Classification and Representation of Types in $\mathcal{TDL}$

Hans-Ulrich Krieger
krieger@dfki.uni-sb.de

German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany
phone: (+49 681) 302-5299 fax: (+49 681) 302-5341

Abstract. This paper presents $\mathcal{TDL}$, a typed feature-based representation language and inference system, specifically designed to support highly lexicalized constraint-based grammar theories. Type definitions in $\mathcal{TDL}$ consist of type and feature constraints over the full Boolean connectives together with coreferences, thus making $\mathcal{TDL}$ Turing-complete. $\mathcal{TDL}$ provides open- and closed-world reasoning over types. Working with partially as well as with fully expanded types is possible. Efficient reasoning in $\mathcal{TDL}$ is accomplished through specialized modules.
In this paper, we will highlight the type/inheritance hierarchy module of $\mathcal{TDL}$ and show how we represent conjunctively and disjunctively defined types. Negated types and incompatible types are handled by specialized bottom symbols. Redefining a type only leads to the redefinition of the dependent types, and not to the redefinition of the whole grammar/lexicon. Undefined types are nothing special.
Reasoning over the type hierarchy is partially realized by a bit vector encoding of types, similar to the one used in Aït-Kaci's LOGIN. However, the underlying semantics does not harmonize with the open-world assumption of $\mathcal{TDL}$. Thus, we have to generalize the GLB/LUB operation to account for this fact.
The system, as presented in the paper, has been fully implemented in Common Lisp and is an integrated part of a large NL system. It has been installed and successfully employed at other sites and runs on various platforms.

1 Introduction

Over the last few years, constraint-based grammar formalisms have become the predominant paradigm in natural language processing and computational linguistics. Their success stems from the fact that they can be seen as a monotonic, high-level representation language for linguistic knowledge which can be given a precise mathematical semantics. The main idea of representing as much linguistic knowledge as possible through a unique data type called *feature structure*, allows the integration of different description levels, spanning phonology, syntax, and semantics. Here, the feature structure itself serves as the interface between the different linguistic strata (actually, coreferences are a means to achieve this). While the first approaches relied on annotated phrase structure rules (e.g., PATR-II), recent formalisms try to specify grammatical knowledge as well as lexicon entries entirely through feature structures. In order to achieve this goal, one must enrich the expressive power of the first unification-based formalisms with different forms of disjunctive descriptions. Later, additional operations came into play, e.g., negation. Other proposals consider the integration of functional/relational dependencies into the formalism which make them, in general, Turing-complete (e.g., ALE; cf. [6]). However the most important extension to formalisms consists of the incorporation of *types*, for instance in contemporary systems like TFS [20], CUF [8], ALE, or $\mathcal{TDL}$ [11]. Types are ordered hierarchically as it is known from object-oriented programming languages. This often leads to multiple inheritance in the description of linguistic entities. If a formalism is intended to be used as a stand-alone system, it must implement *recursive types* if it does not provide phrase-structure recursion.[1] In addition, certain relations (like *append*) or additional extensions of the formalism (like functional uncertainty) can be nicely modelled through recursive types.

2 Motivation

Modern typed unification-based grammar formalisms differ from early untyped systems, like PATR-II, in that they emphasize the notion of a *feature type*. Types can be arranged hierarchically, whereby a subtype monotonically *inherits* all the information from its supertypes and unification plays the role of the primary information-combining operation. In $\mathcal{TDL}$, an abstract *type definition*

$$s := \langle t, \phi \rangle$$

can be seen as an abbreviation for a complex expression, consisting of a complex type constraint t (concerning the sub-/supertype relationship) and a complex feature constraint ϕ (stating the necessary features and their values).

It is worth noting that $\mathcal{TDL}$ does not enforce a grammar writer to specify the type subsumption relation a priori through a set of $\{s_1, \ldots, s_n\} \preceq s$ statements, as is the case for LOGIN [4], ALE, or LIFE [1]. Instead, $\mathcal{TDL}$ automatically derives and extends a type hierarchy from the complex type expression t by means of normalized definitions (see below). In general, the type hierarchy only forms a partial order, i.e., we do not require additional ordering properties, e.g., BCPO/lower semilattice.

Types are a necessary requirement for a grammar development environment because they serve as abbreviations for lexicon entries, immediate dominance rule schemata, and universal as well as language-specific principles, as is familiar from Head-Driven Phrase Structure Grammar (HPSG) [16]. Types in $\mathcal{TDL}$ not

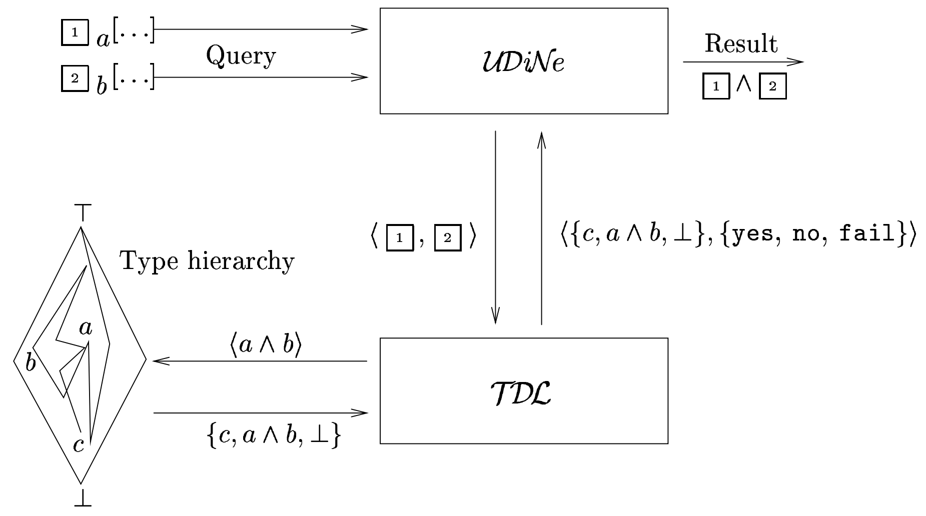
[1] For instance, ALE employs a bottom-up chart parser, whereas TFS relies entirely on type deduction. Note that recursive types can be substituted by definite clauses, as is the case for CUF, such that parsing/generation roughly corresponds to SLD resolution.

only serve as a shorthand, like templates, but also have other advantages over templates:

- STRUCTURING LINGUISTIC KNOWLEDGE
  Hierarchically-ordered types allow for a modular way to adequately represent linguistic knowledge. Moreover, generalizations can be made at the appropriate levels of representation.
- EFFICIENT PROCESSING
  Certain type constraints can be compiled into more efficient representations; for instance, [3, 2] reduce GLB (greatest lower bound), LUB (least upper bound), and $\preceq$ (type subsumption) computation to low-level bit manipulations. Moreover, types can be used to eliminate expensive unification operations, for example, by explicit declaration of type incompatibility. In addition, working with type names only or with partially expanded types minimizes the costs of copying structures during processing.
- REDUNDANCY
  In practice, it is often not possible to hold large lexicons completely in memory. However, only the idiosyncratic information of a lexicon entry needs to be represented in RAM, due to the fact that the lexical types of a lexicon entry contain most of the information, and only these types must be fully expanded.
- TYPE DISCIPLINE
  Type definitions allow a grammarian to declare which attributes are appropriate for a given type and which types are appropriate for a given attribute, therefore disallowing inconsistent feature structures.
- RECURSIVE TYPES
  Recursive types give a grammar writer the opportunity to formulate certain functions/relations or extensions to the formalism (e.g., functional uncertainty) as recursive type specifications. Parsing as Deduction [15] is often achieved by replacing the context-free backbone through recursive types.
- COMPILING TYPES
  Types are a good starting point for further methods of compilation. We have already mentioned bit vector encoding. Types can also serve as the basis for separating "true" and "spurious" constraints [7], for partial evaluation, or for compiling an HPSG grammar into a weaker formalism (e.g., [9]).

3 $\mathcal{TDL}$—An Overview

$\mathcal{TDL}$ is a unification-based grammar development environment and run time system supporting in particular HPSG-style grammars. Work on $\mathcal{TDL}$ has started within the DISCO project of the DFKI [19]. The DISCO grammar currently consists of more than 1500 type specifications written in $\mathcal{TDL}$ and is the largest HPSG grammar for German [13]. The core machine of DISCO consists of $\mathcal{TDL}$ and the feature constraint solver *UDiNe* [5]. *UDiNe* itself is a powerful untyped unification engine which allows the use of distributed disjunctions, general negation, and functional dependencies.



$\boxed{1}\ a[\ldots]$
$\boxed{2}\ b[\ldots]$
Query
$\mathcal{UDiNe}$
Result
$\boxed{1} \wedge \boxed{2}$
$\langle \boxed{1}, \boxed{2} \rangle$
$\langle \{c, a \wedge b, \bot\}, \{$yes, no, fail$\} \rangle$
Type hierarchy
$\top$
a
b
c
$\bot$
$\langle a \wedge b \rangle$
$\{c, a \wedge b, \bot\}$
$\mathcal{TDL}$

Fig. 1. *Interface between $\mathcal{TDL}$ and $\mathcal{UDiNe}$. Depending on the type hierarchy and the type of $\boxed{1}$ and $\boxed{2}$, $\mathcal{TDL}$ either returns c (c is definitely the GLB of a and b) or $a \wedge b$ (open-world reasoning for GLB) or $\bot$ (closed-world reasoning for GLB) if a single type which is equal to the GLB of a and b does not exist. In addition, $\mathcal{TDL}$ determines whether $\mathcal{UDiNe}$ must carry out feature term unification (*`yes`*) or not (*`no`*), i.e., the return type contains all the information one needs to work on properly (*`fail` *signals a global unification failure).*

The modules communicate through an interface, and this communication mirrors exactly the way an abstract typed unification algorithm works: two typed feature structures can only be unified if the attached types are known to be compatible. This is accomplished by the unifier by handing over two typed feature structures to $\mathcal{TDL}$ which gives back a simplified form (plus additional information; see Fig. 1).
The motivation for separating types and features and processing them in specialized modules (which again might consist of specialized components as is the case in $\mathcal{TDL}$) is twofold: (i) this strategy reduces the complexity of the whole system, thus making the architecture clear, and (ii) leads to faster processing because every module is designed to handle only a specialized reasoning task. Furthermore, extensions to $\mathcal{TDL}$ can be integrated easily.

$\mathcal{TDL}$ supports type definitions consisting of type constraints and feature constraints over the standard operators $\wedge$, $\vee$, and $\neg$. The operators are generalized to connect feature descriptions, coreference tags (logical variables) and types. $\mathcal{TDL}$ distinguishes between

- AVM types (open-world semantics; see below)

– SORT types (closed-world semantics; see below)
– BUILT-IN types (through COMMON LISP)
– ATOMS (symbols, strings, numbers, etc.)

When asked for the greatest lower bound of two AVM types a and b which share no common subtype, $\mathcal{TDL}$ returns $a \wedge b$ (open-world reasoning), and not $\perp$, as is the case, e.g., in ALE or LOGIN/LIFE.[2] The reasons for assuming this are manifold:

1. partiality of our linguistic knowledge about a specific domain (in case, we know nothing about the GLB, we simply return the conjunction of the types which is definitely correct—this strategy obviously preserves the denotation)
2. the approach is in harmony with terminological (KL-ONE-like) languages which share a similar semantics
3. this view makes the stepwise refinement of grammars during the development process easier (which has been shown useful in several projects)
4. we must not write superfluous type definitions to guarantee successful type unifications during processing

The opposite case holds for the GLB of SORT types. SORT types differ from AVM types in that they are not further structured (they are featureless), as is the case for ATOMS (which cannot be arranged hierarchically and consume less space).

$\mathcal{TDL}$ allows for the declaration of *partitions*, a feature heavily used in HPSG. One can even declare sets of AVM types as *incompatible*, meaning that their conjunction yields $\perp$, so that specific types can be "closed", if desired.

The kernel of $\mathcal{TDL}$ (and of most other systems) can be given a set-theoretical semantics, e.g., along the lines of [17]. It is easy to translate $\mathcal{TDL}$ statements into denotation-preserving expressions of Smolka's feature logic or even into a set of definite equivalences [18], i.e., a definite program, thus viewing $\mathcal{TDL}$ as just syntactic sugar for first-order predicate logic.[3] The latter point is of special importance, since by viewing the type hierarchy as a pure "transport medium" for constraints, we can transform nonmonotonically defined types into a perfectly definite program.

[2] Thus, typed feature structures in $\mathcal{TDL}$ might be typed with complex types like $a \wedge b$ or $a \wedge (b \vee \neg c)$, and not with type symbols only.

[3] Cf. [10] for a precise description of the semantics of $\mathcal{TDL}$, including a fixpoint characterization of recursive types. In contrast to most settings, we propose the *greatest* fixpoint (of a certain downward continuous function) as the solution of a grammar because it makes the least restrictions on admissible interpretations—note that we are interested in the *satisfiability* (and not in the validity) of a set of grammatical descriptions (i.e., a type system). Perhaps more important, the greatest fixpoint will not rule out cyclic feature structures and certain coreference constraints, as might be the case for the least fixpoint interpretation of a type system.

4 Type Hierarchy

The type hierarchy is either called directly by the control machine of $\mathcal{TDL}$ during the definition of a type (type classification) or indirectly via the symbolic simplifier, both at definition and at run time (typed unification and type expansion).[4]

4.1 Basic Encoding Method

The hierarchy itself is represented as a double linked graph, such that types are associated with forward and backward pointers to their immediate subtypes and supertypes. Because we allow for conjunctively as well as disjunctively defined types, types possess pointers to their disjunction alternatives but also to disjunctive types in which they are involved. Types are equipped with further information, e.g., slots containing the dependent types (important for redefinition) or the specialized bottom symbols in case of incompatible AVM types.

Since we are interested to perform GLB, LUB, and $\preceq$ computations efficiently (important during typed unification and type expansion), not only is the type hierarchy explicitly represented, but also compiled into a special format (actually, every type is associated with a specific code). The compilation is based on Hassan Aït-Kaci's bit vector encoding technique for partial orders [3, 2]) and has been further extended to serve our special requirements.

Here, every type t is assigned a code $\gamma(t)$ (represented through a bit vector) such that $\gamma(t)$ encodes the reflexive and transitive closure of the immediate type subsumption relation with respect to t. Decoding a code c is either realized by a (hash) table look-up (iff $\exists t\,.\; \gamma^{-1}(c) = t$) or by computing the "maximal restriction" of the set of types whose codes are less than c.[5]
Depending on the encoding method, the hierarchy occupies between $O(n \log n)$ (compact encoding) and $O(n^2)$ (transitive closure encoding) bits. Here, a GLB (LUB) operation directly corresponds to bitwise And (Or) instruction. In this framework, GLB, LUB and $\preceq$ computations have the pleasant property that they can be carried out in $O(n)$, where n is the number of types (actually $O(1)$, since n does not change at run time).[6]

Aït-Kaci's method has been extended to account for the open-world nature of AVM types, in that potential GLB/LUB candidates (calculated from their codes)

[4] Type expansion [12] or type unfolding means to make the inherent constraints of a type explicit and to (partially) check for its consistency.

[5] Aït-Kaci has argued that decoding, i.e., calling γ^{-1} is not necessary at run time. However, this is not true for our setting (partially expanded structures, complex type expressions, different semantics; see below). Decoding in our system is similar to encoding, in that we employ a hash table of the inverse images of γ. $\gamma^{-1}(b)$ thus means to access the type symbol that is associated with code b.

[6] One can choose in $\mathcal{TDL}$ between the two encoding techniques and between bit vectors and bignums (arbitrary long integers) for the representation of the codes. In general, operations on bignums in most COMMON LISP implementations are at least an order of magnitude faster than the corresponding operations on bit vectors.

are verified by partially "inspecting" the type hierarchy. Why so? Consider, for example, the following type hierarchy which has been obtained via the definitions $s := \langle \top, \phi \rangle$ and $t := \langle \top, \psi \rangle$:

$$\begin{tikzcd}
 & \top \arrow[r, "\gamma"] \arrow[dl, dash] \arrow[dr, dash] & 11 \\
01 & s \arrow[l, "\gamma"'] & t \arrow[r, "\gamma"] & 10
\end{tikzcd}$$

Simplifying $s \wedge t$ on the basis of the codes would lead us to $\bot$: $\gamma(s) \otimes \gamma(t) = 00 = \gamma(\bot)$, where $\otimes$ means bitwise And. Dual to this, we obtain $\gamma(s \vee t) = \gamma(\top)$ which is often too crude. In general, these results would only hold in $\mathcal{TDL}$ if s and t are declared as incompatible or exhaustively partition $\top$, resp. Rather for this hierarchy, we argue that the GLB of s and t is $s \wedge t$, whereas the LUB should be $s \vee t$ (if s and t are AVM types).

Take another example to see why Aït-Kaci's original treatment is not the right choice for our setting. Consider the following two type definitions:

$$x := \langle y \wedge z, \top \rangle$$
$$x' := \langle y' \wedge z', p \doteq 1 \rangle$$

During processing, we can definitely substitute $y \wedge z$ by x, but rewriting $y' \wedge z'$ to x' is not correct, because x' differs from $y' \wedge z'$—x' is more specific as a consequence of the feature constraint ($p \doteq 1$). If we would rewrite $y' \wedge z'$ to x', type expansion would yield a more specific structure than necessary. Recall that types abbreviate the constraints defined on them.
In order to obtain the "intended" result, we mark types during the definition phase whether their denotation is equivalent to the intersection of the denotation of their direct supertypes or not (same for disjunctively defined types). In our example above, we say that x is the GLB of y and z ($\textit{is-glb}(x) \leftarrow$ TRUE), whereas x' is not considered to be the GLB of y' and z' ($\textit{is-glb}(x') \leftarrow$ FALSE). Note that such information represents only local knowledge about the direct subtypes/supertypes.

Another point in Aït-Kaci's treatment does not harmonize with our setting. It could be the case that the computation of the GLB of $S = \{s_1, \ldots, s_n\}$ leads to a code b that does not have a correspondence in the type hierarchy (same for LUB). In this case, the set of all maximal elements $\{t_1, \ldots, t_m\}$ whose codes are less than b is returned:

$$\begin{tikzcd}
\mathrm{GLB}(S) \arrow[r, "\gamma"] & \bigotimes_{i=1}^{n} \gamma(s_i) \arrow[d, equal] \\
\{t_1, \ldots, t_m\} & b \arrow[l, "\gamma^{-1}"']
\end{tikzcd}$$

But obviously

$$\bigcup_{j=1}^{m} [\![t_j]\!]^{\mathcal{I}} = \bigcap_{i=1}^{n} [\![s_i]\!]^{\mathcal{I}}$$

is **not** the case for every interpretation $\mathcal{I}$ of our type system. Because $\{t_1, \ldots, t_m\}$ is interpreted disjunctively and the GLB is interpreted as logical conjunction in the presence of a type hierarchy, we rather have that $\{t_1, \ldots, t_m\}$ only approximates GLB(S), since for every $j \in \{1, \ldots, m\}$:

$$[\![t_j]\!]^{\mathcal{I}} \subseteq \bigcap_{i=1}^{n} [\![s_i]\!]^{\mathcal{I}}$$

Hence, the implementation of the greatest lower bound distinguishes between the

– INTERNAL GREATEST LOWER BOUND $\mathrm{GLB}_{\preceq}$
  only the type hierarchy, i.e., the type subsumption relation $\preceq$ is taken into account by employing the codes (used in case of SORT types)

$$\mathrm{GLB}_{\preceq}(s,t) := \gamma^{-1}(\gamma(s) \otimes \gamma(t))$$

– EXTERNAL GREATEST LOWER BOUND $\mathrm{GLB}_{\sqsubseteq}$
  take feature constraints into account via the *is–glb* slot (see example above)

```
$\mathrm{GLB}_{\sqsubseteq}(s,t) :=$
  **local** $b$;
  $b \leftarrow \gamma(s) \otimes \gamma(t)$;
  **if** $\gamma^{-1}(b) \uparrow$
    /* if $b$ doesn't have a corresponding type */
    **then return** $s \wedge t$
    **else if** *verify–glb–p*$(\{\gamma^{-1}(b)\}, \{s,t\})$
           **then return** $\gamma^{-1}(b)$
           **else return** $s \wedge t$.

*verify–glb–p*(*supers*, *query*) :=
  **local** $S$;
  $S \leftarrow \bigcup_{s \in supers}$ *direct–supertypes*$(s) \setminus$ *query*;
  **if** $S = \emptyset$
    **then return** TRUE
    **else if** $\forall s \in S$ . *is–glb*$(s)$
           **then** *verify–glb–p*($S$, *query*)
           **else return** FALSE.
```

A similar distinction is made for the LUB.

With $\mathrm{GLB}_{\preceq}$ and $\mathrm{GLB}_{\sqsubseteq}$ in mind, we can define a generalized GLB operation informally by the following table. This GLB operation is actually used during typed unification (*fc*: feature constraint):

GLB	avm_1	$sort_1$	$atom_1$	fc_1
avm_2	1.	$\perp$	$\perp$	2.
$sort_2$	$\perp$	3.	4.	$\perp$
$atom_2$	$\perp$	4.	5.	$\perp$
fc_2	2.	$\perp$	$\perp$	6.

where

$$1.\ \begin{cases} avm_3 \iff \text{GLB}_{\sqsubseteq}(avm_1, avm_2) = avm_3 \\ avm_1 \iff avm_1 = avm_2 \\ \perp \iff \text{GLB}_{\preceq}(avm_1, avm_2) = \perp \text{ (via explicit incompatibility declaration)} \\ avm_1 \wedge avm_2, \text{ otherwise (open-world reasoning for GLB)} \end{cases}$$

$$2.\ \begin{cases} avm_{1,2} \iff \text{type expansion is switched off} \\ avm_{1,2} \iff \textit{expand-tfs}(\langle avm_{1,2}, fc_{2,1} \rangle) \neq \perp \text{ (type exp. switched on)} \\ \perp, \text{ otherwise} \end{cases}$$

$$3.\ \begin{cases} sort_3 \iff \text{GLB}_{\preceq}(sort_1, sort_2) = sort_3 \\ sort_1 \iff sort_1 = sort_2 \\ \perp, \text{ otherwise (closed world reasoning for GLB)} \end{cases}$$

$$4.\ \begin{cases} atom_{1,2} \iff \textit{type-of}(atom_{1,2}) \preceq sort_{2,1}, \text{ where } sort_{2,1} \text{ is a built-in type} \\ \perp, \text{ otherwise} \end{cases}$$

$$5.\ \begin{cases} atom_1 \iff atom_1 = atom_2 \\ \perp, \text{ otherwise} \end{cases}$$

$$6.\ \begin{cases} \top \iff fc_1 \wedge fc_2 \neq \perp \\ \perp, \text{ otherwise} \end{cases}$$

Actually, the GLB definition is a little bit more complicated in that we allow for arbitrary many arguments.

The encoding algorithm has also been extended to proper handle the redefinition of types and the use of undefined types, an essential part of an incremental grammar/lexicon development system. Redefining a type not only means to make changes local to this type. Rather, one has to redefine all *dependents* of this type—all subtypes, in case of a conjunctively defined type and all disjunction elements for a disjunctive type specification, plus, in both cases, all types which mention these types in their definition.
The dependent types of a type t can be characterized graph-theoretically via the *connected component* (CC) of t with respect to the "dependency" relation, informally defined above. This relation is updated every time a new type definition is given to $\mathcal{TDL}$. It is important to redefine the dependents in the "right" order to obtain a new consistent type hierarchy. In general, enriching the type hierarchy with dependency links no longer leads to a cycle-free graph. So it is not obvious how to establish a topological order on the set of types. However, one can topologically sort the CCs of the hierarchy without dependency links (which leads to a total order with respect to a certain CC) and then implode the CCs of the hierarchy into nodes (which ultimately leads to a DAG which itself can be totally ordered too).

4.2 Decomposing Type Definitions

Conjunctively defined types (e.g., $x := \langle y \wedge z, \phi \rangle$) and disjunctively defined ones (e.g., $x' := \langle y' \vee z', \phi' \rangle$) are entered differently into the type hierarchy: x inherits feature constraints from its supertypes y and z, whereas x' defines itself through its disjunction alternatives y' and z'.[7] This distinction is represented through the use of different kinds of edges in the type graph (bold edges denote disjunction elements, see Fig. 3 and 4).
One might ask how conjunctively and disjunctively defined types affect the bit vector encoding method. The answer is simply that this distinction does not have any effects on the encoding algorithms—recall that disjunctive and conjunctive inheritance links both denote the immediate subsumption relation ($x \preceq y$ and $x' \succeq y'$ in the above example), and exactly the transitive closure of $\preceq$ is encoded in the bit vectors.

$\mathcal{TDL}$ decomposes complex type definitions consisting of $\wedge$, $\vee$, and $\neg$ by introducing *intermediate types*, so that the resulting expression is either a pure conjunction or a disjunction of type **symbols** (plus type definitions of the form $s := \langle \neg t, \top \rangle$).
It is not hard to realize that arbitrary type systems can be "normalized" in such a way, simplifying the integration of a new type w.r.t. an existing type hierarchy. Now let $s := \langle t, \phi \rangle$ be a normalized type definition. Thus

- **if** $t = t_1 \wedge \cdots \wedge t_n$ **then** let s inherit from $t'_1, \ldots, t'_m$,
  where $\mathrm{GLB}(\{t_1, \ldots, t_n\}) = t'_1 \wedge \cdots \wedge t'_m$, thus $s \preceq t'_1, \ldots, s \preceq t'_m$
- **if** $t = t_1 \vee \cdots \vee t_n$ **then** let $t'_1, \ldots, t'_m$ be the disjunction alternatives of s,
  where $\mathrm{LUB}(\{t_1, \ldots, t_n\}) = t'_1 \vee \cdots \vee t'_m$, thus $t'_1 \preceq s, \ldots, t'_m \preceq s$
- **if** $t = \neg t_1$ **then** make s incompatible with t_1
  and so we have $\bot \doteq s \wedge t_1$ (incompatibility declaration)

Fig. 2 gives an example of such a normalized type hierarchy. Notice here that the previously introduced intermediate type $|u \wedge v|$ is involved in the definition of the new intermediate $|u \wedge v \wedge w|$ (we enclose intermediate type names in vertical bars).

The same technique is applied when using the xor macro $\veebar$ (see Fig. 3 and 4). $\veebar$ will be decomposed into $\wedge$, $\vee$ and $\neg$, plus additional intermediates. For each negated type $\neg t$, $\mathcal{TDL}$ introduces a new intermediate type symbol $|\neg t|$, having the definition $\neg t$ and declares it incompatible with t (see Section 4.3). In addition, if t is not already present, $\mathcal{TDL}$ will add t as a new type to the hierarchy, directly under $\top$ (see types $|\neg b|$ and $|\neg c|$ in Fig. 3 and 4).

Let us consider the example $a := \langle b \veebar c, \top \rangle$. The decomposition performed by $\mathcal{TDL}$ can be stated informally by the following rewrite steps (assuming that CNF mode is switched on; see Fig. 3):

[7] So one can see conjunctive types as *top-down specializations* of supertypes and disjunctive ones as *bottom-up generalizations* of disjunction elements.

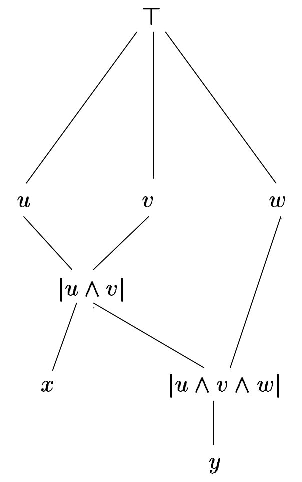


Fig. 2. *The intermediate types* $|u{\wedge}v|$ *and* $|u{\wedge}v{\wedge}w|$ *are introduced during the definition the type* $x := \langle u \wedge v, a \doteq 0\rangle$, *followed by* $y := \langle w \wedge v \wedge u, a \doteq 1\rangle$.

$$\cfrac{a := \langle b \forall\!\!\!\!\vee c, \top\rangle}{\cfrac{a := \langle (b \wedge \neg c) \vee (\neg b \wedge c), \top\rangle}{\cfrac{a := \langle (b \vee \neg b) \wedge (b \vee c) \wedge (\neg b \vee \neg c) \wedge (\neg c \vee c), \top\rangle}{\cfrac{a := \langle (b \vee c) \wedge (\neg b \vee \neg c), \top\rangle}{a := \langle |b \vee c| \wedge |\neg b \vee \neg c|, \top\rangle}}}}$$

where $|b \vee c| := \langle b \vee c, \top\rangle$, $|\neg b \vee \neg c| := \langle |\neg b| \vee |\neg c|, \top\rangle$, $|\neg b| := \langle \neg b, \top\rangle$, $|\neg c| := \langle \neg c, \top\rangle$, $\bot_{\{b,\neg b\}} := \langle b \wedge |\neg b|, \top\rangle$, and $\bot_{\{c,\neg c\}} := \langle c \wedge |\neg c|, \top\rangle$.

Instead, if disjunctive normal form is enforced by the user, the decomposition of $a := \langle b \forall\!\!\!\!\vee c, \top\rangle$ leads of course to a different type hierarchy (Fig. 4):

$$\cfrac{a := \langle b \forall\!\!\!\!\vee c, \top\rangle}{\cfrac{a := \langle (b \wedge \neg c) \vee (\neg b \wedge c), \top\rangle}{a := \langle |b \wedge \neg c| \vee |\neg b \wedge c|, \top\rangle}}$$

4.3 Incompatible Types and Bottom Propagation

Incompatible types lead to the introduction of specialized bottom symbols (see Fig. 3, 4 and 5) which are, however, identified in the underlying logic (this identification is somewhat related to the notion of a *coalesced sum*, known from domain theory). I.e., these symbols are **always** interpreted as representing inconsistent information, thus they denote the empty set.

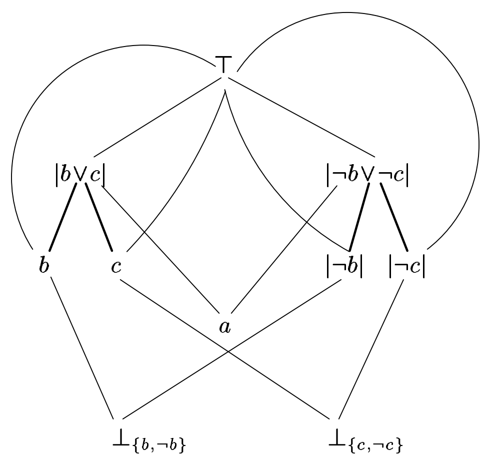


Fig. 3. *Decomposing $a := \langle b \forall c, \top \rangle$ into conjunctive normal form, such that a inherits from the intermediates $|b \vee c|$ and $|\neg b \vee \neg c|$.*

Bottom symbols are propagated "downwards" by a mechanism called *bottom propagation* which takes place at definition time (see Fig. 5). This is important, since we want to apply the GLB operation to incompatibly declared types in order to take advantage of the bit vector encoding. Detecting a bottom symbol with the help of the codes of the types is enough here, thus we only need to employ GLB$_{\preceq}$.[8]

Note that it is important to take not only conjunctively defined subtypes during bottom propagation into account but also disjunction elements, as the following example shows. Assume that the user declares the AVM types a and b as incompatible (via $\perp \doteq a \wedge b$). Thus we have

$$\left\{ \begin{array}{c} \perp \doteq a \wedge b \\ b := \langle b_1 \vee b_2, \top \rangle \end{array} \right\} \xrightarrow{\textit{bottom propagation}} a \wedge b_1 = \perp \text{ and } a \wedge b_2 = \perp$$

It is worth noting that because we employ an explicitly represented type hierarchy during GLB, LUB and $\preceq$ computations, a single bottom symbol that is a subtype of every other minimal type, would lead to false inferences.

[8] In case that the GLB operation allows arbitrary many arguments, this strategy must be modified. Assume that we declare a set of types T as incompatible. The specialized bottom symbol $\perp_T$ then encode that $\bigwedge_{t \in T} = \perp$. Obviously, if $T' \subset T$ then $\bigwedge_{t \in T'} \neq \perp$, in general. Now, given a set of types S and assuming that GLB$_{\preceq}(S) = \perp_T$, we must guarantee that $\forall t \in T, \exists s \in S \,.\, s \preceq t$. This test can be carried out very quickly, since $\preceq$ is implemented through bit vectors.

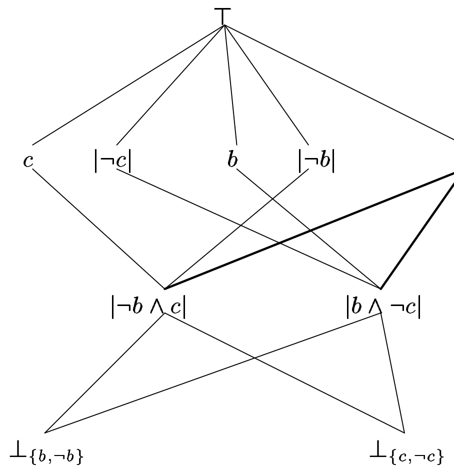


Fig. 4. *Decomposing $a := \langle b \forall c, \top \rangle$ into disjunctive normal form, such that a is defined through its disjunction alternatives $|b \wedge \neg c|$ and $|\neg b \wedge c|$.*

Consider the following example. Assume that we declare a and b, as well as c and d as incompatible. If only a single bottom symbol $\perp$ is used, we would deduce that $a \wedge c$ is $\perp$ which is not necessarily the case. However, introducing two bottom symbols $\perp_{\{a,b\}}$ and $\perp_{\{c,d\}}$ is the right way to guarantee proper results.

One might expect that incompatibility statements together with feature term unification no longer lead to a monotonic, set-theoretical semantics. But this is not the case. To preserve monotonicity, one must assume a *2-level interpretation* of typed feature structures, where feature constraints and type constraints can denote different sets of objects and the global interpretation is determined by the intersection of the two sets.
Take for instance the type definitions $A := \langle \top, a \doteq 1 \rangle$ and $B := \langle \top, b \doteq 1 \rangle$, plus the user declaration $\perp \doteq A \wedge B$, meaning that A and B are incompatible. Then $A \wedge B$ will simplify to $\perp$ although the corresponding feature structures of A and B successfully unify to $(a \doteq 1) \wedge (b \doteq 1)$.

5 Additional Modules

In this section, we hasten to present additional reasoning engines of $\mathcal{TDL}$ that are related to the type hierarchy module.

First of all, the type hierarchy reasoner is part of a larger symbolic simplifier that further implements the standard "syntactic" simplification schemata, e.g., De Morgan's laws, idempotence, double negation, etc.

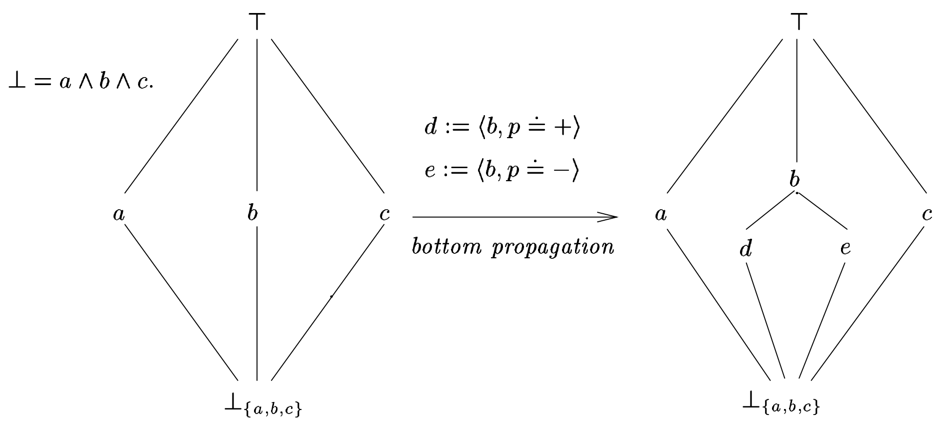

$\bot = a \wedge b \wedge c.$
$\top$
a
b
c
$\bot_{\{a,b,c\}}$
$d := \langle b, p \doteq + \rangle$
$e := \langle b, p \doteq - \rangle$
bottom propagation
$\top$
b
a
d
e
c
$\bot_{\{a,b,c\}}$

Fig. 5. *Bottom propagation triggered through the subtypes d and e of b, so that $a \wedge d \wedge c$ as well as $a \wedge e \wedge c$ will simplify to $\bot$ during processing.*

Second, this simplifier is extensively used during type expansion to reduce the costs of typed unification and copying.

Third, simplified expressions are memoized [14] in order to reuse them later. Here the unsimplified expression serves as the key in a hash table, so that the corresponding value is exactly the simplified formula. To reduce the number of keys, we impose a generalized total order on type expressions, such that there is exactly one representative for a whole class of equivalent formulae.
The time for accessing such a formula is extremely short, e.g., 0.05 ms for an arbitrary access over a hash table of about 4000 entries (Sun SPARC SS10, Allegro CL). This is much faster than the corresponding operations on bit vectors.

References

1. Hassan Aït-Kaci. An introduction to LIFE—programming with logic, inheritance, functions, and equations. In *Proceedings of the International Symposium on Logic Programming*, pages 52–68, 1993.
2. Hassan Aït-Kaci, Robert Boyer, Patrick Lincoln, and Roger Nasr. Efficient implementation of lattice operations. *ACM Transactions on Programming Languages and Systems*, 11(1):115–146, January 1989.
3. Hassan Aït-Kaci, Robert Boyer, and Roger Nasr. An encoding technique for the efficient implementation of type inheritance. Technical Report AI-109-85, MCC, Austin, TX, 1985.
4. Hassan Aït-Kaci and Roger Nasr. LOGIN: A logic programming language with built-in inheritance. *Journal of Logic Programming*, 3:185–215, 1986.
5. Rolf Backofen and Christoph Weyers. *UDiNe*—a feature constraint solver with distributed disjunction and classical negation. Unpublished documentation note, 1994.

6. Bob Carpenter and Gerald Penn. ALE—the attribute logic engine user's guide. version 2.0. Technical report, Laboratory for Computational Linguistics. Philosophy Department, Carnegie Mellon University, Pittsburgh, PA, August 1994.
7. Abdel Kader Diagne, Walter Kasper, and Hans-Ulrich Krieger. Distributed parsing with HPSG grammars. Technical report, German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany, 1995. To appear.
8. Jochen Dörre and Michael Dorna. CUF—a formalism for linguistic knowledge representation. In Jochen Dörre, editor, *Computational Aspects of Constraint-Based Linguistic Description I*. DYANA, 1993.
9. Robert T. Kasper, Bernd Kiefer, Klaus Netter, and K. Vijay-Shanker. Compilation of HPSG yo TAG. In *Proceedings of the 33rd Annual Meeting of the Association for Computational Linguistics, ACL-95*, 1995.
10. Hans-Ulrich Krieger. *$\mathcal{TDL}$—A Type Description Language for Constraint-Based Grammars. Foundations, Implementation, and Applications.* PhD thesis, Universität des Saarlandes, Department of Computer Science, 1995. Forthcoming.
11. Hans-Ulrich Krieger and Ulrich Schäfer. $\mathcal{TDL}$—a type description language for constraint-based grammars. In *Proceedings of the 15th International Conference on Computational Linguistics, COLING-94*, pages 893–899, 1994.
12. Hans-Ulrich Krieger and Ulrich Schäfer. Efficient parameterizable type expansion for typed feature formalisms. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence, IJCAI-95, Montreal, Canada*, 1995. To appear.
13. Klaus Netter. Architecture and coverage of the DISCO grammar. In S. Busemann and Karin Harbusch, editors, *Proceedings of the DFKI Workshop on Natural Language Systems: Modularity and Re-Usability, DFKI, D-93-03*, 1993.
14. Peter Norvig. *Paradigms of Artificial Intelligence Programming.* Morgan Kaufmann, San Mateo, CA, 1991.
15. Fernando C.N. Pereira and David H.D. Warren. Parsing as deduction. In *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics, ACL-83*, pages 137–144, 1983.
16. Carl Pollard and Ivan A. Sag. *Information-Based Syntax and Semantics. Vol. I: Fundamentals.* CSLI Lecture Notes, Number 13. Center for the Study of Language and Information, Stanford, 1987.
17. Gert Smolka. A feature logic with subsorts. LILOG Report 33, WT LILOG–IBM Germany, Stuttgart, May 1988. Also in J. Wedekind and C. Rohrer (eds.), Unification in Grammar, MIT Press, 1991.
18. Gert Smolka. Residuation and guarded rules for constraint-logic programming. Research Report RR-91-13, German Research Center for Artificial Intelligence (DFKI), Saarbrücken, 1991.
19. Hans Uszkoreit, Rolf Backofen, Stephan Busemann, Abdel Kader Diagne, Elizabeth A. Hinkelman, Walter Kasper, Bernd Kiefer, Hans-Ulrich Krieger, Klaus Netter, Günter Neumann, Stephan Oepen, and Stephen P. Spackman. DISCO—an HPSG-based NLP system and its application for appointment scheduling. In *Proceedings of COLING-94*, pages 436–440, 1994. A version of this paper is available as DFKI Research Report RR-94-38.
20. Rémi Zajac. Inheritance and constraint-based grammar formalisms. *Computational Linguistics*, 18(2):159–182, 1992.

This article was processed using the LaTeX macro package with LLNCS style