Typed Feature Structures, Definite Equivalences, Greatest Model Semantics, and Nonmonotonicity

Hans-Ulrich Krieger
German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany
phone: (+49 681) 302-5299 fax: (+49 681) 302-5341
krieger@dfki.uni-sb.de

May 13, 1998

Abstract

Typed feature logics have been employed as description languages in modern type-oriented grammar theories like HPSG and have laid the theoretical foundations for many implemented systems. However, recursivity pose severe problems and have been addressed through specialized powerdomain constructions which depend on the particular view of the logician.

In this paper, we argue that definite equivalences as introduced by Smolka can serve as the formal basis for arbitrarily formalized typed feature structures and typed feature-based grammars/lexicons, as employed in, e.g., TFS or TDL. The idea here is that type definitions in such systems can be transformed into an equivalent definite program, whereas the meaning of the definite program then is identified with the denotation of the type system. Now, models of a definite program P can be characterized by the set of ground atoms which are logical consequences of the definite program. These models are ordered by subset inclusion and, for reasons that will become clear, we propose the greatest model as the intended interpretation of P, or equivalent, as the denotation of the associated type system.

Our transformational approach has also a great impact on nonmonotonically defined types, since under this interpretation, we can view the type hierarchy as a pure transport medium, allowing us to get rid of the transitivity of type information (inheritance), and yielding a perfectly monotonic definite program.

Acknowledgements. This paper has benefited from numerous people at various workshops where parts of it have been presented, in particular, the workshop on *Implementations of Attribute-Value Logics for Grammar Formalisms* at the European Summer School in Language, Logic, and Information (Lisbon); the workshop on *Issues of Computation and Formalization* (Tübingen); and the *Meeting on Mathematics of Language* (Philadelphia).

Contents

1 Introduction

Typed feature logics (e.g., Carpenter 1992) have been employed as description languages in modern type-oriented grammar theories like HPSG (Pollard and Sag 1987, Pollard and Sag 1994) and have laid the theoretical foundations for many implemented systems. However, recursivity poses severe problems and has been addressed through specialized powerdomain constructions which depend on the particular view of the logician (e.g., Pollard and Moshier 1990).

In this paper, we argue that *definite equivalences* as introduced by Smolka 1991 and formally defined in Smolka 1993 can serve as the formal basis for arbitrarily formalized *typed feature structures* and *typed feature-based grammars/lexicons* (for short *type systems*), as employed in, e.g., TFS (Zajac 1992) or $\mathcal{TDL}$ (Krieger and Schäfer 1994).

The idea here is that type definitions in such systems can be transformed into an equivalent definite program, whereas the meaning of the definite program then is identified with the denotation of the type system. Now, models of a definite program P can be characterized by the set of ground atoms which are logical consequences of the definite program (Lloyd 1987). These models are thus ordered by subset inclusion and, for reasons that will become clear, we propose the *greatest model* as the intended interpretation of P, or equivalent, as the denotation $\|\Omega\|$ of the associated type system Ω. This general idea is depicted in Fig. 1.

$$\begin{array}{ccc} \Omega & \xrightarrow{trans} & P \\ \downarrow & & \downarrow \\ \|\Omega\| & \equiv & \text{intended model of } P \\ & & = gfp(T_P) \end{array}$$

Figure 1: *General idea of the transformation schema. The meaning of the type system Ω is given by the greatest model of the corresponding definite program P which is characterized by the greatest fixpoint of the associated function T_P of P.*

Our "transformational" approach has a great impact on *nonmonotonically defined types*, since under this interpretation, we can view the type hierarchy as a pure *transport medium*, allowing us to get rid of the transitivity of type information (inheritance), and yielding a perfectly monotonic definite program.

2 Types, Typed Feature Structures, and Type Definitions

In order to describe our approach, we need only a small inventory to abstract from a concrete implementation. First of all, we assume pairwise disjoint sets of *features* (or *attributes*) $\mathcal{F}$ (f, g, h), *atoms* (or *constants*) $\mathcal{A}$ (a, b, c), (logical) *variables* $\mathcal{V}$ (x, y, z), and *types* (or *sorts*) $\mathcal{T}$ (s, t, u).[1] A sequence of features is called a *path* (e.g., fgh). Let ϵ denote the empty path. The set of paths is abbreviated by $\mathcal{F}^*$. $\mathcal{T}$ contains two special symbols, viz., $\top$, denoting the set of all objects (the universe) and $\bot$, denoting inconsistent information (the empty set). The set of *complex types* $\mathcal{T}^*$ then is defined by employing the Boolean connectives: $s \in \mathcal{T}^*$ iff

[1] We assume here that $\mathcal{T}$, $\mathcal{F}$, and $\mathcal{A}$ are finite for a given grammar/lexicon. Without loss of generality, however, $\mathcal{V}$ can be assumed to be infinite.

$s \in \mathcal{T} \cup \mathcal{A}$ or $s = \bigwedge_i t_i$ or $s = \bigvee_i t_i$ or $s = \neg s'$. In the following, we refer to a *type hierarchy* (or *inheritance hierarchy*) by the pair $\langle \mathcal{T}, \preceq \rangle$, such that $\preceq \subseteq \mathcal{T} \times \mathcal{T}$ is a decidable *partial order*, i.e., $\preceq$ is reflexive, antisymmetric, and transitive. We call the quintuple $\Sigma = \langle \mathcal{F}, \mathcal{V}, \mathcal{A}, \mathcal{T}, \preceq \rangle$ a *signature*. A *typed feature structure* θ is an extension of Aït-Kaci's ψ/ϵ-terms (Aït-Kaci 1986)[2]

$$\theta ::= \langle x, s, \phi \rangle \tag{1}$$

such that $x \in \mathcal{V}$, $s \in \mathcal{T}^*$, and $\phi \in \mathcal{C}^*$. $\mathcal{C}^*$ is inductively defined as follows: $\phi \in \mathcal{C}^*$ iff $\phi = (f \doteq \theta)$ or $\phi = \bigwedge_i \phi_i$ or $\phi = \bigvee_i \phi_i$ or $\phi = \neg\phi'$ or $\phi = \top$ (abbreviating the empty conjunction) or $\phi = \bot$ (empty disjunction). We call $f \doteq \theta$ a *feature constraint* (or *attribute-value pair*). We abbreviate the set of typed feature structures by Θ. Finally, we define a *type system* Ω as a finite set of type definitions: $\Omega = \{t := \theta \mid t \in \mathcal{T}, \theta \in \Theta\}$. I.e., a type definition $t := \theta$ *simultaneously* specifies (via θ) the constraints defined on t as well as the types of the inheritance hierarchy to which t refers to. Clearly, type systems encode grammars and lexicons.

Let us now focus on the intended denotation of feature constraints, types, typed feature structures, etc. to get an intuitive feeling for the meaning of such descriptions. Clearly, a clean definition of this will also be given indirectly in Section 4 and 5 through the transformation approach into definite equivalences (recall Fig. 1). First of all, we notice that $\top$, $\bot$, as well as $\wedge$, $\vee$, and $\neg$ are overloaded w.r.t. $\mathcal{T}^*$ and $\mathcal{C}^*$, but are interpreted in both cases purely classical, viz., denoting in every interpretation the whole universe, the empty set, set intersection, set union, and set complement. Given a typed feature structure $\theta = \langle x, s, \phi \rangle$, an interpretation $\mathcal{I}$, and a variable assignment α, we define the denotation of θ under $\mathcal{I}$ and α as follows:

$$[\![\theta]\!]^{\mathcal{I}}_{\alpha} := \{\alpha(x)\} \cap [\![s]\!]^{\mathcal{I}} \cap [\![\phi]\!]^{\mathcal{I}}_{\alpha} \tag{2}$$

This due to the fact that θ implicitly expresses a conjunction of x, s, and ϕ. Now, by abstracting from the variable assignment α, we obtain

$$[\![\theta]\!]^{\mathcal{I}} := \bigcup_{\alpha} [\![\theta]\!]^{\mathcal{I}}_{\alpha} \tag{3}$$

Let us quickly finish this informal definition by giving feature constraints $f \doteq \theta$ a denotation:

$$[\![f \doteq \theta]\!]^{\mathcal{I}}_{\alpha} := \{o \mid f^{\mathcal{I}}(o) \in [\![\theta]\!]^{\mathcal{I}}_{\alpha}\} \tag{4}$$

I.e., $f \doteq \theta$ denotes exactly those objects o on which f is defined, such that the value of f is in the denotation of θ. The interested reader is referred to Krieger 1995a for the full formal setting.

It is worth noting here that the transformation schema to come is independent of the underlying type logic, e.g., whether the existence of a greatest lower bound (GLB) is obligatory as for instance in TFS, or optional as in $\mathcal{TDL}$ (Krieger 1995b).

[2] Typed feature structures in our sense are a true extension of the ψ/ϵ-terms in LOGIN/LIFE since disjunctive typed feature structures (= ϵ-terms) are always typed here and the type of a typed feature structure can even be an arbitrary complex type expression $s \in \mathcal{T}^*$ (and must not be a mere type symbol $s \in \mathcal{T}$).

3 Definite Equivalences

Definite equivalences date back to Clark's completion schema (Clark 1978) and to the work of Höhfeld and Smolka on definite relations over constraint languages (Höhfeld and Smolka 1988) which has its root in the constraint logic programming paradigm (Jaffar and Lassez 1987). Definite equivalences extend work in logic programming in many ways: (i) we focus on true equivalences (contrary to, e.g., the Horn clause logic in PROLOG), (ii) we can mix $\wedge$, $\vee$, and $\exists$ (no need to go to DNF), and most important, (iii) we are no longer dedicated to a particular constraint system (e.g., the Herbrand structure/interpretation of PROLOG).
In our case, the constraint system is the equational theory of typed feature structures, i.e., constraints are essentially equations $s \doteq t$ and disequations $s \not\doteq t$, where s and t are terms. Note that in our special case, s and t are either variables or constant symbols because our signature does not contain real function symbols. It is worth noting that we restrict negation (in this paper) to the constraint level to guarantee the existence of a least and a greatest model of a definite program. Notice too that we do not treat negated constants since $x \doteq \neg c$ can be expressed equivalently as $x \not\doteq c$.
Having fixed the constraint inventory, we define a *definite equivalence* as

$$r(\vec{x}) \leftrightarrow \sigma \tag{5}$$

where $r \in R$ is a relation symbol, $\vec{x}$ a sequence of variables, and σ a definite formula.[3]
We then define the set of *definite formulae* σ, τ inductively by the following rewrite rules:

$$\sigma, \tau \ ::= \ s \doteq t \mid s \not\doteq t \mid r(\vec{x}) \mid \sigma \wedge \tau \mid \sigma \vee \tau \mid \exists \vec{x} \,.\, \sigma \tag{6}$$

A *definite program* P is a finite set of definite equivalences $\{r_i(\vec{x_i}) \leftrightarrow \sigma_i \mid i \in \mathbf{N}\}$, such that

1. variables in $\vec{x}$ are pairwise disjoint,
2. σ contains at most the free variables $\vec{x}$, and
3. for every $r \in R$, P contains exactly one equivalence $r(\vec{x}) \leftrightarrow \sigma$.

Clearly, the set of predicate symbols R of P is finite, since P is finite.

4 Transformation Schema

Our transformation schema makes no assumption about restrictions on the type hierarchy, i.e., we are not enforced that it must be, for instance, a lower semi-lattice as in ALE (Carpenter and Penn 1994) or any arbitrary partial order, as assumed in $\mathcal{TDL}$. However, it must preserve *incompatibilities* between types, so that we can abstract to a certain extent from the underlying type logics (see below).
We are now ready to transform an arbitrary type system into a definite program. The idea is simple and intuitive: every *type* $t \in \mathcal{T}$ is interpreted as a *unary predicate* $t(x)$ and every *feature* $f \in \mathcal{F}$ as a *binary relation* $f(x, y)$.
Now let Ω be an arbitrary type system that employ negation on the constraint level via negated atoms and negated coreferences only. The overall transformation schema is as follows:

[3] Of course, the free variables $\vec{x}$ will be interpreted universally.

1. Substitute every $\omega = (s := \langle x, t, \phi \rangle) \in \Omega$ with the definite equivalence p_ω given by

$$p_\omega := (s(x) \leftrightarrow \exists x_1, \ldots, \exists x_n \,.\, (\mathit{trans}(t, x) \wedge \mathit{trans}(\phi, x)))$$

   where $x_1, \ldots, x_n$ are the free variables within ϕ minus x, plus those that are introduced by *trans* in case of paths (see below).

2. Then replace Ω by the definite program P, where

$$P := \bigcup_{\omega \in \Omega} p_\omega$$

3. Let $\{S_i \mid S_i \subseteq \mathcal{T}, 1 \leq i \leq n\}$ be the set of sets of incompatible types. To preserve incompatibility, extend P by the following equivalence:

$$P := P \cup \left\{ \bot(x) \leftrightarrow \bigvee_{i=1}^{n} S_i(x) \right\}$$

   $S_i(x)$ means $s_1(x) \wedge \cdots \wedge s_m(x)$, i.e., $\mathrm{GLB}(s_1, \ldots, s_m) = \bot$, where $S_i = \{s_1, \ldots, s_m\}$.

Defining *trans* for complex types is straightforward (variables y are handled here):

- $\mathit{trans}(t, x) := t(x)$ **iff** $t \in \mathcal{T}$
- $\mathit{trans}(y, x) := (x \doteq y)$ **iff** $y \in \mathcal{V}$
- $\mathit{trans}(\neg y, x) := (x \not\doteq y)$ **iff** $y \in \mathcal{V}$
- $\mathit{trans}(a, x) := (x \doteq a)$ **iff** $a \in \mathcal{A}$
- $\mathit{trans}(\neg a, x) := (x \not\doteq a)$ **iff** $a \in \mathcal{A}$
- $\mathit{trans}(t_1 \wedge \cdots \wedge t_n, x) := \mathit{trans}(t_1, x) \wedge \cdots \wedge \mathit{trans}(t_n, x)$
- $\mathit{trans}(t_1 \vee \cdots \vee t_n, x) := \mathit{trans}(t_1, x) \vee \cdots \vee \mathit{trans}(t_n, x)$

Constructing *trans* for feature constraints is also nothing special:

- $\mathit{trans}(\epsilon \doteq \langle y, s, \phi \rangle, x) := \mathit{trans}(y, x) \wedge \mathit{trans}(s, x) \wedge \mathit{trans}(\phi, x)$
- $\mathit{trans}(f \doteq \langle y, s, \phi \rangle, x) := f(x, y) \wedge \mathit{trans}(s, y) \wedge \mathit{trans}(\phi, y)$ **iff** $f \in \mathcal{F}$
- $\mathit{trans}(f \doteq \langle x, s, \phi \rangle, x_0) := \left(\bigwedge_{i=1}^{n-1} (f_i(x_{i-1}, x_i) \wedge \top(x_i)) \right) \wedge \mathit{trans}(f_n \doteq \langle x, s, \phi \rangle, x_{n-1})$, **iff** $f \in \mathcal{F}^*$, where $f = f_1 \cdots f_n$, $f_1, \ldots, f_n \in \mathcal{F}$, and $x_1, \ldots, x_{n-1}$ are *new* variables
- $\mathit{trans}(\phi_1 \wedge \cdots \wedge \phi_n, x) := \mathit{trans}(\phi_1, x) \wedge \cdots \wedge \mathit{trans}(\phi_n, x)$
- $\mathit{trans}(\phi_1 \vee \cdots \vee \phi_n, x) := \mathit{trans}(\phi_1, x) \vee \cdots \vee \mathit{trans}(\phi_n, x)$

$$
\begin{aligned}
\mathit{likes}(z) \leftrightarrow\ & \exists z_1, \ldots, \exists z_{13}, \exists x, \exists y\,. \\
& \mathit{main}(z) \wedge \mathit{3rdsng}(z) \wedge \mathit{strict\text{-}trans}(z) \wedge \\
& \mathtt{PHON}(z, z_1) \wedge \top(z_1) \wedge \\
& \mathtt{SYN}(z, z_2) \wedge \top(z_2) \wedge \\
& \mathtt{LOC}(z_2, z_3) \wedge \top(z_3) \wedge \\
& \mathtt{SUBCAT}(z_3, z_4) \wedge \mathit{cons}(z_4) \wedge \\
& \mathtt{FIRST}(z_4, z_5) \wedge \top(z_5) \wedge \\
& \mathtt{REST}(z_4, z_6) \wedge \mathit{cons}(z_6) \wedge \\
& \mathtt{FIRST}(z_6, z_7) \wedge \top(z_7) \wedge \\
& \mathtt{REST}(z_6, z_8) \wedge \top(z_8) \wedge \\
& \mathtt{SEM}(z, z_9) \wedge \top(z_9) \wedge \\
& \mathtt{CONT}(z_9, z_{10}) \wedge \top(z_{10}) \wedge \\
& \mathtt{RELN}(z_{10}, z_{11}) \wedge \top(z_{11}) \wedge \\
& \mathtt{LIKER}(z_{10}, z_{12}) \wedge \top(z_{12}) \wedge \\
& \mathtt{LIKED}(z_{10}, z_{13}) \wedge \top(z_{13}) \wedge \\
& (z_1 \doteq \textit{“likes”}) \wedge \\
& (z_5 \doteq y) \wedge (z_7 \doteq x) \wedge (z_8 \doteq \mathit{nil}) \wedge \\
& (z_{11} \doteq \mathit{like}) \wedge (z_{12} \doteq x) \wedge (z_{13} \doteq y)
\end{aligned}
$$

Figure 2: *The outcome of applying trans to likes.*

Obviously, systems which employ arbitrary relations (relational dependencies) in their description language (e.g., TFS) can be given a formal semantics this way too (as long as they are definite), since we already stick to first-order logic in the transformation schema.

Let us give an example to see the outcome of this transformation schema. Consider the following simplified type definition of the transitive verb *likes* (Pollard and Sag 1987, p. 208).

$$
\mathit{likes} \equiv \left[\begin{array}{ll}
\multicolumn{2}{l}{\mathit{main} \wedge \mathit{3rdsng} \wedge \mathit{strict\text{-}trans}} \\
\multicolumn{2}{l}{\mathtt{PHON}\ \textit{“likes”}} \\
\multicolumn{2}{l}{\mathtt{SYN|LOC|SUBCAT}\ \left\langle \boxed{\mathtt{y}}, \boxed{\mathtt{x}} \right\rangle} \\
\mathtt{SEM|CONT} & \left[\begin{array}{l}
\mathtt{RELN}\ \mathit{like} \\
\mathtt{LIKER}\ \boxed{\mathtt{x}} \\
\mathtt{LIKED}\ \boxed{\mathtt{y}}
\end{array}\right]
\end{array}\right] \qquad (7)
$$

Applying *trans* to (7) yields the definite equivalence depicted in Fig. 2.

Clearly, merely defining definite equivalences on the basis of the predicate/type symbols $t \in \mathcal{T}$ is only one part of the whole transformation story—notice that the set of relation symbols R of a definite program is given by $R = \mathcal{T} \uplus \mathcal{F}$. I.e., we have to give also definite equivalences for every feature $f \in \mathcal{F}$. In the previous example, for instance, the value of the FIRST feature at the end of path SYN|LOC|SUBCAT is (in general) unrestricted (i.e., $= \top$). Furthermore, FIRST is (in general) only appropriate for feature structures of type *cons*. Thus we have the following definite equivalence for FIRST:

$$\begin{array}{c} \texttt{FIRST}(x,y) \leftrightarrow cons(x) \wedge \top(y) \\ \swarrow \quad \searrow \\ \text{intro type} \quad \text{value type} \end{array} \tag{8}$$

In case that a feature f is specified in different, subsumption-incomparable types, i.e., there is no unique, most general type which introduces f, we have to list these alternatives disjunctively:

$$f(x,y) \leftrightarrow \bigvee_{i=1}^{n} (intro_i(x) \wedge value_i(y)) \text{ and } n > 1 \tag{9}$$

Here, f is exactly what CUF (Dörre and Dorna 1993) calls a *polyfeature*. Of course, such polyfeatures violate the *feature introduction* condition in Carpenter's book (Carpenter 1992, p. 86). Now, in order to complete our schema *trans*, we have to extend the definite program P by adding definite equivalences for every feature $f \in \mathcal{F}$:

4. For every $f \in \mathcal{F}$, let T_f denote the set of most general, subsumption-incomparable types which introduce feature f. Then extend P by the following definite equivalences:

$$P := P \cup \bigcup_{f \in \mathcal{F}} \left\{ f(x,y) \leftrightarrow \bigvee_{t \in T_f} t(x) \wedge trans(t@f, y) \right\}$$

   $t@f$ yields the value (type) of feature f in the type definition/appropriateness specification for t.[4]

In case that $r \in \mathcal{T}$ is undefined (has no definition in the definite program P), we extend P by the following definition:

$$r(x) \leftrightarrow r(x) \tag{10}$$

Recall that this is due to the fact that for every $r \in \mathcal{T} \uplus \mathcal{F}$, there must be exactly one equivalence $(r(\vec{x}) \leftrightarrow \sigma) \in P$.

If we only know that a type s is the direct subtype of $\top$ but nothing more, i.e., $s := \langle x, \top, \top \rangle$, *trans* will clearly yield the following equivalence:

$$s(x) \leftrightarrow \top(x) \tag{11}$$

This might be OK on first sight, but remember that the equivalence sign here denotes equality. Thus for a given interpretation $\mathcal{I}$, we would have that $s^{\mathcal{I}} = \top^{\mathcal{I}}$ (which must not be the case). Now assume that our type system contains a similar definition: $s' := \langle x, \top, \top \rangle$. Clearly, applying *trans* to the definition of s' yields $s'(x) \leftrightarrow \top(x)$. Because s and s' have the same right-hand side, it is legal to infer that $s^{\mathcal{I}} = s'^{\mathcal{I}}$, for every interpretation $\mathcal{I}$ (which is obviously strange).

[4]Notice that @ here can be seen as an extension of Carpenter's appropriate function *Approp*, in that it is total and now maps to the set of complex type expressions $\mathcal{T}^*$, i.e., $@ : \mathcal{T} \times \mathcal{F} \longmapsto \mathcal{T}^*$. Note further that because we allow for complex type expressions, we must therefore apply *trans* to the value of $t@f$.

In order to overcome this problem, we must encode the information that s and s' are *subtypes* of $\top$. There are several ways to achieve this. We will propose the hereditary subset encoding $\mathcal{H}(\mathcal{T}) = \{\underline{t} \mid t \in \mathcal{T}\}$ of $\langle \mathcal{T}, \preceq \rangle$ (see Grätzer 1978, p. 61 and Euler 1989, pp. 34). The idea here is that a type $t \in \mathcal{T}$ can be represented through the set $\underline{t}$ of types which are smaller or equal than t:

$$\underline{t} := \{s \in \mathcal{T} \mid s \preceq t\} \tag{12}$$

Obviously $\langle \mathcal{H}(\mathcal{T}), \subseteq \rangle$ forms a complete distributive lattice, where the greatest lower bound operation corresponds to set intersection and the least upper bound to set union. By interpreting the hereditary sets disjunctively, we can now replace the first step of the transformation schema in that p_ω is extended by an additional clause which encodes the subtypes $\underline{s}$ of a given type s through a special new feature/binary predicate *type*:

$$p_\omega := (s(x) \leftrightarrow \exists x_1, \ldots, \exists x_n, \exists y \,.\, \mathit{trans}(t, x) \wedge \mathit{trans}(\phi, x) \wedge \mathit{type}(x, y) \wedge y \doteq \bigvee_{u \in \underline{s}} u)) \tag{13}$$

Coming back to our example, *trans* now correctly distinguishes between s and s':

$$\begin{array}{l} s(x) \leftrightarrow \exists y \,.\, \top(x) \wedge \mathit{type}(x, y) \wedge y \doteq \bigvee_{t \in \underline{s}} t \\ s'(x) \leftrightarrow \exists y \,.\, \top(x) \wedge \mathit{type}(x, y) \wedge y \doteq \bigvee_{t \in \underline{s'}} t \end{array} \tag{14}$$

5 Greatest Model Semantics

In general, one has an intuition what the meaning of a definite program is. Such a meaning is given by an interpretation of the symbols of a first-order language (atoms, functions, relations) over a specific domain. Formally an *interpretation* (or a *first-order structure*) $\mathcal{I}$ is given by a pair $\langle D^\mathcal{I}, \cdot^\mathcal{I} \rangle$, where $D^\mathcal{I}$ is some domain of discourse over which the variables range and $\cdot^\mathcal{I}$ an interpretation function. Furthermore, let $\alpha : \mathcal{V} \longmapsto D^\mathcal{I}$ be a *variable assignment* (or *valuation*) and let $V^\mathcal{I}$ abbreviate the set of all valuations into $\mathcal{I}$. Clearly, in our case we have

- **if** $a \in \mathcal{A}$ **then** $a^\mathcal{I} \in D^\mathcal{I}$
- **if** $t \in \mathcal{T}$ **then** $t^\mathcal{I} \subseteq D^\mathcal{I}$
- **if** $f \in \mathcal{F}$ **then** $f^\mathcal{I} \subseteq D^\mathcal{I} \times D^\mathcal{I}$

Normally, one is interested in interpretations $\mathcal{I}$ for which every formula p of a definite program P expresses a true statement in $\mathcal{I}$. These interpretations are called *models* of P. If $\mathcal{I}$ is a model of P, we write

$$\mathcal{I} \models P :\Longleftrightarrow \forall p \in P \,.\, \mathcal{I} \models p :\Longleftrightarrow \forall \alpha \in V^\mathcal{I}, \forall p \in P \,.\, \mathcal{I}, \alpha \models p \tag{15}$$

i.e., every p must be *valid* in $\mathcal{I}$, or equivalently, all valuations into $\mathcal{I}$ are *solutions* of p in $\mathcal{I}$.

We have already noted that under the usual Herbrand interpretation, models of P can be characterized through sets of ground atoms over the (complete) Herbrand base which are logical consequences of P (see Lloyd 1987). However, since we subscribe to the framework of definite equivalences, we are no longer dedicated to ground atoms, but instead would like

to propose *rational feature trees* (or *regular feature trees*) as the natural interpretation of typed feature structures.[5] Recently, rational feature trees have been formally investigated in the constraint system FT (Aït-Kaci et al. 1992), due to Colmerauer's work in PROLOG II (Colmerauer 1982).
Now let $\Re = \Re(\Sigma)$ denote the set of all (possible infinite) rational feature trees w.r.t. signature Σ. A rational feature tree is a feature tree which (i) has only finitely many subtrees and (ii) is only finitely branching. A feature tree is a tree whose edges are labelled with features and whose nodes are decorated with types.[6] Furthermore, feature trees are "functional" in that a node is not allowed to have several edges which bear the same feature symbol. Formally, a *feature tree* $\triangle \in \Re$ can be seen as a partial function $\triangle : \mathcal{F}^* \longmapsto \mathcal{T} \cup \mathcal{A}$, where $dom(\triangle)$ is a tree domain.[7] A *tree domain* F is a nonempty set $F \subseteq \mathcal{F}^*$ that is *prefix-closed*, i.e., **if** $fg \in F$ **then** $f \in F$. Let $F_\triangle$ denote the tree domain of $\triangle$. A *subtree* $f\triangle$ of a feature tree $\triangle$ at path $f \in F_\triangle$ is the feature tree $\triangle_f$ which can be obtained by following f in $\triangle$. Given a path $f \in \mathcal{F}^*$, we define the *length* of a path $|\cdot| : \mathcal{F}^* \longmapsto \mathbf{N}$ as follows: $|\epsilon| := 0$; $|f| := 1$ **iff** $f \in \mathcal{F}$; and $|fg| := |f| + |g|$.

Following Aït-Kaci et al. 1992, we then define the *feature tree interpretation* $\mathcal{I}$ as follows:

- $D^\mathcal{I} = \Re(\Sigma)$ (i.e., the universe of $\mathcal{I}$ is the set of all feature trees w.r.t. Σ)
- $\triangle \in s^\mathcal{I}$ **iff** $\triangle(\epsilon) \preceq s$ (i.e., the root of $\triangle$ is labelled with a subtype of s)
- $(\triangle, \triangle_f) \in f^\mathcal{I}$ **iff** $f \in F_\triangle$ **and** $f\triangle = \triangle_f$ (i.e., $\triangle_f$ is a subtree of $\triangle$ at f)
- **if** $(f\triangle)(\epsilon) \in \mathcal{A}$ **then** $\nexists g \in \mathcal{F} \,.\, fg \in F_\triangle$ (i.e., no feature is defined on an atom)

Let us now obtain a deeper insight into the models of a definite program P. Constructing a (specific) model $\mathcal{M}$ of P and proving certain properties about $\mathcal{M}$ is often achieved by a fixpoint construction over a certain monotonic (or even continuous) function T_P, the so-called *associated function of* P (Lloyd 1987). T_P itself operates on the powerset of our domain, mapping interpretations into interpretations, i.e.,

$$T_P : \wp(\Re) \longmapsto \wp(\Re) \tag{16}$$

The idea here is that in the "limit", T_P will exactly enumerate those objects which are logical consequences of P. Thus we are interested in interpretations $\mathcal{M}$, where

$$T_P(\mathcal{M}) = \mathcal{M} \tag{17}$$

[5] Feature trees are even present in imperative programming languages like PASCAL, since they model records. Records, when compared with PROLOG-like (constructor) terms, have several advantages: (i) we are no longer restricted to a fixed arity; (ii) the tags of a record allow us to interchange argument positions; (iii) the type hierarchy gives us the flexibility to successfully unify two records that are not labelled with the same symbol. It is worth noting that rational trees have even been proposed for one of the earliest feature structure formalisms, viz., PATR-II; see Pereira and Shieber 1984, p. 126.

[6] Clearly, because feature trees are trees, they have a unique root and in addition, are connected.

[7] To make things easier, we restrict the range of $\triangle$ to $\mathcal{T} \cup \mathcal{A}$ instead of assuming the superset $\mathcal{T}^*$. This can be done without loss of generality for a finite type system by adding further definitions, extending the type lattice, and substituting complex type expressions through the corresponding denotation-preserving type symbols (see Krieger 1995b).

since in this case, $\mathcal{M}$ has been "saturated", and so we can stop our iteration process via T_P (see below). We call $\mathcal{M}$ a *fixpoint* of T_P and for a suitable T_P, $\mathcal{M}$ will then be exactly a model of P. Now, how will T_P look like? For a given interpretation $\mathcal{I}$ and valuation α, $T_P(\mathcal{I})$ will be exactly those feature trees $\alpha(x)$ which

1. satisfy the right-hand side of some definite equivalence $p \in P$ w.r.t. $\mathcal{I}$, and
2. are labelled with the left-hand side symbol of p at the root.

Actually, the second condition is too strong, since a type system has to be interpreted w.r.t. the underlying inheritance hierarchy. Thus we require that $\alpha(x)$ must be decorated with a *subtype* of the corresponding type symbol. This idea leads us to the following definition for T_P:

$$T_P(\mathcal{I}) = \bigcup_{s \in \mathcal{T}} \bigcup_{\alpha \in V^{\mathcal{I}}} \{\alpha(x) \in \Re(\Sigma) \mid s(x) \leftrightarrow \sigma \in P \textbf{ and } \mathcal{I}, \alpha \models \sigma \textbf{ and } (\alpha(x))(\epsilon) \preceq s\} \tag{18}$$

Now, if $T_P(\mathcal{I}) = \mathcal{J}$ and $s(x) \leftrightarrow \sigma \in P$, we have

$$s^{\mathcal{J}} = \bigcup_{\alpha \in V^{\mathcal{I}}} \{\alpha(x) \mid \mathcal{I}, \alpha \models \sigma \textbf{ and } (\alpha(x))(\epsilon) \preceq s\} \tag{19}$$

The satisfaction relation $\models$ used above is defined inductively over the structure of definite formulae (6), relatively to $\mathcal{I}$ and $\alpha : \mathcal{V} \longmapsto \mathcal{I}$:

- $\mathcal{I}, \alpha \models x \doteq a$ **iff** $(\alpha(x))(\epsilon) = a$, for $x \in \mathcal{V}$ and $a \in \mathcal{A}$
- $\mathcal{I}, \alpha \models x \doteq y$ **iff** $\alpha(x) = \alpha(y)$, for $x, y \in \mathcal{V}$
- $\mathcal{I}, \alpha \models x \not\doteq a$ **iff** $(\alpha(x))(\epsilon) \neq a$, for $x \in \mathcal{V}$ and $a \in \mathcal{A}$
- $\mathcal{I}, \alpha \models x \not\doteq y$ **iff** $\alpha(x) \neq \alpha(y)$, for $x, y \in \mathcal{V}$
- $\mathcal{I}, \alpha \models r(x)$ **iff** $\{\alpha(x) \mid r(x) \leftrightarrow \sigma \in P \textbf{ and } \mathcal{I}, \alpha \models \sigma \textbf{ and } (\alpha(x))(\epsilon) \preceq r\} \neq \emptyset$
- $\mathcal{I}, \alpha \models \sigma \wedge \tau$ **iff** $\mathcal{I}, \alpha \models \sigma$ **and** $\mathcal{I}, \alpha \models \tau$
- $\mathcal{I}, \alpha \models \sigma \vee \tau$ **iff** $\mathcal{I}, \alpha \models \sigma$ **or** $\mathcal{I}, \alpha \models \tau$
- $\mathcal{I}, \alpha \models \exists x \,.\, \sigma$ **iff** $\{\alpha(x/a) \mid a \in \mathcal{A} \textbf{ and } \mathcal{I}, \alpha \models \sigma\} \neq \emptyset$

It is worth noting that the power set of all feature tree interpretations $\wp(\Re(\Sigma))$ of P (w.r.t. signature Σ) forms a *complete lattice*, partially ordered under set inclusion. Thus, for all $\mathcal{I}, \mathcal{J} \in \wp(\Re)$, we can define an ordering $\subseteq$ between interpretations:

$$\mathcal{I} \subseteq \mathcal{J} \Longleftrightarrow \forall r \in R \,.\, r^{\mathcal{I}} \subseteq s^{\mathcal{J}} \tag{20}$$

The top element of this lattice is $\Re$ and the bottom element is $\emptyset$. Furthermore, $\wp(\Re)$ is even a *distributive lattice*, where *meet* $\sqcap$ and *join* $\sqcup$ correspond to set intersection $\cap$ and set union $\cup$, resp. (see Davey and Priestley 1990, p. 131).

Now, by applying Tarski's famous fixpoint theorem, we immediately know that T_P will have a *least fixpoint* $lfp(T_P)$ and a *greatest fixpoint* $gfp(T_P)$, assuming that T_P (i) operates on a complete lattice and (ii) is *monotonic*, i.e.,

$$\forall \mathcal{I}, \mathcal{J} \in \wp(\Re) \,.\, \mathcal{I} \subseteq \mathcal{J} \Longrightarrow T_P(\mathcal{I}) \subseteq T_P(\mathcal{J}) \tag{21}$$

As we have noted earlier, $\wp(\Re)$ forms a complete lattice. Obviously, T_P is also monotonic: given $\mathcal{I} \subseteq \mathcal{J}$ and $(s(x) \leftrightarrow \sigma) \in P$, assume that there exists $\alpha(x) \in \mathcal{J}$, such that $(\alpha(x))(\epsilon) \preceq s$ and $\mathcal{J}, \alpha \models \sigma$. Clearly, because $\mathcal{I} \subseteq \mathcal{J}$, it is possible that there exists **no** $\alpha(x) \in \mathcal{I}$, such that $\mathcal{I}, \alpha \models \sigma$. Given the definition of T_P, we thus have $T_P(\mathcal{I}) \subseteq T_P(\mathcal{J})$, i.e., T_P is monotonic.

Usually, there is more than just one model of P, since normally T_P has more than one fixpoint. Two models obviously stand out: the *least* and the *greatest* one. Contrary to most other related treatments (see Section 7), we have decided to choose a *greatest model semantics* for typed feature structures. This kind of semantics has in fact been implemented in the typed feature formalism $\mathcal{TDL}$. Our choice was mainly based on the following two observations:

1. A type system (grammar/lexicon) restricts the set of admissible utterances by specifying constraints to be fulfilled. Exactly the greatest fixpoint models utterances which are licensed only by the *linguistic* constraints. Other fixpoints, especially the least one, make further *non-linguistically* motivated assumptions about described objects; for instance, the least fixpoint rules out *cyclic* structures or assigns an empty denotation to types which are not founded on atoms (see below). We think that as long as such extra-linguistic constraints can not be formulated in the grammar/lexicon, they should not come into play indirectly via the choice of a special fixpoint.[8]

2. Speaking from an operational point of view, we notice that *type expansion* (Krieger and Schäfer 1995) or *sort unfolding* (Aït-Kaci et al. 1993) are operations testing for the *satisfiability* of typed feature structures. This is done by making constraints defined on types explicit (and thus very similar to Bob Carpenter's *total well-typedness*; Carpenter 1992). Operationally, this is achieved by adding more and more information to the input structure (via unification), substituting types through their "improved" definitions. Assuming a fair type unfolding strategy, we then repeat this process for all types in parallel, until a failure is detected (possibly infinitely long). But in fact, such a strategy corresponds to the construction of the greatest fixpoint, namely, to start with $\top$ at the beginning of the fixpoint iteration, as opposed to $\bot$ in case of the least fixpoint (see below). Notice that this process narrows down the set of possible solutions, since unification adds more and more constraints during the iteration (this is characteristic for the approximation of the greatest fixpoint). This idea is very intuitive and we guess that many systems who have proposed a least fixpoint semantics have nevertheless implemented a greatest one (see Section 7). Exactly the greatest fixpoint semantics is valid under the *complete* expansion strategy in $\mathcal{TDL}$ (Krieger and Schäfer 1995).

[8] Clearly, explicitly excluding cycles in feature logics is usually not possible (*universal* quantification over paths!). There is one notable exception, viz., Treinen & Backofen's universal feature theory F, an undecidable logic which merely possesses a 3-place descriptive primitive $\cdot[\cdot]\cdot$, the so-called "generalized feature constraint" (see Treinen 1993 and Backofen 1994; Johnson 1988 has already considered the existential fragment of F). In F (actually, the relational extension of F), cyclic structures can be easily excluded: $non\text{–}cyc(x) := \lambda x \,.\, (\forall p, \nexists y \,.\, x[p]y \wedge x \doteq y)$. *non–cyc* then should hold for all objects x and can be added as an axiom: $\forall x \,.\, non\text{–}cyc(x)$. Of course, we can formulate such meta-constraints in a less specialized logic, e.g., second-order PL.

Before we move on to a formal characterization of the least and the greatest model, we want to present a few examples of type systems, where the least and the greatest interpretation of the corresponding definite program totally differ w.r.t. the rational tree domain (we will not consider the *type* feature introduced in the previous section).

Example 1. Let r be an *undefined* type so that the corresponding definite program P is given by $P = \{(r(x) \leftrightarrow r(x)\}$ (see Section 4). The least model of P is given by $\emptyset$, whereas the greatest one is $\Re$.

Example 2. Let s and t be two mutually *recursive* types which are not "grounded" on atoms; e.g., $\Omega = \{s := \langle x, \top, f \doteq t\rangle,\ t := \langle x, \top, g \doteq s\rangle\}$. The least model of the corresponding definite program is again given by the empty set; however, the greatest model contains the following two (infinite) rational feature trees:

$$\left\{ \begin{array}{ll} \bullet s & \bullet t \\ \downarrow f & \downarrow g \\ \bullet t & \bullet s \\ \downarrow g & \downarrow f \\ \bullet s & \bullet t \\ \downarrow f \quad , & \downarrow g \\ \bullet t & \bullet s \\ \downarrow g & \downarrow f \\ \vdots & \vdots \\ \vdots & \vdots \end{array} \right\}$$

Example 3. Consider the *coreference cycle* in the following non-recursive type definition: $s := \langle x, \top, f \doteq x\rangle$. The corresponding definite program is obviously given by $P = \{s(x) \leftrightarrow \exists y\,.\,\top(x) \wedge f(x,y) \wedge x \doteq y\}$. Again, the least model is empty, contrary to the greatest one:

$$\left\{ \begin{array}{l} \bullet s \\ \downarrow f \\ \bullet s \\ \downarrow f \\ \bullet s \\ \vdots \end{array} \right\}$$

In order to formally understand the differences between the least and the greatest fixpoint of T_P, we need some additional mathematical apparatus. First of all, we notice that because models $\mathcal{M}$ of a definite program P are elements of $\wp(\Re)$ and because $\wp(\Re)$ forms a complete distributive lattice, the intersection/meet and the union/join of models are again models for P. Especially, we have

$$\sqcap\{\mathcal{M} \mid T_P(\mathcal{M}) = \mathcal{M}\} = \cap\{\mathcal{M} \mid T_P(\mathcal{M}) = \mathcal{M}\} = \mathit{lfp}(T_P) \tag{22}$$

$$\sqcup\{\mathcal{M} \mid T_P(\mathcal{M}) = \mathcal{M}\} = \cup\{\mathcal{M} \mid T_P(\mathcal{M}) = \mathcal{M}\} = \mathit{gfp}(T_P) \tag{23}$$

Thus usually, $\mathit{lfp}(T_P) \subset \mathit{gfp}(T_P)$ because $\cap\{\mathcal{M} \mid T_P(\mathcal{M}) = \mathcal{M}\} \subset \cup\{\mathcal{M} \mid T_P(\mathcal{M}) = \mathcal{M}\}$.
We then need the notion of the ordinal powers of T_P (see, e.g., Lloyd 1987, pp. 28). *Ordinals* (or just *ordinal numbers*) are a set-theoretic characterization of what we use to count with.

The *first ordinal* 0 is defined to be the empty set $\emptyset$. We now define an ordinal α inductively as $(\alpha-1)\cup\{\alpha-1\}$. One can verify that 1 is equal to $\{0\}$ because $1 = 0\cup\{0\} = \emptyset\cup\{\emptyset\} = \{\emptyset\}$. The next ordinals are $2 = \{\emptyset, \{\emptyset\}\} = \{0, 1\}$, $3 = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\} = \{0, 1, 2\}$, etc. These ordinals are the *finite* ordinals, the non-negative integers. The first *infinite* ordinal is $\omega = \{0, 1, 2, \ldots\}$. We can specify a *total order* $<$ on ordinals, where $\alpha < \beta$ if $\alpha \in \beta$.
If α is an ordinal, the *successor ordinal* of α is the ordinal $\alpha+1$, the least ordinal greater than α. An ordinal α is called a *limit ordinal* if it is not the successor of any ordinal. The smallest limit ordinal is 0 by definition; the next more important limit ordinal is ω (see above). The successor ordinal of ω is $\omega+1 = \omega\cup\{\omega\}$, followed by $\omega+2$, $\omega+3$ etc. The next limit ordinal is $\omega 2$ which is given by the set $\{n \mid n \in \omega\} \cup \{\omega + n \mid n \in \omega\}$. After $\omega 2$ comes $\omega 2 + 1$, $\omega 2 + 2$, and so on. We now have the prerequisites to define the *ordinal power* of a monotonic function on a complete lattice. Let S be a complete lattice and $T_P : S \longmapsto S$ be monotonic. We define the *upward (ordinal) power* of T_P as follows:

- $T_P\uparrow 0 = \bot$
- $T_P\uparrow\alpha = T_P(T_P\uparrow(\alpha - 1))$, if α is a successor ordinal
- $T_P\uparrow\alpha = \sqcup\{T_P\uparrow\beta \mid \beta < \alpha\}$, if α is a limit ordinal

The *downward (ordinal) power* of T_P is defined in the usual dual way:

- $T_P\downarrow 0 = \top$
- $T_P\downarrow\alpha = T_P(T_P\downarrow(\alpha - 1))$, if α is a successor ordinal
- $T_P\downarrow\alpha = \sqcap\{T_P\downarrow\beta \mid \beta < \alpha\}$, if α is a limit ordinal

Clearly, $\top$ ($\bot$) here means $\Re(\Sigma)$ ($\emptyset$, resp.).

With the help of $T_P\uparrow\alpha$ and $T_P\downarrow\alpha$, we can even estimate the least and the greatest fixpoint (see Lloyd 1987):

$$T_P\uparrow\alpha \subseteq \mathit{lfp}(T_P) \subseteq \mathit{gfp}(T_P) \subseteq T_P\downarrow\alpha, \tag{24}$$

for any ordinal α.
We finally need two additional notions to characterize the least and the greatest fixpoint in terms of ordinal powers, viz., directedness and continuity. Let S be a complete lattice and $S' \subseteq S$. We say that S' is *directed* iff every finite subset of S' has an upper bound in S'. Now let $T_P : S \longmapsto S$ be a mapping. We say that T_P is *continuous* iff

$$T_P(\sqcup S') = \sqcup T_P(S') := \sqcup\{T_P(s) \mid s \in S'\}, \tag{25}$$

for every directed subset S' of S.

If T_P is continuous, Kleene's fixpoint theorem now tells us that

$$\mathit{lfp}(T_P) = T_P\uparrow\omega \tag{26}$$

i.e., the least fixpoint of a continuous mapping can be obtained in at least ω-many iteration steps. However, a dual result must not hold for the greatest fixpoint in which we are primarily interested, as described above.[9]
Now, could at least we guarantee that $gfp(T_P) = T_P{\downarrow}\omega$ is the case for our special setting? In fact, this is the case because one can straightforwardly show that $\langle \Re(\Sigma), d\rangle$ is a *compact metric space* under a suitable metric d.

Before we will prove this proposition, let us first recall the definition of a metric space (cf., e.g., Johnsonbaugh and Pfaffenberger 1981, p. 117). Metric spaces can be seen as a generalization of already familiar spaces, like, for example, the real numbers $\mathbf{R}$ together with the Euclidean metric $d(x,y) = |x - y|$ (i.e., a metric is a kind of distance measure between elements of a metric space). Formally, a *metric* d of a set S is a mapping $d : S \times S \longmapsto \mathbf{R}_0^+$, which satisfies the following requirements:

- $d(s,t) = 0$ **iff** $s = t$, for all $s, t \in S$
- $d(s,t) = d(t,s)$, for all $s, t \in S$
- $d(s,u) \leq d(s,t) + d(t,u)$, for all $s, t, u \in S$

$\langle S, d\rangle$ is called a *metric space*. If furthermore

- $d(s,u) \leq max\{d(s,t),\ d(t,u)\}$, for all $s, t, u \in S$

is the case, d is an *ultrametric* and $\langle S, d\rangle$ is called an *ultrametric space*.

Following Lloyd 1987, pp. 175, we then define the notion of *truncation at depth* n of a feature tree $\triangle \in \Re$, depicted by $\dagger_n(\triangle)$. In order to indicate that a subtree of a feature tree has been cut off in a truncation, we introduce a new symbol $\blacksquare$ of arity 0 (i.e., an atom). Given a signature Σ and adding $\blacksquare$ to the set of atoms $\mathcal{A}$, we obtain an extended signature Σ'. Thus $\dagger_n$ is a mapping from $\Re(\Sigma)$ into $\Re(\Sigma')$. For a given feature tree $\triangle$ and a given $n \in \omega$, $\dagger_n$ yields the truncated feature tree of $\triangle$ at depth n. Remember that a feature tree $\triangle$ is a partial function $\triangle : \mathcal{F}^* \longmapsto \mathcal{T} \cup \mathcal{A}$, hence $\dagger_n$ returns a function, i.e.,

$$\dagger_n : (\mathcal{F}^* \longmapsto \mathcal{T} \cup \mathcal{A}) \longmapsto (\mathcal{F}^* \longmapsto \mathcal{T}^* \cup \mathcal{A} \cup \{\blacksquare\}) \tag{27}$$

The domain of $\dagger_n$ w.r.t. to a feature tree $\triangle$ now is given by

$$dom(\dagger_n(\triangle)) = \{f \in F_\triangle \mid |f| \leq n\} \tag{28}$$

With these comments in mind, the definition for $\dagger_n$ is easy ($f \in \mathcal{F}^*$):

$$\dagger_n(\triangle)(f) := \begin{cases} \triangle(f) & \textbf{iff } |f| < n \\ \blacksquare & \textbf{iff } |f| = n \end{cases} \tag{29}$$

Given the truncation function $\dagger_n$, we are now ready to define a metric d on typed feature trees. First of all, we notice that for feature trees $\triangle_1 \neq \triangle_2$, $\dagger_n(\triangle_1) \neq \dagger_n(\triangle_2)$, for some $n > 0$. Thus for $\triangle_1 \neq \triangle_2$, the set $\{n \mid \dagger_n(\triangle_1) \neq \dagger_n(\triangle_2)\}$ is not empty. In order to obtain the *least depth* at which $\triangle_1$ and $\triangle_2$ differ, we define

[9] See Lloyd 1987, p. 38, for an example of a program P, where $gfp(T_P) = T_P{\downarrow}(\omega + 1)$.

$$\dagger(\triangle_1, \triangle_2) := min\{n \mid \dagger_n(\triangle_1) \neq \dagger_n(\triangle_2)\} \tag{30}$$

Then it is easy to show that $\langle \Re, d \rangle$ is an ultrametric space under

$$d(\triangle_1, \triangle_2) := \begin{cases} 0 & \textbf{iff } \triangle_1 = \triangle_2 \\ 2^{-\dagger(\triangle_1, \triangle_2)} & \textbf{otherwise} \end{cases} \tag{31}$$

Finally, we need some additional definitions (perhaps familiar from high school mathematics; cf. Johnsonbaugh and Pfaffenberger 1981 for an appropriate introduction). We say that a sequence $\langle s_n \rangle_{n \in \omega}$ in a metric space $\langle S, d \rangle$ *converges to* (or *has limit*) $s \in S$ and write $\lim_{n \in \omega} s_n = s$ if for every $\epsilon > 0$ there exists an integer N such that if $n \geq N$ then $d(s_n, s) < \epsilon$. Now let $T \subseteq S$ and $t \in S$. t is called a *limit point* of T if there is a sequence $\langle s_n \rangle_{n \in \omega}$ such that for every $n \in \omega$, we have $s_n \in T$ and $\lim_{n \in \omega} s_n = t$. If furthermore every limit point of T belongs to T, we say that T is *closed* (in S). Otherwise T is called *open*. Given a bounded sequence $\langle s_n \rangle_{n \in \omega}$, we define the notion of the *limit superior* as follows (see Johnsonbaugh and Pfaffenberger 1981, pp. 69): $\limsup_{n \in \omega} s_n = \lim_{n \in \omega} t_n$, where $t_n = \sqcup_{k \in \omega} s_{n+k}$.
Given a metric space $\langle S, d \rangle$, we call S *compact* iff every sequence in S has a convergent subsequence. We call a subset T of S bounded if there exists a number A such that $d(x, y) \leq A$ for all $x, y \in T$. The important fact now is that $\langle S, d \rangle$ is compact **iff** S is closed and bounded (this is the so-called "Bolzano-Weierstrass characterization" for a compact metric space; see Johnsonbaugh and Pfaffenberger 1981, p. 151). Obviously, $\langle \Re(\Sigma), d \rangle$ is a compact metric space under topology d as defined above, since $\Re$ is (i) closed and (ii) bounded. $\Re$ is closed because if $\lim_{n \in \omega} \triangle_n = \triangle$ and each $\triangle_n \in \Re$, $\triangle$ must also be in $\Re$ since each $\triangle_n$ is finitely branching (maximal breadth is $|\mathcal{F}|$) and furthermore, there exists only a finite number of distinct subtrees w.r.t. signature Σ. Recall that $\mathcal{F}$, $\mathcal{T}$, and $\mathcal{A}$ are finite sets for a given definite program.[10] In addition, $\Re$ is bounded because $0 \leq d(\triangle_1, \triangle_2) \leq 1$ for all $\triangle_1, \triangle_2 \in \Re$ (notice that $0 \leq \dagger(\triangle_1, \triangle_2) \leq \omega$). Intuitively, $\Re(\Sigma)$ is compact because it even contains infinite feature trees—if $\Re$ would only contain finite ones, it could be the case that limits of sequences of finite terms are missing from $\Re$.

We now have to show that T_P as defined above is a continuous function. We need the following proposition.

Proposition 1 (INDEPENDENCE).
Let R be a directed subset of $\wp(\Re)$. Then $\{\triangle_1, \ldots, \triangle_n\} \subseteq \sqcup R$ **iff** $\{\triangle_1, \ldots, \triangle_n\} \subseteq \mathcal{I}$, for some $\mathcal{I} \in R$ $(n < \omega)$.

Proof.

"$\Longleftarrow$" $\mathcal{I} \in R$ and $\{\triangle_1, \ldots, \triangle_n\} \subseteq \mathcal{I}$,
thus $\mathcal{I} \subseteq \cup R = \sqcup R$,
i.e., $\{\triangle_1, \ldots, \triangle_n\} \subseteq \sqcup R$.

[10] It is worth noting that $\Re(\Sigma)$ is a *countable-infinite* set, because (i) $\mathcal{T}$ and $\mathcal{A}$ are finite, and so is $\mathcal{T} \cup \mathcal{A}$, and (ii) rational trees have only finitely many subtrees and are finitely branching (since they are functional). To see why this is so, let $\Re_n$ denotes the set of all feature trees of depth n w.r.t. signature Σ. Clearly, $\Re_n$ is a (countable) finite set, thus $\bigcup_{n=0}^{\omega} \Re_n$ must also be countable (but obviously infinite; see also Johnsonbaugh and Pfaffenberger 1981, pp. 30). But exactly this set is $\Re$.

"$\Longrightarrow$" $\{\triangle_1, \ldots, \triangle_n\} \subseteq \sqcup R = \cup R$ and R directed,
i.e., $\forall S \subseteq R, \exists r \in R, \forall s \in S\,.\, s \subseteq r$ (S finite);
now choose $s = \{\triangle_1, \ldots, \triangle_n\}$ and $r = \mathcal{I}$,
hence $\{\triangle_1, \ldots, \triangle_n\} \subseteq \mathcal{I}$, for some $\mathcal{I} \in R$.

Proposition 2 (CONTINUITY OF T_P).
$T_P : \wp(\Re) \longmapsto \wp(\Re)$ as defined by (18) is a continuous function, i.e., $T_P(\sqcup R) = \sqcup T_P(R)$, for every directed subset R of $\wp(\Re)$.

Proof.

$\triangle \in T_P(\sqcup R)$ **iff**
$\triangle \in \bigcup_{s \in \mathcal{T}} \bigcup_{\alpha \in V^{\cup R}} \{\alpha(x) \mid s(x) \leftrightarrow \sigma \in P,\ \cup R, \alpha \models \sigma,\ (\alpha(x))(\epsilon) \preceq s\}$ **iff**
$\triangle \in \bigcup_{s \in \mathcal{T}} \bigcup_{\alpha \in V^{\mathcal{I}}} \{\alpha(x) \mid s(x) \leftrightarrow \sigma \in P,\ \mathcal{I}, \alpha \models \sigma,\ (\alpha(x))(\epsilon) \preceq s\}$, for some $\mathcal{I} \in R$ **iff**
$\triangle \in T_P(\mathcal{I})$, for some $\mathcal{I} \in R$ **iff**
$\triangle \in \cup\{T_P(r) \mid r \in R\}$ **iff**
$\triangle \in \sqcup T_P(R)$.

Notice that Proposition 1 is used two times inside the proof of Proposition 2. Note further that this proof actually consists of several subcases, depending on the structure of the definite formula σ.

In order to finish this section, we need some additional propositions.

Proposition 3 (CLOSEDNESS OF T_P).
Let P be a definite program and $\mathcal{I}$ be a closed subset of $\Re$. Then $T_P(I)$ is a closed subset of $\Re$.

Proof.
Due to the fact that $\langle \Re, d \rangle$ is a compact metric space and T_P is continuous, we know that if $\mathcal{I}$ is closed then $\mathcal{I}$ is also compact (since it is bounded under d); hence $T_P(\mathcal{I})$ is compact (Johnsonbaugh and Pfaffenberger 1981, pp. 152) and thus closed.

Corollary 4 (CLOSEDNESS OF DOWNWARD POWER OF T_P).
Because T_P is closed, we immediately know that $T_P{\downarrow}n$ ($n \in \omega$) and even $T_P{\downarrow}\omega$ is closed.

We now give a characterization of the limit superior for closed sets.

Proposition 5 (LIMIT SUPERIOR).
If $\langle \mathcal{I}_n \rangle_{n \in \omega}$ is a decreasing sequence of closed subsets of $\Re$, then $\limsup_{n \in \omega} \mathcal{I}_n = \cap_{n \in \omega} \mathcal{I}_n$.

Proof.
Since every $\mathcal{I}_n$ is bounded, we have by definition that $\limsup_{n \in \omega} \mathcal{I}_n$ is equal to $\lim_{n \in \omega} \sqcup_{k \in \omega} \mathcal{I}_{n+k}$ which reduces to $\lim_{n \in \omega} \cup_{k \in \omega} \mathcal{I}_{n+k}$ in the metric space $\Re$. Since $\langle \mathcal{I}_n \rangle_{n \in \omega}$ is decreasing, this is equal to $\lim_{n \in \omega} \mathcal{I}_n$, and again, because $\mathcal{I}_n$ is decreasing, we finally have $\cap_{n \in \omega} \mathcal{I}_n$.

Using the limit superior, we can establish a weak continuity result for T_P.

Proposition 6 (DOWNWARD CONTINUITY OF T_P).
Let P be a definite program and $\langle \mathcal{I}_n \rangle_{n\in\omega}$ be a sequence of sets in $\Re$. Then $\limsup_{n\in\omega} T_P(\mathcal{I}_n) \subseteq T_P(\limsup_{n\in\omega} \mathcal{I}_n)$.

Proof.
Adaption of the proof in Lloyd 1987, p. 184.

Corollary 7 (INTERSECTION PROPERTY FOR T_P).
Let P be a definite program and $\langle \mathcal{I}_n \rangle_{n\in\omega}$ be a decreasing sequence of closed sets in $\Re$. Then $T_P(\cap_{n\in\omega}\mathcal{I}_n) = \cap_{n\in\omega} T_P(\mathcal{I}_n)$.

Proof.
By Proposition 5, $T_P(\cap_{n\in\omega}\mathcal{I}_n)$ is equal to $T_P(\limsup_{n\in\omega} \mathcal{I}_n)$, since the $\mathcal{I}_n$ are closed and decreasing. By Proposition, this set is a superset of $\limsup_{n\in\omega} T_P(\mathcal{I}_n)$ which is equal to $\cap_{n\in\omega} T_P(\mathcal{I}_n)$, since the $T_P(\mathcal{I}_n)$ are closed and decreasing. For the other direction, notice that T_P is monotonic (21), thus $T_P(\cap_{n\in\omega}\mathcal{I}_n) \subseteq \cap_{n\in\omega} T_P(\mathcal{I}_n)$.

With the help of the above given two corollaries, we can finally show that the greatest fixpoint of the associated continuous function T_P of a definite program P can be reached in at most ω-many steps.

Proposition 8 (GREATEST FIXPOINT OF T_P).
Let P be a definite program. Then $gfp(T_P) = T_P{\downarrow}\omega$.

Proof.
If $T_P{\downarrow}\omega$ is a fixpoint, it suffices to show that $T_P(T_P{\downarrow}\omega) = T_P{\downarrow}\omega$. By definition of the downward ordinal power, $T_P(T_P{\downarrow}\omega)$ is equal to $T_P(\cap_{n\in\omega} T_P{\downarrow}n)$; using the two corollaries, this reduces to $\cap_{n\in\omega} T_P(T_P{\downarrow}n)$ which is exactly $T_P{\downarrow}\omega$.

6 Nonmonotonically Defined Types

Without extending the formalism, we finally show that the translation into definite equivalences might even serve as the basis for a reformulation of *nonmonotonically defined types*. Such extensions (overwriting, defaults, etc.) have been claimed by theorists and practitioners to be extremely useful during the development of huge grammars and lexicons, in order to formulate subregularities and exceptions adequately.[11] On first sight, our thesis might stand in contrast to the fact that definite equivalences are conceived as a *monotonic* framework. However, by slightly modifying the compilation schema, we will see in a moment that the outcome of the transformation of a nonmonotonically defined type system is in fact a monotonic definite program.

The idea here is that the type hierarchy can be seen as a pure *transport medium*, allowing us to formulate constraints on types in a compact and redundant-free way. With the help of the type hierarchy, generalizations can be made at the appropriate levels of representation and inheritance plays the role of gathering information from supertypes. Under this view,

[11] We have neither the space to discus the pros and the cons of nonmonotonic extensions to monotonic unification-based NL formalism, nor can we address issues of their formalization; see, e.g., Bouma 1992, Carpenter 1993, Copestake 1993, Daelemans et al. 1992, Evans and Gazdar 1990, Flickinger 1987, Russell et al. 1992, or Young and Rounds 1993 for more arguments on this theme.

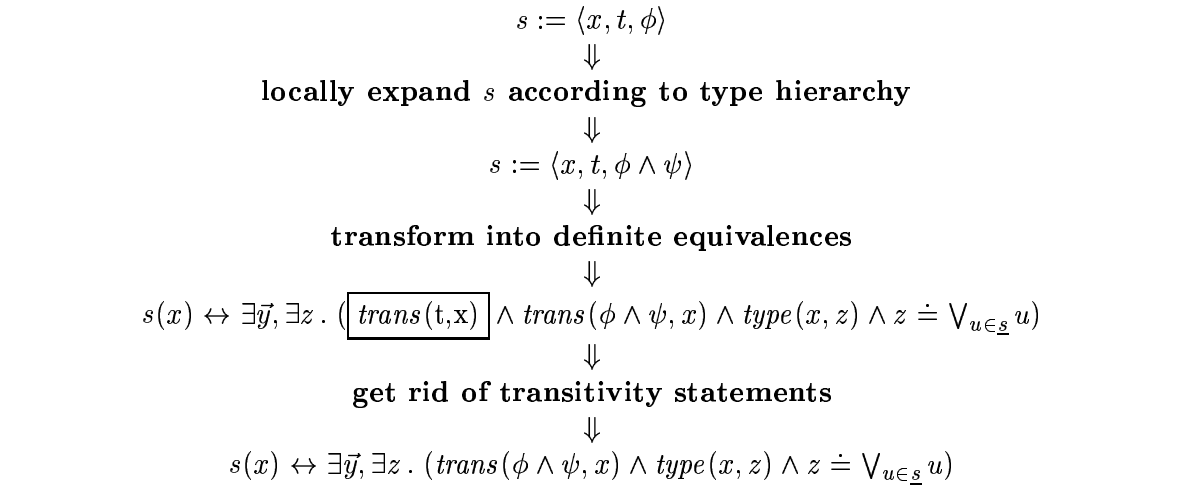

Figure 3: *Transformation schema from nonmonotonically defined types into definite equivalences. ψ denotes the additional constraints of the supertypes of s that have already been inherited (modulo inheritance conflicts) before calling trans. After this, one can get rid of t which might destroy the monotonicity of the resulting definite program (see rectangular box). This is due to the fact that the constraints defined on t might conflict with information from $\phi \wedge \psi$.*

a type name is nothing more than a *shorthand* for abbreviating idiosyncratic and inherited constraints. Given an arbitrary type system, the crucial point here is that the construction of the "locally expanded type system", i.e., the type system where every type has inherited the complete information of all its supertypes modulo inheritance conflict strategies makes the inheritance hierarchy superfluous.
Exactly the transitivity property (32) of the type subsumption relation and its interpretation as denoting the subset relationship (33) leads to conflicts, since at run time conflicting constraints might come into play through supertypes:

$$\forall s, t, u \in \mathcal{T}\,.\ \textbf{if } s \preceq t \textbf{ and } t \preceq u \textbf{ then } s \preceq u \tag{32}$$

$$\forall s, t \in \mathcal{T}\,.\ \textbf{if } s \preceq t \textbf{ then } s^{\mathcal{I}} \subseteq t^{\mathcal{I}}, \text{ for every interpretation } \mathcal{I} \tag{33}$$

Hence, having inherited these constraints, we can get rid of the supertypes (since they contain no further information) and get a perfectly monotonic definite program. This idea leads us to the translation schema of Fig. 3.

As we have said before, we will not give any hints how to resolve a specific inheritance conflict. The primary goal here is instead on the construction of a monotonic definite program *after* such conflicts have been resolved. Consider, for instance, the following example to see the outcome of the modified transformation schema. The type definitions are given in the concrete syntax of $\mathcal{TDL}$ (Krieger and Schäfer 1994).

```
boolean := '+ | '-.
u := [A:boolean '+, B w].
v != u & [A '-].
w := v & [C u].
```
(34)

We hasten to add that $\mathcal{TDL}$ has implemented a restricted kind of overwriting, called "single link overwriting" (SLO) that allows a grammarian to overwrite features in a type definition iff the type under definition inherits from a *single* supertype (see Krieger 1995a). Furthermore, overwriting can be restricted to certain alternatives, as is the case for variables in imperative programming languages, such as PASCAL.[12] In the above example, the type `v` has been defined nonmonotonically (indicated through `!=`) by inheriting from `u`. `v` overwrites feature `A` with the atom `-` which is legal, since `A` has been restricted in `u` to take only `boolean` values. Now, by inheriting the constraints of the supertypes, we get the following typed feature structures for `u`, `v`, and `w`:

$$\begin{bmatrix} u & \\ \mathtt{A} & + \\ \mathtt{B} & w \end{bmatrix} \qquad \begin{bmatrix} v & \\ \mathtt{A} & - \\ \mathtt{B} & w \end{bmatrix} \qquad \begin{bmatrix} w & \\ \mathtt{A} & - \\ \mathtt{B} & w \\ \mathtt{C} & u \end{bmatrix} \tag{35}$$

Applying the modified transformation schema of Fig. 3, we obtain the following definite program P:

$$\begin{aligned} P = \{ u(x) \leftrightarrow\ & \exists y_1, \exists y_2, \exists z \,.\, \boxed{\top(x)} \wedge \\ & \mathtt{A}(x, y_1) \wedge (y_1 \doteq +) \wedge \mathtt{B}(x, y_2) \wedge w(y_2) \wedge \\ & \mathit{type}(x, z) \wedge z \doteq (u \vee v \vee w), \\ v(x) \leftrightarrow\ & \exists y_1, \exists y_2, \exists z \,.\, \boxed{u(x)} \wedge \\ & \mathtt{A}(x, y_1) \wedge (y_1 \doteq -) \wedge \mathtt{B}(x, y_2) \wedge w(y_2) \wedge \\ & \mathit{type}(x, z) \wedge z \doteq (v \vee w), \\ w(x) \leftrightarrow\ & \exists y_1, \exists y_2, \exists y_3, \exists z \,.\, \boxed{v(x)} \wedge \\ & \mathtt{A}(x, y_1) \wedge (y_1 \doteq -) \wedge \mathtt{B}(x, y_2) \wedge w(y_2) \wedge \\ & \mathtt{C}(x, y_3) \wedge u(y_3) \wedge \mathit{type}(x, z) \wedge z \doteq w \} \end{aligned} \tag{36}$$

Recall that the types in the rectangular boxes must be thrown away to guarantee that, e.g., v will have a non-empty model. Otherwise, the reduction of $\boxed{u(x)}$ through its right-hand side would lead to a description for v which can not be satisfied by any interpretation (viz., $y_1 \doteq +$ **and** $y_1 \doteq -$).

7 Other Approaches

Contrary to most other related treatments which come up with a least fixpoint semantics, e.g., LOGIN/KBL (Aït-Kaci 1986), TFS (Emele and Zajac 1991), CUF (Dörre and Dorna 1993), ALE

[12] It is worth noting here that this kind of overwriting can be implemented very efficiently, since the single-supertype property leads to a *tree*-like structure of the type hierarchy if "subsumption-preserving partitions" are imploded into nodes. This view on the type hierarchy allows for efficient indexing. Speaking from an algebraic point of view, we note further that SLO can be characterized in terms of ideals and filters; see Krieger 1995a.

(Carpenter and Penn 1994), or PROLOG (Lloyd 1987), we have decided to choose a *greatest model semantics* for typed feature structures.[13] Our choice was based on a theoretical and a practical observation:

1. The GFP provides us with the maximal denotation of a given description (it models exactly the "linguistic" constraints). It captures certain "incomplete" situations (undefined types, recursive types not grounded on atoms) which are ruled out under a LFP interpretation. Within the rational tree domain, the GFP even handles cyclic descriptions properly.

2. Type expansion (or sort unfolding) is a way of checking satisfiability of typed feature structures (Krieger and Schäfer 1995, Aït-Kaci et al. 1993). This is achieved by adding more and more information to the input structure, substituting types through their improved definition, and repeating this process until a failure occurs. But in fact, this strategy approximates the construction of the greatest fixpoint, namely, to start with $\top$, as opposed to $\bot$ in case of the least fixpoint (see definition of upward/downward ordinal power of T_P). This idea is very intuitive and has been realized in $\mathcal{TDL}$ (Krieger and Schäfer 1994). We guess that many systems who have proposed the least fixpoint, e.g., TFS, have nevertheless implemented the greatest one.

To the best of our knowledge, only two other more linguistically-oriented approaches have also established a greatest fixpoint interpretation in our sense, viz., Pollard and Moshier 1990 and Rounds 1988. Both Pollard&Moshier and Rounds are interested in formalizing set values. Although Pollard&Moshier have started with $\bot$, their information ordering is dual to ours, thus $\bot$ denotes the whole universe (= no information) and $\top$ corresponds to overspecification. Pollard&Moshier's function T operates on a Heyting lattice (or pseudo-Boolean lattice), the dual to a Brouwerian lattice which is used in the least fixpoint interpretation of KBL (Aït-Kaci 1986). Contrary to Pollard&Moshier and Rounds, Aït-Kaci does not make a strict distinction between syntax and semantics in the KBL language. Pollard&Moshier and Rounds instead use an extension of Kasper-Rounds KR logic, where KR formulae denote directed acyclic graphs.
However both approaches rely on very specific powerdomain constructions and the ordering relation among elements is somewhat complicated (Pollard&Moshier: Smyth order; Rounds: Hoare order). In contrast to this, our interpretation domain is given by the powerset of the set of rational feature trees which is much easier and more natural. Elements are thus ordered by subset inclusion and meet/join simply corresponds to intersection/union of sets.
We finally mention that Baader 1990 and Nebel 1991 argue that terminological cycles in KL-ONE-like languages should be treated by a greatest fixpoint semantics. We understand their work as a hint that our choice is correct, since terminological cycles are the analogue to recursive type definitions in typed feature formalisms.

Let us have a few words on the operational realization of our approach. Because the outcome of our compilation schema are definite descriptions over a certain constraint language, it is clear that instances of CLP languages are the right candidates for executing definite programs.

[13] Interestingly to note, Lloyd 1987 argues in Chapter 6 on "Perpetual Processes" that a greatest fixpoint semantics is a more appropriate setting for the foundations of logic programming than the standard least fixpoint interpretation.

To see whether the transformation approach makes sense form the viewpoint of efficiency, it is planned to employ Oz (Henz et al. 1993, Smolka et al. 1993) as a testbed. Oz realizes definite equivalences through the **proc** construct and employs sophisticated delaying mechanisms, different search strategies, and finite domains. We think that these ingredients are a good starting point for efficient processing and for guaranteeing termination. Let us give an example to see how easy it is to encode a definite equivalence as a proc(edure). We choose the definition of u in the definite program (36) and get

```
proc {U x}
 local y1, y2, z in
  {Top x}
  {A x y1}
  y1 = +
  {B x y2}
  {W y2}
  {Type x z}
  or z = u [ ]
     z = v [ ]
     z = w
  ro
 end
end
```
(37)

Whether such code is efficiently executable needs further investigation.

8 Summary and Conclusions

In this paper, we have argued that definite equivalences are the right basis for formalizing arbitrary type systems, thus putting typed feature structures on the solid ground of first-order predicate logic. This means that we have strictly distinguished between syntax (definite equivalences) and semantics (typed feature trees) of our first-order language. We have presented a fully automated method which transforms type systems into definite programs, whereas the meaning of the definite program gives the intended denotation of the type system. Operationally, such programs can then be executed by implementations of CLP languages.

As known from logic programming, the meaning of a definite program P here is given in terms of a certain fixpoint over a continuous function T_P, operating on the powerset of our domain (rational feature trees). This has several advantages when compared with other approaches: (i) we must not rely on specific powerdomain constructions as has been proposed in the literature and (ii) subsumption/entailment is defined in terms of set membership and not through specialized ordering relations.

Contrary to most approaches, we have decided to choose a greatest model semantics as the intended interpretation of P, or equivalently, as the denotation of the type system. Our choice was based on both theoretical and practical reasons, as described in this paper. We have further shown that the greatest fixpoint of T_P can even be reached in at most ω-many steps which might have an impact on the operational semantics and the non-satisfiability question of typed feature structures/definite descriptions.

Our transformation approach is also important when formalizing nonmonotonically defined types, since under our interpretation, we can view the type hierarchy as a pure transport medium which can be thrown away after an additional compilation phase, so that we ultimately obtain a perfectly monotonic definite program.

Several questions remain open and certain lines of research might be worthwhile to follow. We will only mention three of them.

In Section 3, we have restricted negation within definite formulae to the constraint level, i.e., to negated equations of the form $s \not\doteq t$, where $s, t \in \mathcal{V} \cup \mathcal{A}$. The essential reason for ensuring such a strong condition is simple—if we would allow unrestricted (classical) negation, the function T_P obviously is no longer monotonic, and thus we are not allowed to apply our well-known fixpoint theorems (we lost what Rounds and Kasper 1986 call the "upward-closure" property of $\models$). However, non-recursive relations within the scope of a negation symbol together with finite domains do not pose severe problems, since in this case we can push down negation to the constraint level and afterwards substituting possibly existing universal quantifiers through a finite collection of alternatives.

Clearly, properly handling general negation in this theoretical framework is unclear.[14] Work in logic programming, especially by Fitting and Kunen, indicates that by modifying the definition of T_P (and thus the satisfaction relation $\models$), it possible to retain monotonicity. However, such approaches come up with a three-valued (e.g., Fitting 1985, Kunen 1987), four-valued (Fitting 1991, Fitting 1993) or even higher multi-valued logics. Using an interpretation ordering based on the degree of knowledge (instead of degree of truth), a different T_P (called Φ_P) can be obtained which will be monotonic w.r.t. this ordering, whether negations are present or not. Recently, Fitting 1994 has applied Banach's contraction theorem to logic programming, replacing the Knaster-Tarski fixpoint theorem. The nice thing here is that the process of finding a monotonic function $f : S \longmapsto S$ is replaced by making f a contractive mapping over a metric space $\langle S, d \rangle$. I.e., if there is a number $0 < k < 1$ such that for all $s, t \in \Re$, $d(f(s), f(t)) \leq k \cdot d(s, t)$, then f will have a unique fixpoint and the sequence $\langle f^n(s) \rangle_{n \in \omega}$ converges to this fixpoint, for any $s \in S$.

At the end of Section 5, we have shown that the greatest fixpoint of T_P can be reached in at most ω-many steps. This result is important for two practical reasons (and not only a nice theoretical discovery).

First of all, although the satisfiability question is undecidable, we believe that Proposition 8 is a precondition for having a proper operational semantics under this special interpretation of definite programs, or equivalently, of type systems. Speaking from the viewpoint of type systems, at most ω-many steps guarantee that for a given typed feature structure, the type expansion process is a "true" approximation of the final structure (and not remains constant), since at infinity ($= \omega$) we are able to say whether a typed feature structure/definite description is satisfiable or not. This result makes sure that we come arbitrarily close to the final result as processing time increases.

Second, we conjecture in the presence of Proposition 8 that non-satisfiability of typed feature

[14] Despite this fact, general negation over recursive type definitions has been implemented in $\mathcal{TDL}$, due to the following observation: when checking for the satisfiability of a given typed feature structure containing negation, the sequence of approximations originating from the type expansion process converges, although it is not monotonic—however, for a finite type system, we can obtain a finite number of subsequences of this sequence, such that each subsequence is monotonic. The same is true for the "famous" real-valued sequence $\left\langle (-1)^k \cdot \frac{1}{k} \right\rangle_{k \geq 1}$.

structures (definite descriptions) is semi-decidable. The intuitive argument is as follows. Given an arbitrary recursive typed feature structure and assuming a fair type unfolding strategy, the only event under which the fixpoint approximation terminates in finite time follows from a local failure under an initial path which then leads to a global failure. In every other case, the unfolding process proceeds by substituting types through their improved definitions, and hence this iteration might go on infinitely long. Thus the only thing we can expect is that testing for non-satisfiability is semi-decidable, i.e., if our structure is not satisfiable, we get a positive answer in finite time. Notice that this supposition has formally been shown only for conjunctive descriptions (Aït-Kaci et al. 1993). By adding disjunctions, a formal proof is still missing.

The transformation schema presented in this paper is a way to assign a meaning to a type system/typed feature structure without setting up a "designer" logic. However, this approach does not tell us anything about the efficiency (or even the termination) of the corresponding definite program when executed in a CLP programming language. As is often the case, specialized inference engines have a better run-time performance than general-purpose reasoners; thus grammars might be processed more efficient in implementations of specialized typed feature logics than in general logic programming languages. This question is under evaluation in the near future.

References

Aït-Kaci, H. 1986. An Algebraic Semantics Approach to the Effective Resolution of Type Equations. *Theoretical Computer Science* 45:293–351.

Aït-Kaci, H., A. Podelski, and S. C. Goldstein. 1993. Order-Sorted Feature Theory Unification. Technical Report 32, Digital Equipment Corporation, DEC Paris Research Laboratory, France, May. Also in Proceedings of the International Symposium on Logic Programming, Oct. 1993, MIT Press.

Aït-Kaci, H., A. Podelski, and G. Smolka. 1992. A Feature-Based Constraint System for Logic Programming with Entailment. Technical Report RR-92-17, German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany, March. Also in Proceedings of the International Conference on Fifth Generation Computer Systems 1992.

Baader, F. 1990. Terminological Cycles in KL-ONE-based Knowledge Representation Languages. Technical Report RR-90-01, German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany.

Backofen, R. 1994. *Expressivity and Decidability of First-Order Languages over Feature Trees.* PhD thesis, Universität des Saarlandes, Department of Computer Science, December.

Bouma, G. 1992. Feature Structures and Nonmonotonicity. *Computational Linguistics* 18(2):183–203.

Carpenter, B. 1992. *The Logic of Typed Feature Structures.* Tracts in Theoretical Computer Science. Cambridge: Cambridge University Press.

Carpenter, B. 1993. Skeptical and Credulous Default Unification with Applications to Templates and Inheritance. In *Inheritance, Defaults, and the Lexicon*, ed. T. Briscoe, A. Copestake, and V. de Paiva. Cambridge University Press.

Carpenter, B., and G. Penn. 1994. 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.

Clark, K. 1978. Negation as Failure. In *Logic and Data Bases*, ed. H. Gallaire and J. Minker, 293–322. New York: Plenum Press.

Colmerauer, A. 1982. PROLOG and Infinite Trees. In *Logic Programming*, ed. K. Clark and S.-A. Tärnlund, 231–251. London: Academic Press. A.P.I.C. Sudies in Data Processing, No. 16.

Copestake, A. 1993. Defaults in Lexical Representation. In *Inheritance, Defaults, and the Lexicon*, ed. T. Briscoe, A. Copestake, and V. de Paiva. Cambridge University Press.

Daelemans, W., K. D. Smedt, and G. Gazdar. 1992. Inheritance in Natural Language Processing. *Computational Linguistics* 18(2):205–218.

Davey, B., and H. Priestley. 1990. *Introduction to Lattices and Order.* Cambridge University Press.

Dörre, J., and M. Dorna. 1993. CUF—A Formalism for Linguistic Knowledge Representation. In *Computational Aspects of Constraint-Based Linguistic Description I*, ed. J. Dörre. DYANA.

Emele, M., and R. Zajac. 1991. A Fixed-Point Semantics for Feature Type Systems. In *Proceedings of the 2nd International CTRS Workshop*, ed. S. Kaplan and M. Okada, 383–388. Berlin. Springer.

Euler, L. 1989. Integration von Funktionen, Relationen und Typen beim Sprachentwurf. Teil I: Konzeption, Typhierarchie und Funktionen. Master's thesis, University of Erlangen-Nürnberg. In German.

Evans, R., and G. Gazdar. 1990. The DATR Papers. Technical Report SCRP 139, School of Cognitive and Computing Sciences, University of Sussex, Brighton BN1 9QH, UK.

Fitting, M. C. 1985. A Kripke/Kleene Semantics for Logic Programs. *Journal of Logic Programming* 2:295–312.

Fitting, M. C. 1991. Bilattices and the Semantics of Logic Programming. *Journal of Logic Programming* 11:91–116.

Fitting, M. C. 1993. The Family of Stable Models. *Journal of Logic Programming* 17:197–225.

Fitting, M. C. 1994. Metric Methods: Three Examples and a Theorem. *Journal of Logic Programming* 21:113–127.

Flickinger, D. 1987. *Lexical Rules in the Hierarchical Lexicon.* PhD thesis, Stanford University.

Grätzer, G. 1978. *General Lattice Theory.* Vol. 52 of Mathematische Reihe. Basel: Birkhäuser.

Henz, M., G. Smolka, and J. Würtz. 1993. Oz—A Programming Language for Multi-Agent Systems. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence IJCAI-93, Vol. 1*, 404–409.

Höhfeld, M., and G. Smolka. 1988. Definite Relations over Constraint Languages. LILOG Report 53, WT LILOG–IBM Germany, Stuttgart, October.

Jaffar, J., and J.-L. Lassez. 1987. Constraint Logic Programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, 111–119.

Johnson, M. 1988. *Attribute Value Logic and the Theory of Grammar.* CSLI Lecture Notes, Number 16. Stanford: Center for the Study of Language and Information.

Johnsonbaugh, R., and W. Pfaffenberger. 1981. *Foundations of Mathematical Analysis.* Pure and Applied Mathematics, Number 62. New York: Marcel Dekker.

Krieger, H.-U. 1995a. *TDL—A Type Description Language for Constraint-Based Grammars. Foundations, Implementation, and Applications.* PhD thesis, Universität des Saarlandes, Department of Computer Science, September.

Krieger, H.-U. 1995b. Classification and Representation of Types in *TDL*. In *Proceedings of the International KRUSE Symposium, Knowledge Retrieval, Use, and Storage for Efficiency*, 74–85. Also available as DFKI Research Report RR-95-17.

Krieger, H.-U., and U. Schäfer. 1994. *TDL*—A Type Description Language for Constraint-Based Grammars. In *Proceedings of the 15th International Conference on Computational Linguistics, COLING-94*, 893–899. An enlarged version of this paper is available as DFKI Research Report RR-94-37.

Krieger, H.-U., and U. Schäfer. 1995. Efficient Parameterizable Type Expansion for Typed Feature Formalisms. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence, IJCAI-95*, 1428–1434. Also available as DFKI Research Report RR-95-18.

Kunen, K. 1987. Negation in Logic Programming. *Journal of Logic Programming* 4:289–308.

Lloyd, J. 1987. *Foundations of Logic Programming.* Springer. 2nd edition.

Nebel, B. 1991. Terminological Cycles: Semantics and Computational Properties. In *Principles of Semantic Networks: Explorations in the Representation of Knowledge*, ed. J. F. Sowa, 331–361. San Mateo, CA: Morgan Kaufmann.

Pereira, F. C., and S. M. Shieber. 1984. The Semantics of Grammar Formalisms Seen as Computer Languages. In *Proceedings of the 10th International Conference on Computational Linguistics*, 123–129.

Pollard, C., and I. A. Sag. 1987. *Information-Based Syntax and Semantics. Vol. I: Fundamentals.* CSLI Lecture Notes, Number 13. Stanford: Center for the Study of Language and Information.

Pollard, C., and I. A. Sag. 1994. *Head-Driven Phrase Structure Grammar.* Studies in Contemporary Linguistics. Chicago: University of Chicago Press.

Pollard, C. J., and M. D. Moshier. 1990. Unifying Partial Descriptions of Sets. In *Information, Language, and Cognition. Vol. 1 of Vancouver Studies in Cognitive Science*, ed. P. Hanson, 285–322. University of British Columbia Press.

Rounds, W. C. 1988. Set Values for Unification-Based Grammar Formalisms and Logic Programming. Technical Report CSLI-88-129, Center for the Study of Language and Information.

Rounds, W. C., and R. T. Kasper. 1986. A Complete Logical Calculus for Record Structures Representing Linguistic Information. In *Proceedings of the 15th Annual Symposium of the IEEE on Logic in Computer Science.*

Russell, G., A. Ballim, J. Carroll, and S. Warwick-Armstrong. 1992. A Practical Approach to Multiple Default Inheritance for Unification-Based Lexicons. *Computational Linguistics* 18(3):311–337.

Smolka, G. 1991. Residuation and Guarded Rules for Constraint-Logic Programming. Research Report RR-91-13, German Research Center for Artificial Intelligence (DFKI), Saarbrücken.

Smolka, G. 1993. Logische Programmierung. Manuscript of a lecture on logic programming, held at the University of Saarbrücken (in German).

Smolka, G., M. Henz, and J. Würtz. 1993. Object-Oriented Concurrent Constraint Programming in Oz. Research Report RR-93-16, German Research Center for Artificial Intelligence (DFKI), Saarbrücken.

Treinen, R. 1993. Feature Constraints with First-Class Features. In *Mathematical Foundations of Computer Science, MFCS-93*, 734–743. Berlin. Springer. Lecture Notes in Computer Science, Vol. 711.

Young, M. A., and B. Rounds. 1993. A Logical Semantics for Nonmonotonic Sorts. In *Proceedings of the 31st Annual Meeting of the Association for Computational Linguistics, ACL-93*, 209–215.

Zajac, R. 1992. Inheritance and Constraint-Based Grammar Formalisms. *Computational Linguistics* 18(2):159–182.