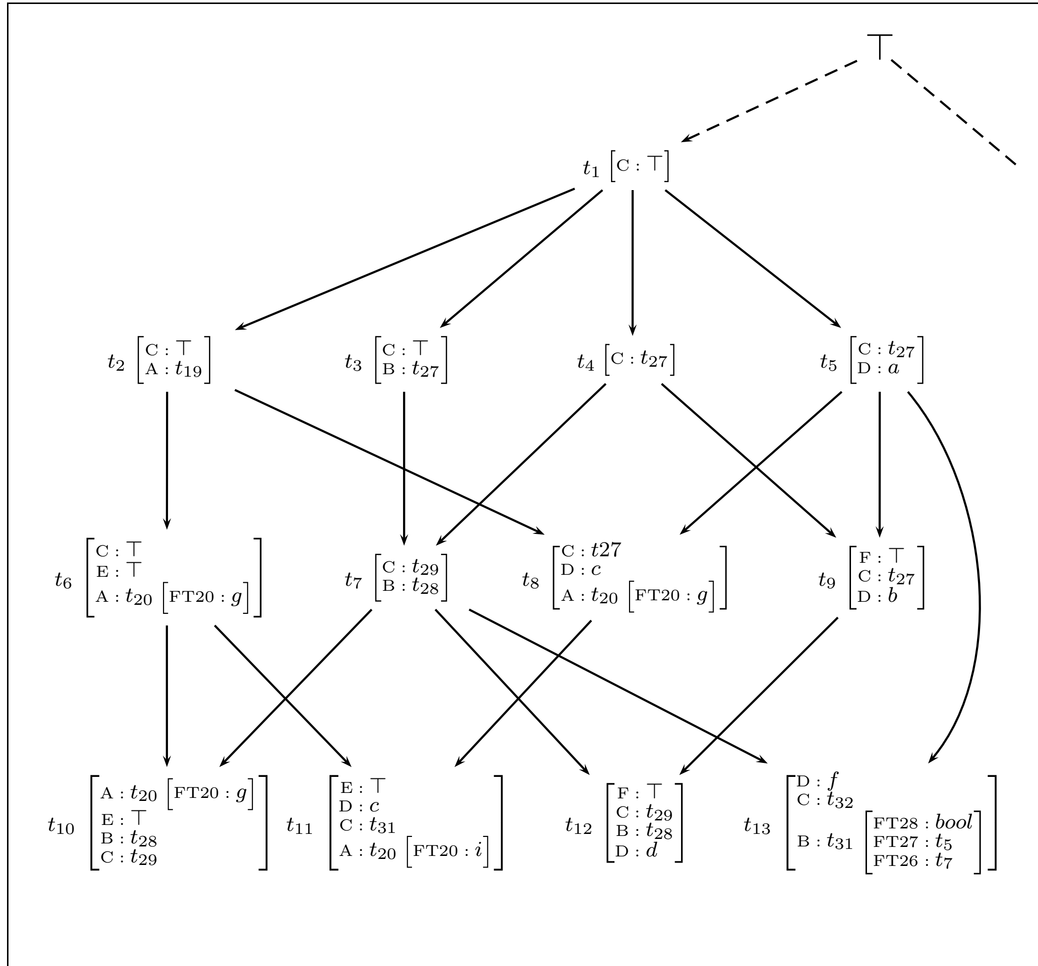

Figure B.1: Type Hierarchy A

Notice that most non-expanded TFSs within the internal structure of the type definitions in the type hierarchy A are specified one level further in the type hierarchy B. Moreover the values of features in the type definitions of the latter hierarchy refers back again to the type definitions in A. This is solely motivated for the purpose to obtain the demonstrative effect of the default unification example, as we can create complex scenarios without the usage of an oversized hierarchy.

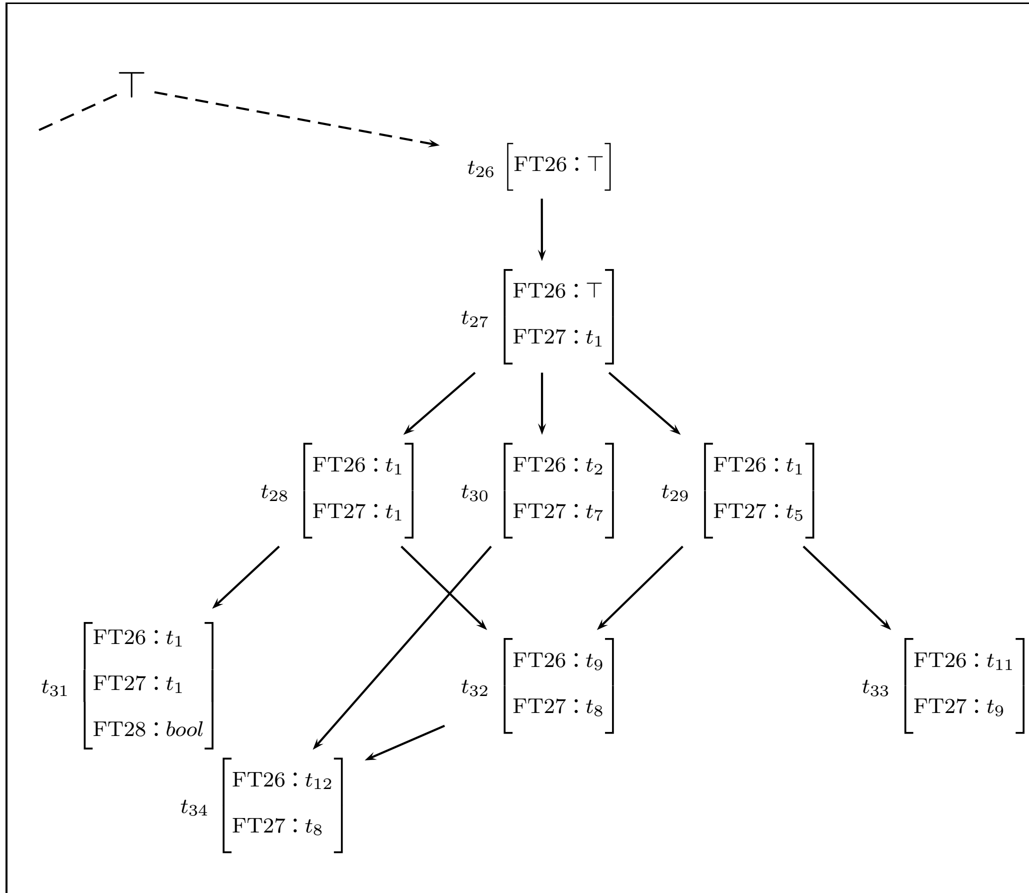

Figure B.2: Type Hierarchy B

As we have mentioned concerning the illustration of well-formed type feature structures in the analysis of chapter 3, we must consider that type definitions would allow cyclicity if the subsumption ordering is not preserved for the types situated along the path of a TFS . Though the application of the partial expansion technique prevents the construction of TFS objects from suffering infinite progression. In particular the implementation of the default unifier is still valid when operating in a hierarchy with type definitions maintaining the compatability condition. The type hierarchy B includes 2 leaf types t_{31} and t_{32}. Even if it seems to be the case, B does not perform multiple joins or meets.

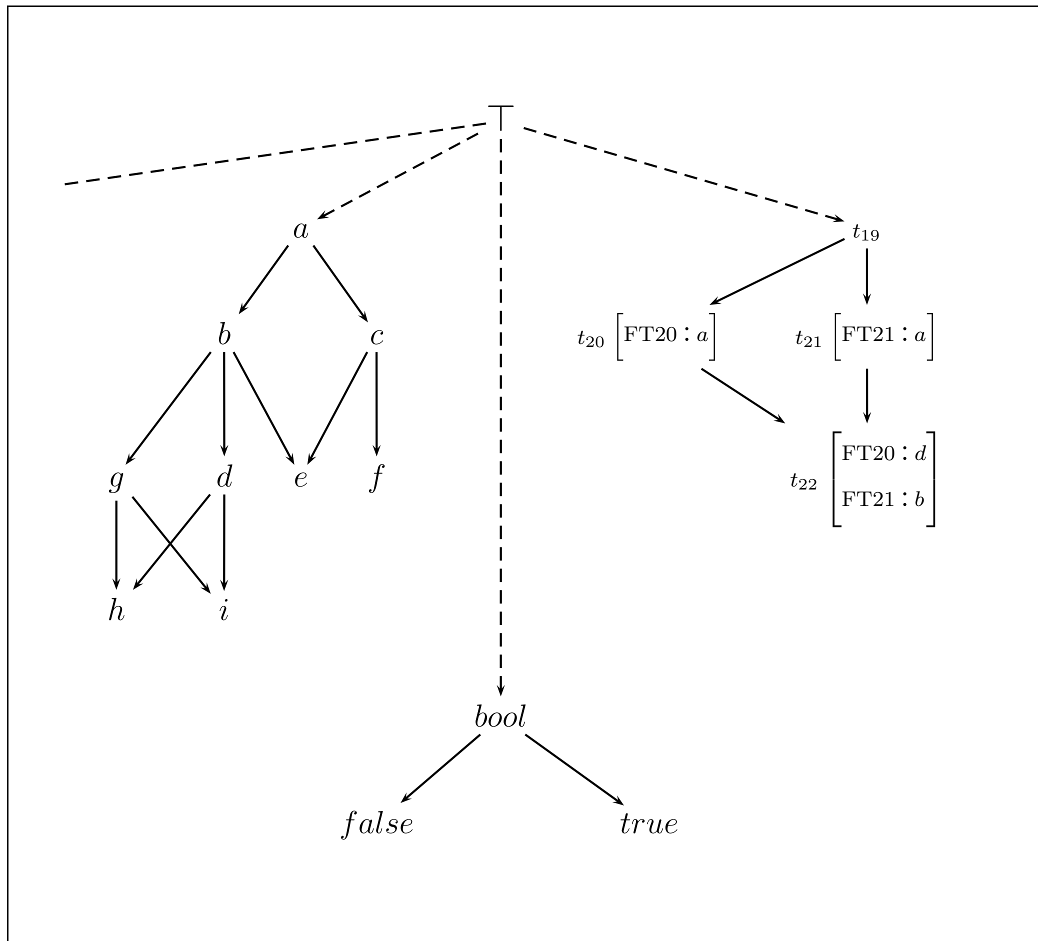

Figure B.3: Type Hierarchy C

In turn the type hierarchy C has one synthetic type under the structure consisting of only atomic values. The synthetic type is inserted evidently as the *glb* between the types g and d. Furthermore in C, the five types f, *bool*, *true* and *false* are among the leaf types.

Appendix C

Commented Output of a Concrete Default Unification

```
background
t10 & [ A t22 & [ FT20 d
                  FT21 d ]
        B t31 & [ FT26 t11 & [ ]
                  FT27 t10 & [ ]
                  FT28 bool ]
        C t33 & [ FT26 t11 & [ ]
                  FT27 t9 & [ ]]
        E *top* ]

cover
t5 & [ C t30 & [ FT27 t2 & [ ]
                 FT26 t7 & [ ]]
       D c ]


**************************************************************
**************************************************************
START NEW DEFAULT UNIFICATION PROCESS

**************************************************************
```

```
STARTING DELTAITERATOR FOR BG, CO : t10, t5
LUB(s) [t1]

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :true
assimilation deltas.next : t5 t6 t11

***************************************************************
SPECIALIZATION
unify between following arguments:
t5 & [ C t30 & [ FT27 t2 & [ ]
                 FT26 t7 & [ ]]
       D c ]
t11 & [ E *top*
        D c
        C t31 & [ ]
        A t20 & [ FT20 i ]]

Warning : unification between cover and the prototype structure
of the GLB(cover,bg') has FAILED!

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :true
assimilation deltas.next : t5 t7 t12

***************************************************************
SPECIALIZATION
unify between following arguments:
t5 & [ C t30 & [ FT27 t2 & [ ]
                 FT26 t7 & [ ]]
       D c ]
t12 & [ F *top*
        C t29 & [ ]
        B t28 & [ ]
        D d ]
```

```
Warning : unification between cover and the prototype structure
of the GLB(cover,bg') has FAILED!

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :true
assimilation deltas.next : t5 t7 t13

************************************************************
SPECIALIZATION
unify between following arguments:
t5 & [ C t30 & [ FT27 t2 & [ ]
                 FT26 t7 & [ ]]
       D c ]
t13 & [ D f
        C t32 & [ ]
        B t31 & [ FT28 bool
                  FT27 t5 & [ ]
                  FT26 t7 & [ ]]]


GENERALIZATION
raise the first TFS to the second TFS:
t10 & [ A t22 & [ FT20 d
                  FT21 d ]
        B t31 & [ FT26 t11 & [ ]
                  FT27 t10 & [ ]
                  FT28 bool ]
        C t33 & [ FT26 t11 & [ ]
                  FT27 t9 & [ ]]
        E *top* ]
t7 & [ C t29 & [ ]
       B t28 & [ ]]

generalized bg :
t7 & [ C t33 & [ FT26 t11 & [ ]
                 FT27 t9 & [ ]]
       B t31 & [ FT26 t11 & [ ]
```

```
                 FT27 t10 & [ ]
                 FT28 bool ]]
specialized co :
t13 & [ B t31 & [ FT28 bool
                  FT27 t5 & [ ]
                  FT26 t7 & [ ]]
        C t34 & [ ]
        D f ]

**************************************************************
FOR feat D in set of features only in the specialized co :
new FeatureValuePair (D,f )

**************************************************************
FOR feat B in set of shared features DO
default unify between following arguments:

 new bg argument :
t31 & [ FT26 t11 & [ ]
        FT27 t10 & [ ]
        FT28 bool ]
 new co argument :
t31 & [ FT28 bool
        FT27 t5 & [ ]
        FT26 t7 & [ ]]

**************************************************************
**************************************************************
START NEW DEFAULT UNIFICATION PROCESS

**************************************************************
STARTING DELTAITERATOR FOR BG, CO : t31, t31
LUB(s) [t31]

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :true
assimilation deltas.next : t31 t31 t31
```

```
************************************************************
SPECIALIZATION
unify between following arguments:
t31 & [ FT28 bool
        FT27 t5 & [ ]
        FT26 t7 & [ ]]
t31 & [ FT26 t1 & [ ]
        FT27 t1 & [ ]
        FT28 bool ]


GENERALIZATION
raise the first TFS to the second TFS:
t31 & [ FT26 t11 & [ ]
        FT27 t10 & [ ]
        FT28 bool ]
t31 & [ FT26 t1 & [ C *top* ]
        FT27 t1 & [ C *top* ]
        FT28 bool ]

generalized bg :
t31 & [ FT26 t11 & [ ]
        FT27 t10 & [ ]
        FT28 bool ]
specialized co :
t31 & [ FT28 bool
        FT27 t5 & [ ]
        FT26 t7 & [ ]]

************************************************************
FOR feat FT28 in set of shared features DO
default unify between following arguments:

 new bg argument : bool
 new co argument : bool

************************************************************
```

```
**************************************************************
START NEW DEFAULT UNIFICATION PROCESS

cover.equivalentTFS(background)
SHORTCUT OF DEFAULT UNIFICATION SUCCEED

new FeatureValuePair(FT28,[bool ])

**************************************************************
FOR feat FT27 in set of shared features DO
default unify between following arguments:

 new bg argument : t10 & [ ]
 new co argument : t5 & [ ]

**************************************************************
**************************************************************
START NEW DEFAULT UNIFICATION PROCESS

**************************************************************
STARTING DELTAITERATOR FOR BG, CO : t10, t5
LUB(s) [t1]

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :true
assimilation deltas.next : t5 t6 t11

**************************************************************
SPECIALIZATION
unify between following arguments:
t5 & [ C t27 & [ ]
       D a ]
t11 & [ E *top*
        D c
        C t31 & [ ]
        A t20 & [ FT20 i ]]
```

```
GENERALIZATION
raise the first TFS to the second TFS:
t10 & [ E *top*
        B t28 & [ ]
        C t29 & [ ]
        A t20 & [ FT20 d ]]
t6 & [ C *top*
       E *top*
       A t20 & [ FT20 b ]]

generalized bg :
t6 & [ C t29 & [ ]
       E *top*
       A t20 & [ FT20 d ]]
specialized co :
t11 & [ E *top*
        D c
        C t31 & [ ]
        A t20 & [ FT20 i ]]

*************************************************************
FOR feat D in set of features only in the specialized co :
new FeatureValuePair (D,c )

*************************************************************
FOR feat E in set of shared features DO
default unify between following arguments:

 new bg argument : *top*
 new co argument : *top*

*************************************************************
*************************************************************
START NEW DEFAULT UNIFICATION PROCESS

typeBg == typeHierarchy.TOP_TYPE
SHORTCUT OF DEFAULT UNIFICATION SUCCEED
```

```
new FeatureValuePair(E,[*top* ])

**************************************************************
FOR feat C in set of shared features DO
default unify between following arguments:

 new bg argument : t29 & [ ]
 new co argument : t31 & [ ]

**************************************************************
**************************************************************
START NEW DEFAULT UNIFICATION PROCESS

**************************************************************
STARTING DELTAITERATOR FOR BG, CO : t29, t31
LUB(s) [t27]

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :true
assimilation deltas.next : t31 t27 t31

**************************************************************
SPECIALIZATION
unify between following arguments:
t31 & [ FT26 t1 & [ C *top* ]
        FT27 t1 & [ C *top* ]
        FT28 bool ]
t31 & [ FT26 t1 & [ C *top* ]
        FT27 t1 & [ C *top* ]
        FT28 bool ]


GENERALIZATION
raise the first TFS to the second TFS:
t29 & [ FT27 t5 & [ ]
        FT26 t1 & [ ]]
```

```
t27 & [ FT26 *top*
        FT27 t1 & [ ]]

generalized bg :
t27 & [ FT26 t1 & [ ]
        FT27 t5 & [ ]]
specialized co :
t31 & [ FT26 t1 & [ C *top* ]
        FT27 t1 & [ C *top* ]
        FT28 bool ]

**************************************************************
FOR feat FT28 in set of features only in the specialized co :
new FeatureValuePair (FT28,bool )

**************************************************************
FOR feat FT26 in set of shared features DO
default unify between following arguments:

 new bg argument : t1 & [ ]
 new co argument : t1 & [ C *top* ]

**************************************************************
**************************************************************
START NEW DEFAULT UNIFICATION PROCESS

cover.equivalentTFS(background)
SHORTCUT OF DEFAULT UNIFICATION SUCCEED

new FeatureValuePair(FT26,[t1 & [ C *top* ]])

**************************************************************
FOR feat FT27 in set of shared features DO
default unify between following arguments:

 new bg argument : t5 & [ ]
 new co argument : t1 & [ C *top* ]
```

```
*************************************************************
*************************************************************
START NEW DEFAULT UNIFICATION PROCESS

cover.equivalentTFS(prototypeCo) &&
background.equivalentTFS(prototypeBg) &&
typeHierarchy.subsumesType(typeCo,typeBg)
SHORTCUT OF DEFAULT UNIFICATION SUCCEED

new FeatureValuePair(FT27,
t5 & [ C t27 & [ ]
       D a ])

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :false

END DEFAULT UNIFICATION between following arguments :
t29 & [ FT27 t5 & [ ]
        FT26 t1 & [ ]]
t31 & [ FT26 t1 & [ C *top* ]
        FT27 t1 & [ C *top* ]
        FT28 bool ]
WITH FOLLOWING RESULT :
{t31 & [ FT28 bool
        FT26 t1 & [ C *top* ]
        FT27 t5 & [ C t27 & [ ]
                    D a ]]}
*************************************************************
*************************************************************
new FeatureValuePair(C,
t31 & [ FT28 bool
        FT26 t1 & [ C *top* ]
        FT27 t5 & [ C t27 & [ ]
                    D a ]])

*************************************************************
FOR feat A in set of shared features DO
```

```
default unify between following arguments:

 new bg argument : t20 & [ FT20 d ]
 new co argument : t20 & [ FT20 i ]

**************************************************************
**************************************************************
START NEW DEFAULT UNIFICATION PROCESS

**************************************************************
STARTING DELTAITERATOR FOR BG, CO : t20, t20
LUB(s) [t20]

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :true
assimilation deltas.next : t20 t20 t20

**************************************************************
SPECIALIZATION
unify between following arguments:
t20 & [ FT20 i ]
t20 & [ FT20 a ]


GENERALIZATION
raise the first TFS to the second TFS:
t20 & [ FT20 d ]
t20 & [ FT20 a ]

generalized bg : t20 & [ FT20 d ]

specialized co : t20 & [ FT20 i ]

**************************************************************
FOR feat FT20 in set of shared features DO
default unify between following arguments:
```

```
 new bg argument : d
 new co argument : i

**************************************************************
**************************************************************
START NEW DEFAULT UNIFICATION PROCESS

cover.equivalentTFS(prototypeCo) &&
background.equivalentTFS(prototypeBg) &&
typeHierarchy.hasGLB(typeBg, typeCo)
SHORTCUT OF DEFAULT UNIFICATION SUCCEED

new FeatureValuePair(FT20,[i ])

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :false

END DEFAULT UNIFICATION between following arguments :
t20 & [ FT20 d ]
t20 & [ FT20 i ]
WITH FOLLOWING RESULT :
{t20 & [ FT20 i ]}
**************************************************************
**************************************************************
new FeatureValuePair(A,[t20 & [ FT20 i ]])

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :true
assimilation deltas.next : t5 t7 t12

**************************************************************
SPECIALIZATION
unify between following arguments:
t5 & [ C t27 & [ ]
       D a ]
t12 & [ F *top*
```

```
        C t29 & [ FT27 t5 & [ ]
                  FT26 t1 & [ ]]
        B t28 & [ ]
        D d ]


GENERALIZATION
raise the first TFS to the second TFS:
t10 & [ E *top*
        B t28 & [ ]
        C t29 & [ ]
        A t20 & [ FT20 d ]]
t7 & [ C t29 & [ ]
       B t28 & [ ]]

generalized bg :
t7 & [ C t29 & [ ]
       B t28 & [ ]]
specialized co :
t12 & [ F *top*
        C t29 & [ FT27 t5 & [ ]
                  FT26 t1 & [ ]]
        B t28 & [ ]
        D d ]

**************************************************************
FOR feat F in set of features only in the specialized co :
new FeatureValuePair (F,*top* )

**************************************************************
FOR feat D in set of features only in the specialized co :
new FeatureValuePair (D,d )

**************************************************************
FOR feat C in set of shared features DO
default unify between following arguments:

 new bg argument : t29 & [ ]
```

```
 new co argument :
t29 & [ FT27 t5 & [ ]
        FT26 t1 & [ ]]

**************************************************************
**************************************************************
START NEW DEFAULT UNIFICATION PROCESS

cover.equivalentTFS(background)
SHORTCUT OF DEFAULT UNIFICATION SUCCEED

new FeatureValuePair(C,
t29 & [ FT27 t5 & [ ]
        FT26 t1 & [ ]])

**************************************************************
FOR feat B in set of shared features DO
default unify between following arguments:

 new bg argument : t28 & [ ]
 new co argument : t28 & [ ]

**************************************************************
**************************************************************
START NEW DEFAULT UNIFICATION PROCESS

cover.equivalentTFS(background)
SHORTCUT OF DEFAULT UNIFICATION SUCCEED

new FeatureValuePair(B,[t28 & [ ]])

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :true
assimilation deltas.next : t5 t7 t13

**************************************************************
SPECIALIZATION
```

```
unify between following arguments:
t5 & [ C t27 & [ ]
       D a ]
t13 & [ D f
        C t32 & [ FT27 t8 & [ ]
                  FT26 t9 & [ ]]
        B t31 & [ FT28 bool
                  FT27 t5 & [ ]
                  FT26 t7 & [ ]]]


GENERALIZATION
raise the first TFS to the second TFS:
t10 & [ E *top*
        B t28 & [ ]
        C t29 & [ ]
        A t20 & [ FT20 d ]]
t7 & [ C t29 & [ ]
       B t28 & [ ]]

generalized bg :
t7 & [ C t29 & [ ]
       B t28 & [ ]]
specialized co :
t13 & [ D f
        C t32 & [ FT27 t8 & [ ]
                  FT26 t9 & [ ]]
        B t31 & [ FT28 bool
                  FT27 t5 & [ ]
                  FT26 t7 & [ ]]]

**************************************************************
FOR feat D in set of features only in the specialized co :
new FeatureValuePair (D,f )

**************************************************************
FOR feat C in set of shared features DO
default unify between following arguments:
```

```
 new bg argument : t29 & [ ]
 new co argument :
t32 & [ FT27 t8 & [ ]
        FT26 t9 & [ ]]

**************************************************************
**************************************************************
START NEW DEFAULT UNIFICATION PROCESS

cover.equivalentTFS(prototypeCo) &&
background.equivalentTFS(prototypeBg) &&
typeHierarchy.hasGLB(typeBg, typeCo)
SHORTCUT OF DEFAULT UNIFICATION SUCCEED

new FeatureValuePair(C,[t32 & [ ]])

**************************************************************
FOR feat B in set of shared features DO
default unify between following arguments:

 new bg argument : t28 & [ ]
 new co argument :
t31 & [ FT28 bool
        FT27 t5 & [ ]
        FT26 t7 & [ ]]

**************************************************************
**************************************************************
START NEW DEFAULT UNIFICATION PROCESS

background.equivalentTFS(prototypeBg) &&
typeHierarchy.subsumesType(typeBg, typeCo)
SHORTCUT OF DEFAULT UNIFICATION SUCCEED

new FeatureValuePair(B,
t31 & [ FT28 bool
        FT27 t5 & [ ]
```

```
        FT26 t7 & [ ]])

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :true
assimilation deltas.next : t5 t3 t12
this delta is more general than some already computed one

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :true
assimilation deltas.next : t5 t3 t13
this delta is more general than some already computed one

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :true
assimilation deltas.next : t5 t4 t9
this delta is more general than some already computed one

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :true
assimilation deltas.next : t5 t4 t13
this delta is more general than some already computed one

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :true
assimilation deltas.next : t5 t2 t8
this delta is more general than some already computed one

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :true
assimilation deltas.next : t5 t1 t5
this delta is more general than some already computed one
```

```
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :false

END DEFAULT UNIFICATION between following arguments :
t10 & [ E *top*
        B t28 & [ ]
        C t29 & [ ]
        A t20 & [ FT20 d ]]
t5 & [ C t27 & [ ]
       D a ]
WITH FOLLOWING RESULT :
{t11 & [ D c
        E *top*
        C t31 & [ FT28 bool
                  FT26 t1 & [ C *top* ]
                  FT27 t5 & [ C t27 & [ ]
                              D a ]]
        A t20 & [ FT20 i ]],
t12 & [ F *top*
        D d
        C t29 & [ FT27 t5 & [ ]
                  FT26 t1 & [ ]]
        B t28 & [ ]],
t13 & [ D f
        C t32 & [ ]
        B t31 & [ FT28 bool
                  FT27 t5 & [ ]
                  FT26 t7 & [ ]]]
}
**************************************************************
**************************************************************
new FeatureValueList(FT27,{t11 & [ D c
        E *top*
        C t31 & [ FT28 bool
                  FT26 t1 & [ C *top* ]
                  FT27 t5 & [ C t27 & [ ]
                              D a ]]
```

```
        A t20 & [ FT20 i ]],
t12 & [ F *top*
        D d
        C t29 & [ FT27 t5 & [ ]
                  FT26 t1 & [ ]]
        B t28 & [ ]],
t13 & [ D f
        C t32 & [ ]
        B t31 & [ FT28 bool
                  FT27 t5 & [ ]
                  FT26 t7 & [ ]]]
})

**************************************************************
FOR feat FT26 in set of shared features DO
default unify between following arguments:

 new bg argument : t11 & [ ]
 new co argument : t7 & [ ]

**************************************************************
**************************************************************
START NEW DEFAULT UNIFICATION PROCESS

**************************************************************
STARTING DELTAITERATOR FOR BG, CO : t11, t7
LUB(s) [t1]

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :true
assimilation deltas.next : t7 t6 t10

**************************************************************
SPECIALIZATION
unify between following arguments:
t7 & [ C t29 & [ ]
       B t28 & [ ]]
```

```
t10 & [ E *top*
        B t28 & [ ]
        C t29 & [ ]
        A t20 & [ FT20 d ]]


GENERALIZATION
raise the first TFS to the second TFS:
t11 & [ E *top*
        D c
        C t31 & [ FT26 t1 & [ C *top* ]
                  FT27 t1 & [ C *top* ]
                  FT28 bool ]
        A t20 & [ FT20 i ]]
t6 & [ C *top*
       E *top*
       A t20 & [ FT20 b ]]

generalized bg :
t6 & [ C t31 & [ FT26 t1 & [ C *top* ]
                 FT27 t1 & [ C *top* ]
                 FT28 bool ]
       E *top*
       A t20 & [ FT20 i ]]
specialized co :
t10 & [ E *top*
        B t28 & [ ]
        C t29 & [ ]
        A t20 & [ FT20 d ]]

**************************************************************
FOR feat B in set of features only in the specialized co :
new FeatureValuePair (B,t28 & [ ])

**************************************************************
FOR feat E in set of shared features DO
default unify between following arguments:
```

```
 new bg argument : *top*
 new co argument : *top*

**************************************************************
**************************************************************
START NEW DEFAULT UNIFICATION PROCESS

typeBg == typeHierarchy.TOP_TYPE
SHORTCUT OF DEFAULT UNIFICATION SUCCEED

new FeatureValuePair(E,[*top* ])

**************************************************************
FOR feat C in set of shared features DO
default unify between following arguments:

 new bg argument :
t31 & [ FT26 t1 & [ C *top* ]
        FT27 t1 & [ C *top* ]
        FT28 bool ]
 new co argument : t29 & [ ]

**************************************************************
**************************************************************
START NEW DEFAULT UNIFICATION PROCESS

**************************************************************
STARTING DELTAITERATOR FOR BG, CO : t31, t29
LUB(s) [t27]

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :true
assimilation deltas.next : t29 t28 t32

**************************************************************
SPECIALIZATION
unify between following arguments:
```

```
t29 & [ FT27 t5 & [ ]
        FT26 t1 & [ ]]
t32 & [ FT27 t8 & [ ]
        FT26 t9 & [ ]]


GENERALIZATION
raise the first TFS to the second TFS:
t31 & [ FT26 t1 & [ C *top* ]
        FT27 t1 & [ C *top* ]
        FT28 bool ]
t28 & [ FT27 t1 & [ ]
        FT26 t1 & [ ]]

generalized bg :
t28 & [ FT27 t1 & [ C *top* ]
        FT26 t1 & [ C *top* ]]
specialized co :
t32 & [ FT27 t8 & [ ]
        FT26 t9 & [ ]]

**************************************************************
FOR feat FT27 in set of shared features DO
default unify between following arguments:

 new bg argument : t1 & [ C *top* ]
 new co argument : t8 & [ ]

**************************************************************
**************************************************************
START NEW DEFAULT UNIFICATION PROCESS

cover.equivalentTFS(prototypeCo) &&
background.equivalentTFS(prototypeBg) &&
typeHierarchy.hasGLB(typeBg, typeCo)
SHORTCUT OF DEFAULT UNIFICATION SUCCEED

new FeatureValuePair(FT27,[t8 & [ ]])
```

```
************************************************************
FOR feat FT26 in set of shared features DO
default unify between following arguments:

 new bg argument : t1 & [ C *top* ]
 new co argument : t9 & [ ]

************************************************************
************************************************************
START NEW DEFAULT UNIFICATION PROCESS

cover.equivalentTFS(prototypeCo) &&
background.equivalentTFS(prototypeBg) &&
typeHierarchy.hasGLB(typeBg, typeCo)
SHORTCUT OF DEFAULT UNIFICATION SUCCEED

new FeatureValuePair(FT26,[t9 & [ ]])

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :true
assimilation deltas.next : t29 t27 t29
this delta is more general than some already computed one

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :false

END DEFAULT UNIFICATION between following arguments :
t31 & [ FT26 t1 & [ C *top* ]
        FT27 t1 & [ C *top* ]
        FT28 bool ]
t29 & [ FT27 t5 & [ ]
        FT26 t1 & [ ]]
WITH FOLLOWING RESULT :
{t32 & [ FT27 t8 & [ ]
        FT26 t9 & [ ]]}
```

```
**************************************************************
**************************************************************
new FeatureValuePair(C,
t32 & [ FT27 t8 & [ ]
        FT26 t9 & [ ]])

**************************************************************
FOR feat A in set of shared features DO
default unify between following arguments:

 new bg argument : t20 & [ FT20 i ]
 new co argument : t20 & [ FT20 d ]

**************************************************************
**************************************************************
START NEW DEFAULT UNIFICATION PROCESS

**************************************************************
STARTING DELTAITERATOR FOR BG, CO : t20, t20
LUB(s) [t20]

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :true
assimilation deltas.next : t20 t20 t20

**************************************************************
SPECIALIZATION
unify between following arguments:
t20 & [ FT20 d ]
t20 & [ FT20 a ]


GENERALIZATION
raise the first TFS to the second TFS:
t20 & [ FT20 i ]
t20 & [ FT20 a ]
```

```
generalized bg : t20 & [ FT20 i ]

specialized co : t20 & [ FT20 d ]

**************************************************************
FOR feat FT20 in set of shared features DO
default unify between following arguments:

 new bg argument : i
 new co argument : d

**************************************************************
**************************************************************
START NEW DEFAULT UNIFICATION PROCESS

cover.equivalentTFS(prototypeCo) &&
background.equivalentTFS(prototypeBg) &&
typeHierarchy.subsumesType(typeCo,typeBg)
SHORTCUT OF DEFAULT UNIFICATION SUCCEED

new FeatureValuePair(FT20,[i ])

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :false

END DEFAULT UNIFICATION between following arguments :
t20 & [ FT20 i ]
t20 & [ FT20 d ]
WITH FOLLOWING RESULT :
{t20 & [ FT20 i ]}
**************************************************************
**************************************************************
new FeatureValuePair(A,[t20 & [ FT20 i ]])

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :true
```

```
assimilation deltas.next : t7 t5 t12

***********************************************************
SPECIALIZATION
unify between following arguments:
t7 & [ C t29 & [ ]
       B t28 & [ ]]
t12 & [ F *top*
        C t29 & [ FT27 t5 & [ ]
                  FT26 t1 & [ ]]
        B t28 & [ ]
        D d ]


GENERALIZATION
raise the first TFS to the second TFS:
t11 & [ E *top*
        D c
        C t31 & [ FT26 t1 & [ C *top* ]
                  FT27 t1 & [ C *top* ]
                  FT28 bool ]
        A t20 & [ FT20 i ]]
t5 & [ C t27 & [ ]
       D a ]

generalized bg :
t5 & [ C t31 & [ FT26 t1 & [ C *top* ]
                 FT27 t1 & [ C *top* ]
                 FT28 bool ]
       D c ]
specialized co :
t12 & [ F *top*
        C t29 & [ FT27 t5 & [ ]
                  FT26 t1 & [ ]]
        B t28 & [ ]
        D d ]

***********************************************************
```

```
FOR feat F in set of features only in the specialized co :
new FeatureValuePair (F,*top* )

**************************************************************
FOR feat B in set of features only in the specialized co :
new FeatureValuePair (B,t28 & [ ])

**************************************************************
FOR feat C in set of shared features DO
default unify between following arguments:

 new bg argument :
t31 & [ FT26 t1 & [ C *top* ]
        FT27 t1 & [ C *top* ]
        FT28 bool ]
 new co argument :
t29 & [ FT27 t5 & [ ]
        FT26 t1 & [ ]]

**************************************************************
**************************************************************
START NEW DEFAULT UNIFICATION PROCESS
result is in cache (computation with arguments occured before)
SHORTCUT OF DEFAULT UNIFICATION SUCCEED

new FeatureValuePair(C,
t32 & [ FT27 t8 & [ ]
        FT26 t9 & [ ]])

**************************************************************
FOR feat D in set of shared features DO
default unify between following arguments:

 new bg argument : c
 new co argument : d

**************************************************************
**************************************************************
```

```
START NEW DEFAULT UNIFICATION PROCESS

**************************************************************
STARTING DELTAITERATOR FOR BG, CO : c, d
LUB(s) [a]

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :true
assimilation deltas.next : d a d

!cover.hasFeatures() &&
!(TypeFeatureStructure)typeHierarchy.getPhi(bgfnuttType)).hasFeatures()
continue

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :false

END DEFAULT UNIFICATION between following arguments :
c
d
WITH FOLLOWING RESULT :
{d }
**************************************************************
**************************************************************
new FeatureValuePair(D,[d ])

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :true
assimilation deltas.next : t7 t5 t13

**************************************************************
SPECIALIZATION
unify between following arguments:
t7 & [ C t29 & [ ]
       B t28 & [ ]]
```

```
t13 & [ D f
        C t32 & [ FT27 t8 & [ C t27 & [ ]
                              D c
                              A t20 & [ FT20 g ]]
                  FT26 t9 & [ C t27 & [ ]
                              F *top*
                              D b ]]
        B t31 & [ FT28 bool
                  FT27 t5 & [ ]
                  FT26 t7 & [ ]]]


GENERALIZATION
raise the first TFS to the second TFS:
t11 & [ E *top*
        D c
        C t31 & [ FT26 t1 & [ C *top* ]
                  FT27 t1 & [ C *top* ]
                  FT28 bool ]
        A t20 & [ FT20 i ]]
t5 & [ C t27 & [ ]
       D a ]

generalized bg :
t5 & [ C t31 & [ FT26 t1 & [ C *top* ]
                 FT27 t1 & [ C *top* ]
                 FT28 bool ]
       D c ]
specialized co :
t13 & [ D f
        C t32 & [ FT27 t8 & [ C t27 & [ ]
                              D c
                              A t20 & [ FT20 g ]]
                  FT26 t9 & [ C t27 & [ ]
                              F *top*
                              D b ]]
        B t31 & [ FT28 bool
                  FT27 t5 & [ ]
```

```
                  FT26 t7 & [ ]]]

**************************************************************
FOR feat B in set of features only in the specialized co :
new FeatureValuePair (B,t31 & [ FT28 bool
        FT27 t5 & [ ]
        FT26 t7 & [ ]])

**************************************************************
FOR feat D in set of shared features DO
default unify between following arguments:

 new bg argument : c
 new co argument : f

**************************************************************
**************************************************************
START NEW DEFAULT UNIFICATION PROCESS

cover.equivalentTFS(prototypeCo) &&
background.equivalentTFS(prototypeBg) &&
typeHierarchy.hasGLB(typeBg, typeCo)
SHORTCUT OF DEFAULT UNIFICATION SUCCEED

new FeatureValuePair(D,[f ])

**************************************************************
FOR feat C in set of shared features DO
default unify between following arguments:

 new bg argument :
t31 & [ FT26 t1 & [ C *top* ]
        FT27 t1 & [ C *top* ]
        FT28 bool ]
 new co argument :
t32 & [ FT27 t8 & [ C t27 & [ ]
                    D c
                    A t20 & [ FT20 g ]]
```

```
        FT26 t9 & [ C t27 & [ ]
                    F *top*
                    D b ]]

**************************************************************
**************************************************************
START NEW DEFAULT UNIFICATION PROCESS

**************************************************************
STARTING DELTAITERATOR FOR BG, CO : t31, t32
LUB(s) [t28]

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :true
assimilation deltas.next : t32 t28 t32

**************************************************************
SPECIALIZATION
unify between following arguments:
t32 & [ FT27 t8 & [ C t27 & [ ]
                    D c
                    A t20 & [ FT20 g ]]
        FT26 t9 & [ C t27 & [ ]
                    F *top*
                    D b ]]
t32 & [ FT27 t8 & [ C t27 & [ ]
                    D c
                    A t20 & [ FT20 g ]]
        FT26 t9 & [ C t27 & [ ]
                    F *top*
                    D b ]]


GENERALIZATION
raise the first TFS to the second TFS:
t31 & [ FT26 t1 & [ C *top* ]
        FT27 t1 & [ C *top* ]
```

```
        FT28 bool ]
t28 & [ FT27 t1 & [ ]
        FT26 t1 & [ ]]

generalized bg :
t28 & [ FT27 t1 & [ C *top* ]
        FT26 t1 & [ C *top* ]]
specialized co :
t32 & [ FT27 t8 & [ C t27 & [ ]
                    D c
                    A t20 & [ FT20 g ]]
        FT26 t9 & [ C t27 & [ ]
                    F *top*
                    D b ]]

*************************************************************
FOR feat FT27 in set of shared features DO
default unify between following arguments:

 new bg argument : t1 & [ C *top* ]
 new co argument :
t8 & [ C t27 & [ ]
       D c
       A t20 & [ FT20 g ]]

*************************************************************
*************************************************************
START NEW DEFAULT UNIFICATION PROCESS
result is in cache (computation with arguments occured before)
SHORTCUT OF DEFAULT UNIFICATION SUCCEED

new FeatureValuePair(FT27,[t8 & [ ]])

*************************************************************
FOR feat FT26 in set of shared features DO
default unify between following arguments:

 new bg argument : t1 & [ C *top* ]
```

```
 new co argument :
t9 & [ C t27 & [ ]
       F *top*
       D b ]

**************************************************************
**************************************************************
START NEW DEFAULT UNIFICATION PROCESS
result is in cache (computation with arguments occured before)
SHORTCUT OF DEFAULT UNIFICATION SUCCEED

new FeatureValuePair(FT26,[t9 & [ ]])

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :false

END DEFAULT UNIFICATION between following arguments :
t31 & [ FT26 t1 & [ C *top* ]
        FT27 t1 & [ C *top* ]
        FT28 bool ]
t32 & [ FT27 t8 & [ C t27 & [ ]
                    D c
                    A t20 & [ FT20 g ]]
        FT26 t9 & [ C t27 & [ ]
                    F *top*
                    D b ]]
WITH FOLLOWING RESULT :
{t32 & [ FT27 t8 & [ ]
        FT26 t9 & [ ]]}
**************************************************************
**************************************************************
new FeatureValuePair(C,
t32 & [ FT27 t8 & [ ]
        FT26 t9 & [ ]])

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
```

```
hasNext? :true
assimilation deltas.next : t7 t2 t10
this delta is more general than some already computed one

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :true
assimilation deltas.next : t7 t1 t7
this delta is more general than some already computed one

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :false

END DEFAULT UNIFICATION between following arguments :
t11 & [ E *top*
        D c
        C t31 & [ FT26 t1 & [ C *top* ]
                  FT27 t1 & [ C *top* ]
                  FT28 bool ]
        A t20 & [ FT20 i ]]
t7 & [ C t29 & [ ]
       B t28 & [ ]]
WITH FOLLOWING RESULT :
{t10 & [ B t28 & [ ]
        E *top*
        C t32 & [ FT27 t8 & [ ]
                  FT26 t9 & [ ]]
        A t20 & [ FT20 i ]],
t12 & [ F *top*
        B t28 & [ ]
        C t32 & [ FT27 t8 & [ ]
                  FT26 t9 & [ ]]
        D d ],
t13 & [ B t31 & [ FT28 bool
                  FT27 t5 & [ ]
                  FT26 t7 & [ ]]
        D f
```

```
        C t32 & [ FT27 t8 & [ ]
                  FT26 t9 & [ ]]]
}
**************************************************************
**************************************************************
new FeatureValueList(FT26,{t10 & [ B t28 & [ ]
        E *top*
        C t32 & [ FT27 t8 & [ ]
                  FT26 t9 & [ ]]
        A t20 & [ FT20 i ]],
t12 & [ F *top*
        B t28 & [ ]
        C t32 & [ FT27 t8 & [ ]
                  FT26 t9 & [ ]]
        D d ],
t13 & [ B t31 & [ FT28 bool
                  FT27 t5 & [ ]
                  FT26 t7 & [ ]]
        D f
        C t32 & [ FT27 t8 & [ ]
                  FT26 t9 & [ ]]]
})

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :false

END DEFAULT UNIFICATION between following arguments :
t31 & [ FT26 t11 & [ ]
        FT27 t10 & [ ]
        FT28 bool ]
t31 & [ FT28 bool
        FT27 t5 & [ C t27 & [ ]
                    D a ]
        FT26 t7 & [ C t29 & [ ]
                    B t28 & [ ]]]
WITH FOLLOWING RESULT :
{t31 & [ FT28 bool
```

```
        FT27  { t11 & [ D c
                        E *top*
                        C t31 & [ FT28 bool
                                  FT26 t1 & [ C *top* ]
                                  FT27 t5 & [ C t27 & [ ]
                                              D a ]]
                        A t20 & [ FT20 i ]],
                t12 & [ F *top*
                        D d
                        C t29 & [ FT27 t5 & [ ]
                                  FT26 t1 & [ ]]
                        B t28 & [ ]],
                t13 & [ D f
                        C t32 & [ ]
                        B t31 & [ FT28 bool
                                  FT27 t5 & [ ]
                                  FT26 t7 & [ ]]] }
        FT26  { t10 & [ B t28 & [ ]
                        E *top*
                        C t32 & [ FT27 t8 & [ ]
                                  FT26 t9 & [ ]]
                        A t20 & [ FT20 i ]],
                t12 & [ F *top*
                        B t28 & [ ]
                        C t32 & [ FT27 t8 & [ ]
                                  FT26 t9 & [ ]]
                        D d ],
                t13 & [ B t31 & [ FT28 bool
                                  FT27 t5 & [ ]
                                  FT26 t7 & [ ]]
                        D f
                        C t32 & [ FT27 t8 & [ ]
                                  FT26 t9 & [ ]]] } ]}
**************************************************************
**************************************************************
new FeatureValuePair(B,
t31 & [ FT28 bool
        FT27  { t11 & [ D c
```

```
                    E *top*
                    C t31 & [ FT28 bool
                              FT26 t1 & [ C *top* ]
                              FT27 t5 & [ C t27 & [ ]
                                          D a ]]
                    A t20 & [ FT20 i ]],
            t12 & [ F *top*
                    D d
                    C t29 & [ FT27 t5 & [ ]
                              FT26 t1 & [ ]]
                    B t28 & [ ]],
            t13 & [ D f
                    C t32 & [ ]
                    B t31 & [ FT28 bool
                              FT27 t5 & [ ]
                              FT26 t7 & [ ]]] }
    FT26  { t10 & [ B t28 & [ ]
                    E *top*
                    C t32 & [ FT27 t8 & [ ]
                              FT26 t9 & [ ]]
                    A t20 & [ FT20 i ]],
            t12 & [ F *top*
                    B t28 & [ ]
                    C t32 & [ FT27 t8 & [ ]
                              FT26 t9 & [ ]]
                    D d ],
            t13 & [ B t31 & [ FT28 bool
                              FT27 t5 & [ ]
                              FT26 t7 & [ ]]
                    D f
                    C t32 & [ FT27 t8 & [ ]
                              FT26 t9 & [ ]]] } ])

**************************************************************
FOR feat C in set of shared features DO
default unify between following arguments:

 new bg argument :
```

```
t33 & [ FT26 t11 & [ ]
        FT27 t9 & [ ]]
 new co argument : t34 & [ ]

**************************************************************
**************************************************************
START NEW DEFAULT UNIFICATION PROCESS

**************************************************************
STARTING DELTAITERATOR FOR BG, CO : t33, t34
LUB(s) [t29]

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :true
assimilation deltas.next : t34 t29 t34

**************************************************************
SPECIALIZATION
unify between following arguments:
t34 & [ FT27 t8 & [ ]
        FT26 t12 & [ ]]
t34 & [ FT27 t8 & [ ]
        FT26 t12 & [ ]]


GENERALIZATION
raise the first TFS to the second TFS:
t33 & [ FT26 t11 & [ ]
        FT27 t9 & [ ]]
t29 & [ FT27 t5 & [ ]
        FT26 t1 & [ ]]

generalized bg :
t29 & [ FT27 t9 & [ ]
        FT26 t11 & [ ]]
specialized co :
t34 & [ FT27 t8 & [ ]
```

```
        FT26 t12 & [ ]]

*************************************************************
FOR feat FT27 in set of shared features DO
default unify between following arguments:

 new bg argument : t9 & [ ]
 new co argument : t8 & [ ]

*************************************************************
*************************************************************
START NEW DEFAULT UNIFICATION PROCESS

*************************************************************
STARTING DELTAITERATOR FOR BG, CO : t9, t8
LUB(s) [t5]

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :true
assimilation deltas.next : t8 t5 t8

*************************************************************
SPECIALIZATION
unify between following arguments:
t8 & [ C t27 & [ ]
       D c
       A t20 & [ FT20 g ]]
t8 & [ C t27 & [ ]
       D c
       A t20 & [ FT20 g ]]


GENERALIZATION
raise the first TFS to the second TFS:
t9 & [ C t27 & [ ]
       F *top*
       D b ]
```

```
t5 & [ C t27 & [ ]
       D a ]

generalized bg :
t5 & [ C t27 & [ ]
       D b ]
specialized co :
t8 & [ C t27 & [ ]
       D c
       A t20 & [ FT20 g ]]

*************************************************************
FOR feat A in set of features only in the specialized co :
new FeatureValuePair (A,t20 & [ FT20 g ])

*************************************************************
FOR feat C in set of shared features DO
default unify between following arguments:

 new bg argument : t27 & [ ]
 new co argument : t27 & [ ]

*************************************************************
*************************************************************
START NEW DEFAULT UNIFICATION PROCESS

cover.equivalentTFS(background)
SHORTCUT OF DEFAULT UNIFICATION SUCCEED

new FeatureValuePair(C,[t27 & [ ]])

*************************************************************
FOR feat D in set of shared features DO
default unify between following arguments:

 new bg argument : b
 new co argument : c
```

```
**************************************************************
**************************************************************
START NEW DEFAULT UNIFICATION PROCESS

cover.equivalentTFS(prototypeCo) &&
background.equivalentTFS(prototypeBg) &&
typeHierarchy.hasGLB(typeBg, typeCo)
SHORTCUT OF DEFAULT UNIFICATION SUCCEED

new FeatureValuePair(D,[e ])

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :false

END DEFAULT UNIFICATION between following arguments :
t9 & [ C t27 & [ ]
       F *top*
       D b ]
t8 & [ C t27 & [ ]
       D c
       A t20 & [ FT20 g ]]
WITH FOLLOWING RESULT :
{t8 & [ A t20 & [ FT20 g ]
       C t27 & [ ]
       D e ]}
**************************************************************
**************************************************************
new FeatureValuePair(FT27,
t8 & [ A t20 & [ FT20 g ]
       C t27 & [ ]
       D e ])

**************************************************************
FOR feat FT26 in set of shared features DO
default unify between following arguments:

 new bg argument : t11 & [ ]
```

```
 new co argument : t12 & [ ]

*************************************************************
*************************************************************
START NEW DEFAULT UNIFICATION PROCESS

*************************************************************
STARTING DELTAITERATOR FOR BG, CO : t11, t12
LUB(s) [t5]

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :true
assimilation deltas.next : t12 t5 t12

*************************************************************
SPECIALIZATION
unify between following arguments:
t12 & [ F *top*
        C t29 & [ FT27 t5 & [ ]
                  FT26 t1 & [ ]]
        B t28 & [ ]
        D d ]
t12 & [ F *top*
        C t29 & [ FT27 t5 & [ ]
                  FT26 t1 & [ ]]
        B t28 & [ ]
        D d ]


GENERALIZATION
raise the first TFS to the second TFS:
t11 & [ E *top*
        D c
        C t31 & [ FT26 t1 & [ C *top* ]
                  FT27 t1 & [ C *top* ]
                  FT28 bool ]
        A t20 & [ FT20 i ]]
```

```
t5 & [ C t27 & [ ]
       D a ]

generalized bg :
t5 & [ C t31 & [ FT26 t1 & [ C *top* ]
                 FT27 t1 & [ C *top* ]
                 FT28 bool ]
       D c ]
specialized co :
t12 & [ F *top*
        C t29 & [ FT27 t5 & [ ]
                  FT26 t1 & [ ]]
        B t28 & [ ]
        D d ]

*************************************************************
FOR feat F in set of features only in the specialized co :
new FeatureValuePair (F,*top* )

*************************************************************
FOR feat B in set of features only in the specialized co :
new FeatureValuePair (B,t28 & [ ])

*************************************************************
FOR feat C in set of shared features DO
default unify between following arguments:

 new bg argument :
t31 & [ FT26 t1 & [ C *top* ]
        FT27 t1 & [ C *top* ]
        FT28 bool ]
 new co argument :
t29 & [ FT27 t5 & [ ]
        FT26 t1 & [ ]]

*************************************************************
*************************************************************
START NEW DEFAULT UNIFICATION PROCESS
```

```
result is in cache (computation with arguments occured before)
SHORTCUT OF DEFAULT UNIFICATION SUCCEED

new FeatureValuePair(C,
t32 & [ FT27 t8 & [ ]
        FT26 t9 & [ ]])

**************************************************************
FOR feat D in set of shared features DO
default unify between following arguments:

 new bg argument : c
 new co argument : d

**************************************************************
**************************************************************
START NEW DEFAULT UNIFICATION PROCESS
result is in cache (computation with arguments occured before)
SHORTCUT OF DEFAULT UNIFICATION SUCCEED

new FeatureValuePair(D,[d ])

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :false

END DEFAULT UNIFICATION between following arguments :
t11 & [ E *top*
        D c
        C t31 & [ FT26 t1 & [ C *top* ]
                  FT27 t1 & [ C *top* ]
                  FT28 bool ]
        A t20 & [ FT20 i ]]
t12 & [ F *top*
        C t29 & [ FT27 t5 & [ ]
                  FT26 t1 & [ ]]
        B t28 & [ ]
        D d ]
```

```
WITH FOLLOWING RESULT :
{t12 & [ F *top*
        B t28 & [ ]
        C t32 & [ FT27 t8 & [ ]
                  FT26 t9 & [ ]]
        D d ]}
**************************************************************
**************************************************************
new FeatureValuePair(FT26,
t12 & [ F *top*
        B t28 & [ ]
        C t32 & [ FT27 t8 & [ ]
                  FT26 t9 & [ ]]
        D d ])

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :false

END DEFAULT UNIFICATION between following arguments :
t33 & [ FT26 t11 & [ ]
        FT27 t9 & [ ]]
t34 & [ FT27 t8 & [ ]
        FT26 t12 & [ ]]
WITH FOLLOWING RESULT :
{t34 & [ FT27 t8 & [ A t20 & [ FT20 g ]
                    C t27 & [ ]
                    D e ]
        FT26 t12 & [ F *top*
                     B t28 & [ ]
                     C t32 & [ FT27 t8 & [ ]
                               FT26 t9 & [ ]]
                     D d ]]}
**************************************************************
**************************************************************
new FeatureValuePair(C,
t34 & [ FT27 t8 & [ A t20 & [ FT20 g ]
                    C t27 & [ ]
```

```
                  D e ]
        FT26 t12 & [ F *top*
                     B t28 & [ ]
                     C t32 & [ FT27 t8 & [ ]
                               FT26 t9 & [ ]]
                     D d ]])

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :true
assimilation deltas.next : t5 t3 t12
Warning : the unification between cover and the prototype structure
of the GLB(cover,bg') already failed before

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :true
assimilation deltas.next : t5 t3 t13
this delta is more general than some already computed one

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :true
assimilation deltas.next : t5 t4 t9

**************************************************************
SPECIALIZATION
unify between following arguments:
t5 & [ C t30 & [ FT27 t2 & [ ]
                 FT26 t7 & [ ]]
       D c ]
t9 & [ C t27 & [ ]
       F *top*
       D b ]


GENERALIZATION
raise the first TFS to the second TFS:
```

```
t10 & [ A t22 & [ FT20 d
                  FT21 d ]
        B t31 & [ FT26 t11 & [ ]
                  FT27 t10 & [ ]
                  FT28 bool ]
        C t33 & [ FT26 t11 & [ ]
                  FT27 t9 & [ ]]
        E *top* ]
t4 & [ C t27 & [ ]]

generalized bg : t4 & [ C t33 & [ FT26 t11 & [ ]
                 FT27 t9 & [ ]]]

specialized co :
t9 & [ F *top*
       C t30 & [ ]
       D e ]

**************************************************************
FOR feat F in set of features only in the specialized co :
new FeatureValuePair (F,*top* )

**************************************************************
FOR feat D in set of features only in the specialized co :
new FeatureValuePair (D,e )

**************************************************************
FOR feat C in set of shared features DO
default unify between following arguments:

 new bg argument :
t33 & [ FT26 t11 & [ ]
        FT27 t9 & [ ]]
 new co argument : t30 & [ ]

**************************************************************
**************************************************************
START NEW DEFAULT UNIFICATION PROCESS
```

```
*************************************************************
STARTING DELTAITERATOR FOR BG, CO : t33, t30
LUB(s) [t27]

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :true
assimilation deltas.next : t30 t29 t34

*************************************************************
SPECIALIZATION
unify between following arguments:
t30 & [ FT27 t2 & [ ]
        FT26 t7 & [ ]]
t34 & [ FT27 t8 & [ ]
        FT26 t12 & [ ]]


GENERALIZATION
raise the first TFS to the second TFS:
t33 & [ FT26 t11 & [ ]
        FT27 t9 & [ ]]
t29 & [ FT27 t5 & [ ]
        FT26 t1 & [ ]]

generalized bg :
t29 & [ FT27 t9 & [ ]
        FT26 t11 & [ ]]
specialized co :
t34 & [ FT27 t8 & [ ]
        FT26 t12 & [ ]]

*************************************************************
FOR feat FT27 in set of shared features DO
default unify between following arguments:

 new bg argument : t9 & [ ]
```

```
 new co argument : t8 & [ ]

**************************************************************
**************************************************************
START NEW DEFAULT UNIFICATION PROCESS
result is in cache (computation with arguments occured before)
SHORTCUT OF DEFAULT UNIFICATION SUCCEED

new FeatureValuePair(FT27,
t8 & [ A t20 & [ FT20 g ]
       C t27 & [ ]
       D e ])

**************************************************************
FOR feat FT26 in set of shared features DO
default unify between following arguments:

 new bg argument : t11 & [ ]
 new co argument : t12 & [ ]

**************************************************************
**************************************************************
START NEW DEFAULT UNIFICATION PROCESS
result is in cache (computation with arguments occured before)
SHORTCUT OF DEFAULT UNIFICATION SUCCEED

new FeatureValuePair(FT26,
t12 & [ F *top*
        B t28 & [ ]
        C t32 & [ FT27 t8 & [ ]
                  FT26 t9 & [ ]]
        D d ])

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :true
assimilation deltas.next : t30 t27 t30
this delta is more general than some already computed one
```

```
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :false

END DEFAULT UNIFICATION between following arguments :
t33 & [ FT26 t11 & [ ]
        FT27 t9 & [ ]]
t30 & [ FT27 t2 & [ ]
        FT26 t7 & [ ]]
WITH FOLLOWING RESULT :
{t34 & [ FT27 t8 & [ A t20 & [ FT20 g ]
                    C t27 & [ ]
                    D e ]
        FT26 t12 & [ F *top*
                     B t28 & [ ]
                     C t32 & [ FT27 t8 & [ ]
                               FT26 t9 & [ ]]
                     D d ]]}
**************************************************************
**************************************************************
new FeatureValuePair(C,
t34 & [ FT27 t8 & [ A t20 & [ FT20 g ]
                    C t27 & [ ]
                    D e ]
        FT26 t12 & [ F *top*
                     B t28 & [ ]
                     C t32 & [ FT27 t8 & [ ]
                               FT26 t9 & [ ]]
                     D d ]])

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :true
assimilation deltas.next : t5 t4 t13
this delta is more general than some already computed one

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
```

```
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :true
assimilation deltas.next : t5 t2 t8

**************************************************************
SPECIALIZATION
unify between following arguments:
t5 & [ C t30 & [ FT27 t2 & [ ]
                 FT26 t7 & [ ]]
       D c ]
t8 & [ C t27 & [ ]
       D c
       A t20 & [ FT20 g ]]


GENERALIZATION
raise the first TFS to the second TFS:
t10 & [ A t22 & [ FT20 d
                  FT21 d ]
        B t31 & [ FT26 t11 & [ ]
                  FT27 t10 & [ ]
                  FT28 bool ]
        C t33 & [ FT26 t11 & [ ]
                  FT27 t9 & [ ]]
        E *top* ]
t2 & [ C *top*
       A t19 ]

generalized bg :
t2 & [ C t33 & [ FT26 t11 & [ ]
                 FT27 t9 & [ ]]
       A t22 & [ FT20 d
                 FT21 d ]]
specialized co :
t8 & [ A t20 & [ FT20 g ]
       C t30 & [ ]
       D c ]
```

```
*************************************************************
FOR feat D in set of features only in the specialized co :
new FeatureValuePair (D,c )

*************************************************************
FOR feat A in set of shared features DO
default unify between following arguments:

 new bg argument :
t22 & [ FT20 d
        FT21 d ]
 new co argument : t20 & [ FT20 g ]

*************************************************************
*************************************************************
START NEW DEFAULT UNIFICATION PROCESS

*************************************************************
STARTING DELTAITERATOR FOR BG, CO : t22, t20
LUB(s) [t20]

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :true
assimilation deltas.next : t20 t22 t22

*************************************************************
SPECIALIZATION
unify between following arguments:
t20 & [ FT20 g ]
t22 & [ FT21 b
        FT20 d ]


GENERALIZATION
raise the first TFS to the second TFS:
t22 & [ FT20 d
        FT21 d ]
```

```
t22 & [ FT21 b
        FT20 d ]

generalized bg :
t22 & [ FT21 d
        FT20 d ]
specialized co :
t22 & [ FT21 b
        FT20  { h ,
                i  } ]

**************************************************************
FOR feat FT21 in set of shared features DO
default unify between following arguments:

 new bg argument : d
 new co argument : b

**************************************************************
**************************************************************
START NEW DEFAULT UNIFICATION PROCESS

cover.equivalentTFS(prototypeCo) &&
background.equivalentTFS(prototypeBg) &&
typeHierarchy.subsumesType(typeCo,typeBg)
SHORTCUT OF DEFAULT UNIFICATION SUCCEED

new FeatureValuePair(FT21,[d ])

**************************************************************
FOR feat FT20 in set of shared features DO
default unify between following arguments:

 new bg argument: d
 new co argument: [h , i ]

**************************************************************
**************************************************************
```

```
START NEW DEFAULT UNIFICATION PROCESS

cover.equivalentTFS(prototypeCo) &&
background.equivalentTFS(prototypeBg) &&
typeHierarchy.hasGLB(typeBg, typeCo)
SHORTCUT OF DEFAULT UNIFICATION SUCCEED

**************************************************************
**************************************************************
START NEW DEFAULT UNIFICATION PROCESS
result is in cache (computation with arguments occured before)
SHORTCUT OF DEFAULT UNIFICATION SUCCEED

new FeatureValueList(FT20,[h , i ])

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :true
assimilation deltas.next : t20 t21 t22
this delta is more general than some already computed one

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :true
assimilation deltas.next : t20 t20 t20
this delta is more general than some already computed one

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :false

END DEFAULT UNIFICATION between following arguments :
t22 & [ FT20 d
        FT21 d ]
t20 & [ FT20 g ]
WITH FOLLOWING RESULT :
{t22 & [ FT21 d
        FT20  { h ,
```

```
                i  } ]}
*************************************************************
*************************************************************
new FeatureValuePair(A,
t22 & [ FT21 d
        FT20  { h ,
                i  } ])

*************************************************************
FOR feat C in set of shared features DO
default unify between following arguments:

 new bg argument :
t33 & [ FT26 t11 & [ ]
        FT27 t9 & [ ]]
 new co argument : t30 & [ ]

*************************************************************
*************************************************************
START NEW DEFAULT UNIFICATION PROCESS
result is in cache (computation with arguments occured before)
SHORTCUT OF DEFAULT UNIFICATION SUCCEED

new FeatureValuePair(C,
t34 & [ FT27 t8 & [ A t20 & [ FT20 g ]
                    C t27 & [ ]
                    D e ]
        FT26 t12 & [ F *top*
                     B t28 & [ ]
                     C t32 & [ FT27 t8 & [ ]
                               FT26 t9 & [ ]]
                     D d ]])

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :true
assimilation deltas.next : t5 t1 t5
this delta is more general than some already computed one
```

```
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXT QUERY TO THE DELTA ITERATOR
hasNext? :false

END DEFAULT UNIFICATION between following arguments :
t10 & [ A t22 & [ FT20 d
                  FT21 d ]
        B t31 & [ FT26 t11 & [ ]
                  FT27 t10 & [ ]
                  FT28 bool ]
        C t33 & [ FT26 t11 & [ ]
                  FT27 t9 & [ ]]
        E *top* ]
t5 & [ C t30 & [ FT27 t2 & [ ]
                 FT26 t7 & [ ]]
       D c ]
WITH FOLLOWING RESULT :
{t13 & [ D f
        B t31 & [ FT28 bool
                  FT27  { t11 & [ D c
                                  E *top*
                                  C t31 & [ FT28 bool
                                            FT26 t1 & [ C *top* ]
                                            FT27 t5 & [ C t27 & [ ]
                                                        D a ]]
                                  A t20 & [ FT20 i ]],
                          t12 & [ F *top*
                                  D d
                                  C t29 & [ FT27 t5 & [ ]
                                            FT26 t1 & [ ]]
                                  B t28 & [ ]],
                          t13 & [ D f
                                  C t32 & [ ]
                                  B t31 & [ FT28 bool
                                            FT27 t5 & [ ]
                                            FT26 t7 & [ ]]] }
                  FT26  { t10 & [ B t28 & [ ]
```

```
                                   E *top*
                                   C t32 & [ FT27 t8 & [ ]
                                             FT26 t9 & [ ]]
                                   A t20 & [ FT20 i ]],
                           t12 & [ F *top*
                                   B t28 & [ ]
                                   C t32 & [ FT27 t8 & [ ]
                                             FT26 t9 & [ ]]
                                   D d ],
                           t13 & [ B t31 & [ FT28 bool
                                             FT27 t5 & [ ]
                                             FT26 t7 & [ ]]
                                   D f
                                   C t32 & [ FT27 t8 & [ ]
                                             FT26 t9 & [ ]]] } ]
        C t34 & [ FT27 t8 & [ A t20 & [ FT20 g ]
                              C t27 & [ ]
                              D e ]
                  FT26 t12 & [ F *top*
                               B t28 & [ ]
                               C t32 & [ FT27 t8 & [ ]
                                         FT26 t9 & [ ]]
                               D d ]]],
t9 & [ F *top*
       D e
       C t34 & [ FT27 t8 & [ A t20 & [ FT20 g ]
                             C t27 & [ ]
                             D e ]
                 FT26 t12 & [ F *top*
                              B t28 & [ ]
                              C t32 & [ FT27 t8 & [ ]
                                        FT26 t9 & [ ]]
                              D d ]]],
t8 & [ D c
       A t22 & [ FT21 d
                 FT20  { h ,
                         i  } ]
       C t34 & [ FT27 t8 & [ A t20 & [ FT20 g ]
```

```
                              C t27 & [ ]
                              D e ]
                 FT26 t12 & [ F *top*
                               B t28 & [ ]
                               C t32 & [ FT27 t8 & [ ]
                                         FT26 t9 & [ ]]
                               D d ]]]
}
**************************************************************
**************************************************************
Processing time :0.089 sec
```

References

Krieger, H.-U., & Schäfer, U. *Efficient parameterizable type expansion for typed feature formalisms* (Tech. Rep. No. RR-95-18).

Aït-Kaci, H. (1993). An introduction to life-programming with logic, inheritance, functions, and equations. In *Ilps '93: Proceedings of the 1993 international symposium on logic programming* (pp. 52–68). Cambridge, MA, USA: MIT Press.

Aït-Kaci, H., Boyer, R., Lincoln, P., & Nasr, R. (1989). Efficient implementation of lattice operations. *ACM Trans. Program. Lang. Syst.*, *11*(1), 115–146.

Aït-Kaci, H., & Nasr, R. (1986). Login: A logic programming language with built-in inheritance. *J. Log. Program.*, *3*(3), 185–215.

Alexandersson, J., & Becker, T. (2003, February). The formal foundations underlying overlay. In *Proceedings of the* 5th *international workshop on computational semantics (iwcs-5).* Tilburg, The Netherlands.

Alexandersson, J., & Becker, T. (2004). Default Unification for Discourse Modelling. In H. Bunt & R. Muskens (Eds.), *Computing meaning* (Vol. 3). Dordrecht: Kluwer Academic Publishers. (Forthcoming)

Alexandersson, J., & Becker, T. (2007). Efficient computation of overlay for multiple inheritance hierarchies in discourse modeling. In H. Bunt & R. Muskens (Eds.), (Vol. 3, pp. pp.423–455). Dordrecht:Kluwer.

Alexandersson, J., Becker, T., & Pfleger, N. (2004, September). Scoring for overlay based on informational distance. In *Konvens-04* (pp. 1–4). Vienna, Austria.

B. Davey, A. H. P. (1990). *Introduction to lattices and order.* Cambridge University Press.

Bommel, M. F. van, & Beck, T. J. (1999). Incremental encoding of multiple inheritance hierarchies. In *Cikm '99: Proceedings of the eighth international conference on information and knowledge management* (pp. 507–513). New York, NY, USA: ACM Press.

Bouma, G. (1990). Defaults in unification grammar. In *Proceedings of the 28th annual meeting on association for computational linguistics* (pp. 165–172). Morristown, NJ, USA: Association for Computational Linguistics.

Bouma, G. (1992). Feature structures and nonmonotonicity. *Computational Linguistics*, *18*(2), 183–203.

Callmeier, U. (2001). *Efficient parsing with large-scale unification grammars.* Unpublished master's thesis, Saarland University.

Carpenter, B. (1992). *The logic of typed feature structures.* Cambridge, England: Cambridge University Press.

Carpenter, B. (1993). Skeptical and Credulous Default Unification with Application to Templates and Inheritance. In A. C. E. J. Briscoe & V. de Paiva (Eds.), *Inheritance, Defaults and the Lexicon* (pp. 13–37). Cambridge, England: Cambridge University Press.

Carpenter, B., & Penn, G. (1998). *Ale - the attribute logic engine - user's guide version 3.2.1.*

Copestake, A. (1992). *The Representation of Lexical Semantic Information.* Doctoral dissertation, University of Sussex.

Copestake, A. (1996). *Inheritance in lexical representation.*

Copestake, A. (2002). *Implementing Typed Feature Structure Grammars* (No. 110). CSLI Publications.

Daelemans, W., Smedt, K. D., & Gazdar, G. (1992). Inheritance in natural language processing. *Comput. Linguist.*, *18*(2), 205–218.

Duchier, D. (2003, September). *Configuration of labeled trees under lexicalized constraints and principles.*

Emele, M. C., & Zajac, R. (1990). Typed unification grammars. In *Proceedings of the 13th conference on computational linguistics* (pp. 293–298). Morristown, NJ, USA: Association for Computational Linguistics.

Fraser, N. M., & Hudson, R. A. (1992). Inheritance in word grammar. *Computational Linguistics*, *18*(2), 133–158.

Gazdar, G. (1987). Linguistic applications of default inheritance mechanisms. In P. Whitelock, M. M. Wood, H. L. Somers, R. Johnson, & P. Bennett (Eds.), *Linguistic theory and computer applications* (pp. 37–67). London: Academic Press.

Gerdemann, D. (1995a). Open and closed world types in NLP systems. In *Proceedings of 5. fachtagung der sektion computerlinguistik der dgfs* (pp. 25–30). Duesseldorf.

Gerdemann, D. (1995b). *Term encoding of typed feature structures.*

Gerdemann, D., & King, P. J. (1994). The correct and efficient implementation of appropriateness specifications for typed feature structures. In *Proceedings of the 15th conference on computational linguistics* (pp. 956–960). Morristown, NJ, USA: Association for Computational Linguistics.

Götz, T. (1993). *A normal form for typed feature structures.* Unpublished master's thesis, Universität T̈bingen, Germany.

Grover, C., Brew, C., Manandhar, S., & Moens, M. (1994). Priority union and generalization in discourse grammars. In *32nd. annual meeting of the association for computational linguistics* (pp. 17 – 24). Las Cruces, NM: Association for Computational Linguistics.

Halliday, M. A., & Matthiessen, C. M. (2004). *An introduction to functional grammar.* Arnold Publishers.

Kaplan, R. M. (1987). Three seductions of computational psycholinguistics. In D. et al. (Ed.), (chap. Linguistic Theory and Computer Applications). London Academic Press.

Krieger, H.-U. (1994a). *TDL—a type description language for HPSG part i* (Tech. Rep.). *DFKI*.

Krieger, H.-U. (1994b). *TDL—a type description language for HPSG part ii* (Tech. Rep.). *DFKI*.

Krieger, H.-U. (1995). *TDL*—a type description language for constraint-based grammars. foundations, implementation, and applications. (Doctoral dissertation, Universität des Saarlandes, Department of Computer Science). *Saarbrücken Dissertations in Computational Linguistics and Language Technology.*

Lascarides, A., & Copestake, A. A. (1999). Default representation in constraint-based frameworks. *Computational Linguistics*, *25*(1), 55-105.

Michie, D. (1968). Memo functions and machine learning. *In Nature*, *218*, 19–22.

Moore, R. C. (1995). *Logic and representation* (No. 39). Stanford, CA: CSLI Publications.

Moshier, M. (1988). *Extensions to unification grammar for the description of programming languages.* Unpublished doctoral dissertation, Ann Arbor, MI, USA.

Ninomiya, T., Miyao, Y., & Tsujii, J. (2002). Lenient default unification for robust processing within unification based grammar formalisms. In *Proceedings of the 19th international conference on computational linguistics, coling 2002* (pp. 744–750). Taipei, Taiwan.

Plotkin, G. D. (1976). A powerdomain construction. *SIAM J. Comput.*, *5*(3), 452–487.

Pollard, C., & Sag, I. A. (1994). *Head-driven phrase structure grammar.* Chicago, Illinois: University of Chicago Press and CSLI Publications.

Pollard, C. J., & Moshier, M. D. (1990). Unifying partial descriptions of sets. In P. P. Hanson (Ed.), *Logic programming and nonmonotonic reasoning: 4th international conference.* University of British Columbia Press.

Prüst, H., Scha, R., & Berg, M. van den. (1994). Linguistics and philosophy 17. Springer.

Romanelli, M. (2005). *Ontology-based representation and processing of plurals for human-machine dialogue systems with unification-based operations.* Unpublished master's thesis, Universität des Saarlandes.

Russell, G., Ballim, A., Carroll, J., & Warwick-Armstrong, S. (1992). A practical approach to multiple default inheritance for unification-based lexicons. *Comput. Linguist.*, *18*(3), 311–337.

Russell, G., Carroll, J., & Warwick, S. (1991). Multiple default inheritance in a unification-based lexicon. In D. Appelt (Ed.), *Proceedings of the 29th meeting of the association for computational linguistics* (pp. 215–211). Morristown, New Jersey: Association for Computational Linguistics.

Shieber, S. M. (1986). A simple reconstruction of gpsg. In *Proc. of the 11th coling* (pp. 211–215). Bonn, Germany.

Siekmann, J. H. (1984). Universal unification. In *Proceedings of the 7th international conference on automated deduction* (pp. 1–42). London, UK: Springer-Verlag.

Smolka, G. (1988). *A feature logic with subsorts* (Tech. Rep.). Stuttgart: IBM Germany. (LILOG report)

Smolka, G. (1992). Feature-constraint logics for unification grammars. *Journal of Logic Programming*, *12*(1&2), 51–87.

Vickers, S. (1989). *Topology via logic.* New York, NY, USA: Cambridge University Press.

Wahllöf, N. (1996). *A Default Extension to Description Logics and its Applications.* (Linköping University, Licentiate Thesis)

Wahlster, W. (2003, September). Towards symmetric multimodality: Fusion and fission of speech, gesture, and facial expression. In B. N. A. Günther R. Kruse (Ed.), *Ki 2003: Advances in artificial intelligence. proceedings of the 26th german conference on artificial intelligence* (pp. 1–18). Berlin, Heidelberg: Springer.

Wahlster, W., & Wahlster, H. W. (2006, 0). Dialogue systems go multimodal: The smartkom experience. In *Smartkom - foundations of multimodal dialogue systems* (p. 3-27). Springer.

Young, M. A., & Rounds, W. C. (1993). A logical semantics for nonmonotonic

sorts. In *Meeting of the association for computational linguistics* (pp. 209–215).