The TIGER language.

A Description Language for Syntax Graphs.

Part 2: Formal Definition

Esther König and Wolfgang Lezius

IMS, University of Stuttgart, Germany
www.ims.uni-stuttgart.de/projekte/TIGER

November 13, 2001

Contents

1 Introduction **3**

1.1 Acknowledgements and References 3

2 Syntax Graph Model **4**

2.1 Feature Records 4

2.2 Nodes 4

2.3 Syntax Graphs 5

2.4 Syntax Graph Relations 7

2.5 Syntax Graph Models 7

3 Syntax and Interpretation of the TIGER Language **8**

3.1 Elementary Symbols 8

3.2 Feature Values 10

3.3 Feature Constraints 10

3.4 Node Descriptions 11

3.5 Graph Descriptions 11

3.6 Satisfaction of local constraints 14

3.7 Corpus Definition 14

3.7.1 Feature Values 14

3.7.2 Feature Constraints 14

3.7.3 Node Descriptions 15

3.7.4 Graph Descriptions 15

3.7.5 Denotation of a corpus definition . . . 15
3.8 Queries . . . 16
3.9 Types . . . 16
3.10 Template Definitions . . . 16
3.11 Semantic Consequence . . . 17

4 TIGER calculus **18**
4.1 Feature Values . . . 19
4.1.1 Disjunctive Normal Form: . . . 19
4.1.2 Disjunction reduction: . . . 20
4.1.3 Consistency check: . . . 20
4.2 Feature Constraints . . . 22
4.2.1 Disjunctive Normal Form: . . . 22
4.2.2 Disjunction reduction: . . . 23
4.2.3 Feature normalization: . . . 23
4.2.4 Consistency check: . . . 23
4.3 Node Descriptions . . . 25
4.3.1 Disjunctive Normal Form: . . . 26
4.3.2 Node description normalization: . . . 26
4.3.3 Node guess: . . . 26
4.4 Graph Descriptions . . . 27
4.4.1 Disjunctive Normal Form: . . . 27
4.4.2 Disjunction reduction: . . . 27
4.4.3 Graph description normalization: . . . 27
4.4.4 Consistency check: . . . 28

Chapter 1

Introduction

This paper gives a formal definition of a decription language for *syntax graphs*, called the TIGER language. For an informal introduction to the TIGER language please refer to [2]. Syntax graphs are close relatives to (syntax) trees, to feature structures or dependency graphs. Syntax graphs are syntax trees with two additions: Edges may carry labels, and crossing edges are permitted. On the other hand, syntax graphs differ from feature structures due to the following two properties: The terminal nodes are ordered (like in syntax trees). Albeit edges may cross, two edges must not join in a common node. This means that the structure sharing mechanism of feature structures is ruled out (at least in the kernel of the TIGER language).

Chapt. 2 defines the syntax graph model. The syntax of the TIGER language and its interpretation with respect to syntax graphs is given in Chapt. 3. A calculus for formula reduction is introduced in Chapt. 4, which serves as the basis for the implementation of the TIGER language.

1.1 Acknowledgements and References

The design and the formal definition of the TIGER language has been heavily influenced by the work cited in [2].

Chapter 2

Syntax Graph Model

One component of a *syntax graph model* is the set of *syntax graphs*. A (syntax) graph G is a relation over a set of *nodes* V. Whereas, from the point of view of the graph, a node is an atomic element, a node can still have its own life, i.e. it can possess internal structure. For example, in the case of TIGER syntax graphs, a node is a pair of a *node identifier* and a *feature record*.

2.1 Feature Records

According to common practice in computational linguistics, *features* are functions. In our case, their range is restricted to constants. For this reason, we prefer the term *feature record* instead of *feature structure* (where values of features are again arbitrary feature structures).

Definition 1 (Feature Record)

A feature record F *is a set of* feature maps $F = \{f_1, \ldots, f_m\}$ *where* $f_i : \{d \in D\} \rightarrow C_i$ *and* C_i *is a set of constants.* D *is a non-empty set, the domain of feature records* $\mathcal{F}$.

2.2 Nodes

Definition 2 (Node)

A node $v \in V$ *is a pair* $v = \langle \nu, F \rangle$ *of a* node identifier ν *and a feature record* F.

This means that a node is unique due to its identifier.

2.3 Syntax Graphs

The following definition specifies the properties of syntax graphs $G \in \mathcal{G}$, where $\mathcal{G}$ is the set of syntax graphs.

Definition 3 (Syntax Graph)

A syntax graph *G is a sextuple $G = (V_{NT}, V_T, L_G, E_G, O_G, R_G)$ where*

1. *V_{NT} is a possibly empty set of* nonterminal *or* inner nodes.
2. *V_T is a non-empty set of* terminal nodes *or* leaves.

   Notation: $V_G = V_{NT} \cup V_T$ denotes the set of all nodes of a graph G.
3. *L_G is a (possibly empty) set of edge labels.*
4. *E_G are the labeled, directed edges of G. E_G is a set of relations l, $l \in L_G$, with $\langle V_{NT}, (V_{NT} \cup V_T)\rangle$,*
5. *O_G is a bijective order transformation $O_G : V_T \to \{1, \ldots, |V_T|\}$*

   I.e. for any pair of terminal nodes $v_i, v_j \in V_T$, $v_i \neq v_j$, it can be decided whether $O_G(v_i) < O_G(v_j)$ or $O_G(v_i) > O_G(v_j)$.
6. *$R_G \in (V_{NT} \cup V_T)$ is the* root node.

G is a graph with the following restrictions:

1. *G is a directed acyclic graph with exactly one root node R_G. (A root node is a node without incoming edges.)*
2. *For all nodes $v \in V_G$ different from the root node R_G, there is exactly one edge leading into v.*
3. *Each nonterminal node $v \in V_{NT}$ must have at least one successor, i.e.*

   $v \in V_{NT}$ *iff* $\exists\, l, v' \quad \langle v, v'\rangle \in l$.

Note that a single node $v \in V_T$ is trivially a syntax graph.

Similar to nodes, there must be a means to distinguish graphs. Since the identifier of a node is unique in the whole domain of nodes, we can simply take the identifier of the root node as the *identifier of a graph*. Furthermore, it follows that each node v_i belongs to a unique graph G_i.

There are few convenient definitions wrt. a syntax graph G.

Definition 4 (Path)

A path $p = \langle v_0, \ldots, v_n \rangle$ *in a syntax graph* G *is an* n*-tuple of nodes with* $\langle v_0, v_1 \rangle \in l$*, for some* l*, and either* $v_1 = v_n$ *or* $p' = \langle v_1, \ldots, v_n \rangle$ *is a path,* $0 < n$*. The number* n *is called the* length *of the path* p*.*

Definition 5 ($>n$, dominance with distance n, $n \geq 0$)

A node v_1 dominates *a node* v_2 with distance n *(*$v_1 > n\ v_2$*) if there is a path* $p = \langle v_1, \ldots, v_n \rangle$ *in* G*.*

Definition 6 ($> @l$, left corner)

A node $v_2 \in V_T$ *is the* left corner *with respect to (the subgraph dominated by) a node* v_1 *(*$v_1 > @l\ v_2$*) if*

- *either* $v_2 = v_1$
- *or* $v_1 > n\ v_2$
  and for all nodes $v_i \in V_T$ *with* $v_1 > *\ v_i$ *it holds that* $O_G(v_i) > O_G(v_2)$*.*

The *right corner* v_2 of a node v_1 ($v_1 > @r\ v_2$) is defined analogously.

The (generalized) precedence of two nodes v_1 and v_2 is defined with respect to the respective leftmost terminal nodes ('left corners') dominated by v_1 and v_2. Since the precedence of terminal nodes is a total order, the order relation between two left corners is always defined.

Definition 7 ($.n$, precedence with distance n, $n \geq 0$)

A node v_1 precedes *a node* v_2 with distance n *(*$v_1\ .n\ v_2$*) if* $n \geq 0$ *and for the left corner* v_3 *of* v_1 *(*$v_1 > @l\ v_3$*) with respect to the left corner* v_4 *of* v_2 *(*$v_2 > @l\ v_4$*) it holds that* $n = O_G(v_4) - O_G(v_3)$*.*

Definition 8 ($\$$, siblings)

A node v_1 *is a* sibling *of a node* v_2 *(*$v_1\$\ v_2$*) if there exists a node* v_0 *which directly dominates both,* v_1 *and* v_2*, i.e.* $\exists\, v_0 \in V_{NT} :\ v_0 > v_1\ \wedge\ v_0 > v_2\ \wedge\ v_1 \neq v_2$*.*

Definition 9 (Arity n of a node with identifier ν)

The arity n *of a node* v *with identifier* ν *is the number* n *of its daughters or outgoing edges, i.e.*

$arity(\nu) = |\ \{l | \langle v, v_l \rangle \in l\ \}\ |.$

Definition 10 (Token arity n of a node with identifier ν)

The token arity n of a node v with identifier ν is the number n of terminal nodes which are dominated by v, i.e.

$tokenarity(\nu) = |\,\{l|\langle v, \ldots, v_l\rangle$ *path in* G *and* $v_l \in V_T\ \}\,|$.

Definition 11 (Continuous graph rooted in v)

The graph which is rooted in a node v with identifier ν is continuous *if its terminal nodes form a continuous string, i.e. for all terminal leaves $v_i = v_1, \ldots, v_n$ dominated by v it holds that*

$(\exists v_j\, O_G(v_j) = O_G(v_i) + 1)$ *or* $(\neg\exists v_j\, O_G(v_j) > O_G(v_i))$

2.4 Syntax Graph Relations

In order to model types and templates, we need a notion of *relation* among constants, feature records, nodes, and syntax graphs. Types correspond to unary relations or predicates. Let $\mathcal{U}$ be the universe of graphs and all their substructures, i.e. the (disjoint) set union of constants C, feature records $\mathcal{F}$, nodes V, and syntax graphs $\mathcal{G}$.

Definition 12 (Syntax Graph Relation)

An n-ary (syntax graph) relation $r \in R$ is an n-tuple $(n > 0)$ $r \subseteq \mathcal{U} \times \ldots \times \mathcal{U}$, where the individual subsets of $\mathcal{U}$ must not be empty.

2.5 Syntax Graph Models

The notion of *syntax graph model* ties together all the previously introduced components.

Definition 13 (Syntax Graph Model)

A syntax graph model *is a quintuple* $\mathcal{M} = (C, \mathcal{F}, V, \mathcal{G}, R)$.

Chapter 3

Syntax and Interpretation of the TIGER Language

This section defines the syntax of the TIGER language in a formal manner. Along the way, the semantic interpretation of each TIGER expression will be defined with respect to a given syntax graph model $\mathcal{M}$ (see Sect. 2). There are three levels of interpretation:

1. denotation of the *elementary symbols*
2. denotation of *local constraints*, i.e. of descriptions of feature values, feature constraints, node descriptions, graph descriptions; and the corresponding notion of constraint satisfaction
3. interpretation/satisfaction of *global definitions and constraints*, i.e. corpus definitions, template definitions, type definitions, and feature declarations.

3.1 Elementary Symbols

The inventory of elementary symbols is defined as follows:

Definition 14 (Signature)

Operators

`=` *(feature value)*,

`!` *(negation)*, `&` *(conjunction)*, `|` *(disjunction)*,

`>` *(direct dominance)*, `>*` *(dominance)*, `>@l` *(left corner)*, `>@r` *(right corner)*,

`.` *(direct precedence)*, `.*` *(precedence)*,

`$` *(siblings)*, `$.*` *(ordered siblings)*

`:=` *(type definition)*

Mark-up

$\texttt{;}$, $\texttt{<}$, $\texttt{>}$, $\texttt{/}$, $\texttt{domain}$, $\texttt{feature}$, $\texttt{name}$, $\texttt{range}$

Constants $\texttt{c1}, \texttt{c2}, \ldots \in \mathbf{C}$

Feature Names $\texttt{f1}, \texttt{f2}, \ldots \in \mathbf{F}$

Edge Labels $\texttt{l1}, \texttt{l2}, \ldots \in \mathbf{L}$

Type Names $\texttt{t1}, \texttt{t2}, \ldots, \texttt{tn}, \texttt{Constant}, \texttt{FREC}, \texttt{Graph}, \texttt{Node}, \texttt{NT}, \texttt{T}, \top$ *(top)* $\in \mathbf{T}$, *and* $\perp$ *(bottom).*

Predicate Names $\texttt{root}$, $\texttt{arity}$, $\texttt{tokenarity}$, $\texttt{continuous}$, $\texttt{discontinuous}$

Template Names $\texttt{r1}, \texttt{r2}, \ldots \in \mathbf{R}$

Constants have to be put into quotes `"..."`. Type names must neither start with `"` nor with the `#`-symbol (in order to avoid confusion with variables, see below). Therefore, the sets **C** and **T** are disjoint.

We constrain the notion of syntax graph model (Sect. 2) by the condition that each element of the signature must have a semantic counterpart in the given syntax graph model $\mathcal{M}$. In order to improve readability, we omit indices, i.e. $c \in C$ means $c_{\mathcal{M}} \in C_{\mathcal{M}}$, etc.

Definition 15 (Denotation of elementary symbols)

Constants: *there is a constant* $c \in C$ *for each* $\texttt{c} \in \texttt{C}$

Feature Names: *there is unary partial function* f *on* D *for each* $\texttt{f} \in \texttt{F}$

Edge Labels: *there is an edge label* $l \in L$ *for each* $\texttt{l} \in \texttt{L}$

Type Names: *there is a unary syntax graph relation* $r_{\texttt{t}} \in R$ *for each* $\texttt{t} \in \texttt{T}$.

In particular, $r_{\texttt{Constant}} = C$, $r_{\texttt{FREC}} = \mathcal{F}$, $r_{\texttt{Node}} = \mathcal{V}$, $r_{\texttt{Graph}} = \mathcal{G}$, $r_{\top} = \mathcal{U}$.

In addition, $[\![\perp]\!] = \emptyset$.

Template Names: *there is an* n*-ary syntax graph relation* $r \in R$ *for each* n*-ary template* $\texttt{r} \in \texttt{R}$

Furthermore, we introduce a general notion of variables. This means that the denotation of TIGER expressions has to be calculated with respect to a given variable substitution σ.

Definition 16 (Variable substitution)

Let $\mathbf{X}$ *be a set of* variable names *(*variables*) with members* $\texttt{\#x1}, \texttt{\#x2}, \ldots \in \mathbf{X}$. *A* variable substitution σ *is a map from variable names on elements of the set* $\mathcal{U}$. *The function value* $\sigma(\texttt{\#x})$ *is the element which is* assigned *to the variable* `#x`.

The sets of variable names for feature values, feature constraints, and node identifiers, respectively, must be pairwise disjoint.

Subsequently, we mean $[\![\gamma]\!]_{\mathcal{M},\sigma}$ ('denotation of γ in model $\mathcal{M}$ under variable substitution σ') when writing $[\![\gamma]\!]$.

3.2 Feature Values

A feature value expression `d` denotes a subset of the set of constants C. A type `t` is interpreted as a one-place relation $r_{\mathtt{t}}$, whose single argument must be a constant.

Syntax	*Explanation*	*Denotation* $[\![\cdot]\!]$
`d` $\Rightarrow$ `c`	constant (string)	$\{ c \}$
$\mid$ `t`	type	$\{ u \mid \langle u \rangle \in r_{\mathtt{t}} \} \cap C$
$\mid$ `! d`	negation	$C \setminus [\![\mathtt{d}]\!]$
$\mid$ `d & d`	conjunction	$[\![\mathtt{d}_1]\!] \cap [\![\mathtt{d}_2]\!]$
$\mid$ `d | d`	disjunction	$[\![\mathtt{d}_1]\!] \cup [\![\mathtt{d}_2]\!]$

Variables may only be used in the place of feature values or as prefixes to feature value expressions, and of course, they must denote constants.

Syntax	*Explanation*	*Denotation* $[\![\cdot]\!]$
`d'` $\Rightarrow$ `d`	feature value	$[\![\mathtt{d}]\!]$
$\mid$ `#x`	variable	$\{ \sigma(\mathtt{\#x}) \} \cap C$
$\mid$ `#x:d`	constrained variable	$\{ \sigma(\mathtt{\#x}) \} \cap [\![\mathtt{d}]\!]$

3.3 Feature Constraints

The syntax and denotation of a *feature constraint* `k` is defined below. In addition to the above mentioned feature value expressions, we also admit regular expressions over strings as feature values. However, a regular expression can neither be refered to by variable nor can it be embedded into a Boolean expression (since this could lead to very large database joins otherwise). We assume that there is a function `evalre` which evaluates a regular expression `z` and returns a set of constants. In the case of feature constraints, types must evaluate to feature records.

Syntax			*Explanation*	*Denotation* $[\![\cdot]\!]$
k	⇒	t	type	$\{ u \mid \langle u\rangle \in r_{\texttt{t}} \} \cap \mathcal{F}$
	|	f = d'	feature-value pair	$\{F \mid f(F) \in [\![\texttt{d'}]\!]\}$
	|	f != d	negated feature-value	$[\![\texttt{f = ! d}]\!]$
	|	f = /z/	regular expression	$\{F \mid f(F) \in \texttt{evalre(z)}\}$
	|	f != /z/	neg. regular expression	$\{F \mid f(F) \notin \texttt{evalre(z)}\}$
	|	! k	negation	$F \setminus [\![\texttt{k}]\!]$
	|	k & k	conjunction	$[\![\texttt{k}_1]\!] \cap [\![\texttt{k}_2]\!]$
	|	k | k	disjunction	$[\![\texttt{k}_1]\!] \cup [\![\texttt{k}_2]\!]$

Similarly to feature values, feature constraints can be prefixed by variables. And finally, a (prefixed) feature constraint comes with square brackets at its outermost level.

Syntax			*Explanation*	*Denotation* $[\![\cdot]\!]$
k'	⇒	[ ]	empty constraint	$\mathcal{F}$
	|	[k]	feature constraint	$[\![\texttt{k}]\!]$
	|	[#x]	variable	$\{ \sigma(\texttt{\#x}) \} \cap \mathcal{F}$
	|	[#x:k]	constrained variable	$\{ \sigma(\texttt{\#x}) \} \cap [\![\texttt{k}]\!]$

3.4 Node Descriptions

A *node description* is a pair of a node identifier `n` and a (prefixed) feature constraint `k'` where either the identifier or the feature constraint can be omitted. The denotation of a node description is actually a single node since the nodes in the syntax graph model come with unique identifiers. However, for reasons of conformity, we let the denotation of a node description be a singleton *set*.

Syntax			*Explanation*	*Denotation* $[\![\cdot]\!]$
n	⇒	c	constant node identifier	$\{ c \}$
	|	#x	variable node identifier	$\{ \sigma(\texttt{\#x}) \} \cap C$
v	⇒	n	unconstrained node	$\{ \langle c, F \rangle \}$ for some $F \in \mathcal{F}$; $c \in [\![\texttt{n}]\!]$
	|	k'	unidentified node	$\{ \langle c, [\![\texttt{k'}]\!] \rangle \}$ for some $c \in C$
	|	n:k'	node description	$\{ \langle c, [\![\texttt{k'}]\!] \rangle \}$, $c \in [\![\texttt{n}]\!]$

3.5 Graph Descriptions

Obviously, a *graph description* should denote a set of graphs G. The details of the definition are given in Fig. 3.5. First, we have to define the syntax of template calls (or template heads).

Syntax			*Explanation*	*Denotation* $[\![\cdot]\!]$
`u`	$\Rightarrow$	`#x`	argument parameter	$\sigma(\texttt{\#x})$
	$\mid$	`#x, u`	arguments	
`r'`	$\Rightarrow$	`r(u)`	template call/head	$\begin{cases} \mathcal{G} & \textit{if } \langle x_1, \ldots, x_n \rangle \in r \\ & \textit{and } \bigwedge_{i=1}^{n} x_i \in \sigma(\texttt{\#x}_i) \\ \emptyset & \textit{otherwise} \end{cases}$

Syntax			*Explanation*	*Denotation* $[\![\cdot]\!]$
g	$\Rightarrow$	`root(n)`	root identification	$\{G \mid [\![\mathtt{n}]\!] = R_G\}$
	$\mid$	`arity(n,`$\mathtt{i}_1$`)`	arity constraint	$\{G \mid arity([\![\mathtt{n}]\!]) = \mathtt{i}_1\}$
	$\mid$	`arity(n,`$\mathtt{i}_1$`,`$\mathtt{i}_2$`)`	arity interval	$\{G \mid \mathtt{i}_1 \leq arity([\![\mathtt{n}]\!]) \leq \mathtt{i}_2\}$
	$\mid$	`tokenarity(n,`$\mathtt{i}_1$`)`	# of terminal leaves	$\{G \mid tokenarity([\![\mathtt{n}]\!]) = \mathtt{i}_1\}$
	$\mid$	`tokenarity(n,`$\mathtt{i}_1$`,`$\mathtt{i}_2$`)`	interval for # of terminals	$\{G \mid \mathtt{i}_1 \leq tokenarity([\![\mathtt{n}]\!]) \leq \mathtt{i}_2\}$
	$\mid$	`continuous(n)`	continuity constraint	$\{G \mid continuous([\![\mathtt{n}]\!])\}$
	$\mid$	`discontinuous(n)`	discontinuity constraint	$\mathcal{G} \setminus \{G \mid continuous([\![\mathtt{n}]\!])\}$
	$\mid$	`r'`	template call	
	$\mid$	`v`	node description	$\{G \mid [\![\mathtt{v}]\!] \subseteq V_G\}$
	$\mid$	`v >l v`	labeled direct dominance	$\{G \mid l(v_1) = v_2\}$
	$\mid$	`v > v`	direct dominance	$\{G \mid v_1 > 1\ v_2\}$
	$\mid$	`v >* v`	dominance	$\{G \mid \exists i\,(v_1 > i\ v_2)\}$
	$\mid$	`v >`$\mathtt{i}_3$` v`	dominance, distance $\mathtt{i}_3$	$\{G \mid v_1 > \mathtt{i}_3\ v_2\}$
	$\mid$	`v >`$\mathtt{i}_3$`,`$\mathtt{i}_4$` v`	dominance, distance $\mathtt{i}_3 \ldots \mathtt{i}_4$	$\{G \mid \exists i\,(\mathtt{i}_3 \leq i \leq \mathtt{i}_4\ \wedge\ v_1 > i\ v_2)\}$
	$\mid$	`v . v`	direct precedence	$\{G \mid v_1\ .1\ v_2\}$
	$\mid$	`v .* v`	precedence	$\{G \mid \exists i\,(v_1\ .i\ v_2)\}$
	$\mid$	`v >@l v`	left corner	$\{G \mid v_1 > @l\ v_2\}$
	$\mid$	`v >@r v`	right corner	$\{G \mid v_1 > @r\ v_2\}$
	$\mid$	`v .`$\mathtt{i}_3$` v`	precedence, distance $\mathtt{i}_3$	$\{G \mid v_1\ .\mathtt{i}_3\ v_2\}$
	$\mid$	`v .`$\mathtt{i}_3$`,`$\mathtt{i}_4$` v`	precedence, distance $\mathtt{i}_3 \ldots \mathtt{i}_4$	$\{G \mid \exists i\,(\mathtt{i}_3 \leq i \leq \mathtt{i}_4 \wedge v_1.i\ v_2)\}$
	$\mid$	`v !R v`	negated node relation	$\mathcal{G} \setminus \{G \mid v_1\,\mathtt{R}\,v_2\}$
	$\mid$	`v $ v`	siblings	$\{G \mid v_1\ \$\ v_2\}$
	$\mid$	`v $.* v`	siblings with precedence	$\{G \mid v_1\ \$\ v_2 \wedge\ \exists i(v_1\ .i\ v_2)\}$
	$\mid$	`g & g`	conjunction	$[\![\mathtt{g}_1]\!] \cap [\![\mathtt{g}_2]\!]$
	$\mid$	`g	g`	disjunction	$[\![\mathtt{g}_1]\!] \cup [\![\mathtt{g}_2]\!]$

Figure 3.1: Syntax and denotation of graph descriptions. $v_1 \in [\![\mathtt{v}_1]\!]$; $v_2 \in [\![\mathtt{v}_2]\!]$; $[\![\mathtt{v}_1]\!]$, $[\![\mathtt{v}_2]\!] \subseteq V_G$; `n` node identifier; $\mathtt{i}_1$, $\mathtt{i}_2 \geq 0$; $\mathtt{i}_3$, $\mathtt{i}_4 \geq 1$; `R` unnegated node relation (`$.*` excluded)

3.6 Satisfaction of local constraints

A local constraint (or description), like a feature constraint or a syntax graph description, denotes a set of structures, and, if ultimatively specific, a singleton set, i.e. one single structure. Satisfaction of a local constraint in a syntax graph model $\mathcal{M}$ is therefore defined with reference to the denotation of the constraint in $\mathcal{M}$.

Definition 17 (Satisfaction, $\mathcal{M}, \sigma \models \gamma$, $\mathcal{M} \models \gamma$)

A constraint γ (γ = `d'`, `k`, `v`, `g`) is satisfied in $\mathcal{M} = (C, \mathcal{F}, V, \mathcal{G}, R)$ under variable substitution σ ($\mathcal{M}, \sigma \models \gamma$) if $[\![\gamma]\!]_{\mathcal{M},\sigma} \neq \emptyset$.

A constraint γ is satisfied in $\mathcal{M}$ ($\mathcal{M} \models \gamma$) if there is some variable substitution σ such that $\mathcal{M}, \sigma \models \gamma$.

Now, we can turn to the global constraints and definitions.

3.7 Corpus Definition

A corpus definition `k` is a non-empty list of restricted graph descriptions or *corpus graphs* $\texttt{g}^r$. The restricted syntax is defined subsequently.

3.7.1 Feature Values

In corpus definitions, feature values $\texttt{d}^r$ must not include types or template calls. Boolean operators are excluded (negation and disjunction) or do not make sense (conjunction of constants).

$$
\begin{array}{lcl}
\texttt{d}^r & \Rightarrow & \texttt{c} \\
\texttt{d'}^r & \Rightarrow & \texttt{d}^r
\end{array}
$$

3.7.2 Feature Constraints

In corpus definitions, feature values $\texttt{d}^r$ cannot be regular expressions. On the level of feature constraints $\texttt{k}^r$, the empty feature constraint, and the Boolean operators `!` and `|` are excluded.

$$
\begin{array}{lcl}
\texttt{k}^r & \Rightarrow & \texttt{f = d'}^r \\
 & | & \texttt{k}^r\ \texttt{\&}\ \texttt{k}^r \\
\texttt{k'}^r & \Rightarrow & \texttt{[}\ \texttt{k}^r\ \texttt{]}
\end{array}
$$

3.7.3 Node Descriptions

A node description $\mathtt{v}^r$ in a corpus definition must be a pair of a constant node identifier and non-empty feature constraint.

$$
\begin{array}{lcl}
\mathtt{n}^r & \Rightarrow & \mathtt{c} \\
\mathtt{v}^r & \Rightarrow & \mathtt{n}^r\mathtt{:k'}^r
\end{array}
$$

3.7.4 Graph Descriptions

In a corpus definition, a graph description $\mathtt{g}^r$ must be conjunctions of basic node relations and fixed arity constraints.

$$
\begin{array}{lcl}
\mathtt{g}^r & \Rightarrow & \mathtt{arity(n,a)} \\
 & | & \mathtt{v}^r\ \mathtt{>l}\ \mathtt{v}^r \\
 & | & \mathtt{v}^r\ \mathtt{.}\ \mathtt{v}^r \\
 & | & \mathtt{g}^r\ \mathtt{\&}\ \mathtt{g}^r
\end{array}
$$

Additional restrictions on corpus graphs are:

1. The precedence relation on terminal nodes must be a total order.
2. For each node, its arity has to be fixed.
3. There is a single node, the root node, which does not occur on the righthand side of any labeled direct dominance expression.
4. Each node (except for the root node) occurs exactly once as the righthand side of a labeled direct dominance expression.

3.7.5 Denotation of a corpus definition

Whereas a conjunction of local constraints means the intersection of the respective denotations, this will not work e.g. for corpus definitions, since we would need a set of sets of syntax graphs as a component of the syntax graph model. In order to avoid that kind of complication, for global constraints, we immediately define the notion of satisfaction in a model $\mathcal{M}$.

Syntax	*Explanation*	*Interpretation*
$\mathtt{k} \Rightarrow \mathtt{g}^r$	single graph corpus	$\mathcal{M} \models \mathtt{g}^r$
$\quad\mid\ \mathtt{g}^r\ \mathtt{k}$	corpus	$\mathcal{M} \models \mathtt{g}^r$ and $\mathcal{M} \models \mathtt{k}$

Note (for future extensions of corpus graphs): there may be distinct variable substitutions σ, σ' for testing the satisfaction of the individual corpus graphs $\mathtt{g}^r$, $\mathtt{g}^{r\prime}$, i.e. the scope of a variable is a single corpus graph.

3.8 Queries

A *query* is simply a graph description.

3.9 Types

A type hierarchy is defined by a set of *type definitions* or type axioms a. A 'subtype' t' may be either a real type or a constant. A type t is the disjoint set union of its subtypes $\mathtt{t}_i$'. A *feature declaration* postulates the existence of a total function for its domain type. A type symbol t must be defined at most once, i.e. it may not occur more than once as the lefthand side of a type definition. A type symbol must occur exactly once in a righthand side of a type definition. For a feature symbol f, there must be exactly one feature declaration.

Syntax			*Explanation*	*Interpretation*
t'	$\Rightarrow$	c		
	$\mid$	t		
s	$\Rightarrow$	t'	subtype	
	$\mid$	t', s	subtypes	
a	$\Rightarrow$	t := s ;	type definition	$[\![\mathtt{t}]\!] = \bigcup_{1\leq i\leq n}[\![\mathtt{t}_i\text{'}]\!]$ and $[\![\mathtt{t}_i\text{'}]\!] \cap [\![\mathtt{t}_j\text{'}]\!] = \emptyset$, $1 \leq i < j \leq n$
	$\mid$	<feature name="f" domain="t" range="t"> </feature>	feature declaration	f is a total function $[\![\mathtt{t}_1]\!] \times [\![\mathtt{t}_2]\!]$

3.10 Template Definitions

The syntax and semantics of a template defining clause s is defined as follows.

Syntax			*Explanation*	*Interpretation*
s	$\Rightarrow$	r' <- g;	clause	$\exists\,\sigma$: if $\mathcal{M},\sigma \models \mathtt{g}$ then $\mathcal{M},\sigma \models \mathtt{r'}$

The scope of a variable covers a defining clause.

Note that the body of a clause is an embedded query. This means that graph descriptions always occur with the same 'polarity', i.e. only in queries, but never in template heads. Therefore, graph descriptions are interpreted in a uniform way (always in the 'if'-clause of the interpretation of a template definition).

3.11 Semantic Consequence

Definition 18 (TIGER database)

Let `k` *be a corpus definition,* `S` *a set of template definitions, and* `A` *a set of type definitions and feature declarations. A* TIGER database `DB` *is a triple* $\langle \texttt{k}, \texttt{S}, \texttt{A} \rangle$*.*

Definition 19 (Consequence)

A query `g` *is a* consequence *of a TIGER database* $\texttt{DB} = \langle \texttt{k}, \texttt{S}, \texttt{A} \rangle$ *if for all models* $\mathcal{M}$ *it holds that*

$$\textit{if } \mathcal{M} \models_{\texttt{A,S}} \texttt{k} \textit{ then } \mathcal{M} \models \texttt{g}.$$

As before, the scope of a variable does not exceed a graph description, i.e. a corpus graph or a query `g`.

Chapter 4

TIGER calculus

As a step towards interpreter for the TIGER language, we present a set of inference rules, the TIGER calculus. The calculus is a straightforward adaptation of the resolution calculus for Horn clauses in logic programming languages. Each graph in a corpus definition corresponds to a fact in the *database*, say with predicate name `graph` whose argument is the graph under consideration. Similarly, a query can be thought of as being enclosed by a `graph`-predicate name (template calls remain separate). The database includes also the template defining clauses. Queries (and template subgoals) are handled by an adaptation of the resolution rule. Instead of first order terms, we plug in graph descriptions (with embedded feature constraints). Accordingly, term unification is replaced by constraint solving. The TIGER calculus constitutes a naive interpreter for the TIGER language with the goal of keeping the correctness proofs with respect to the formal semantics as simple as possible. For an efficient implementation, certain portions of the calculus have to be redesigned. But then, it is sufficient to give correctness proofs with respect to the original TIGER calculus, not with respect to the formal semantics of the TIGER language.

Subsequently, we assume that there is a given TIGER database, i.e. a corpus definition `k`, a set of template definitions `S`, and a set of type definitions and feature declarations `A` which contains at least the type definition

$$\top \texttt{ := Constant, FRec, Node, Graph;} \tag{4.1}$$

Instead of a variable substitution σ, now a *constraint store* Σ and *store* Θ *of variable renamings* are used to keep track of variable substitutions. A constraint store Σ is a set of pairs $\langle \texttt{\#x}, \phi \rangle$, where the 'constraint' ϕ is either a feature value formula `d`, a feature constraint `k`, or a pair $\langle \texttt{c}, \texttt{\#x} \rangle$ of a constant node identifier `c` (or the $\top$-symbol) and a 'feature constraint variable' `#x` (i.e. $\{\, \sigma(\texttt{\#x}) \,\} \cap \mathcal{F} \neq \emptyset$). We assume that, initially, the constraint store Σ contains a pair $\langle \texttt{\#x}, \top \rangle$ resp. $\langle \texttt{\#x}, \langle \top, \texttt{\#x'} \rangle \rangle$ (`#x'` fresh feature constraint variable) for every possible variable `#x`. Or, more practically, in the case that a membership test for a variable `#x` in Σ would fail, one assumes the existence of the pair $\langle \texttt{\#x}, \top \rangle$ resp. $\langle \texttt{\#x}, \langle \top, \texttt{\#x'} \rangle \rangle$ in Σ (`#x'` fresh feature constraint variable).

The store Θ of variable renamings $\langle \texttt{\#x}_1, \texttt{\#x}_2 \rangle$ initially is an empty set. We assume that there exists a function $\texttt{deref(\#x)} \rightarrow \texttt{\#x}$ which is defined as follows:

$$\texttt{deref(\#x}_1\texttt{)} := \begin{cases} \texttt{\#x}_1 & \text{if } \langle \texttt{\#x}_1, \texttt{\#x}_2 \rangle \notin \Theta \\ \texttt{deref(\#x}_2\texttt{)} & \text{if } \langle \texttt{\#x}_1, \texttt{\#x}_2 \rangle \in \Theta \end{cases}$$

In order to illustrate how complex the treatment of variables in the scope of negation is, we add reduction rules for these cases, as well (but we refrain from the implementation of this part of the calculus.) For variables in the scope of negation, one more data structure is required: a store Δ of *'dif' constraints*[1] $\langle \texttt{\#x}_1, \texttt{\#x}_2 \rangle$. Initially, the dif store is empty. A dif constraint $\langle \texttt{\#x}_1, \texttt{\#x}_2 \rangle$ (on the level of feature values) is interpreted as

$$[\![\texttt{\#x}_1]\!] \cap (C \setminus [\![\texttt{\#x}_2]\!])$$

We chose the above interpretation from a variety of ways how two variables can be considered different:

- different variable names

  Two variables denote two (physically) non-identical structures.

- inequality

  The denotations of two variables are not equivalent, i.e.

  $\texttt{!((!\#x}_1 \texttt{ | \#x}_2\texttt{) \& (!\#x}_2 \texttt{ | \#x}_1\texttt{)).}$

- disjointness of denotations

  $\texttt{!(\#x}_1 \texttt{ \& \#x}_2\texttt{)}$

- the denotation of variable $\texttt{\#x}_2$ is a negative constraint on variable $\texttt{\#x}_1$:

  $\texttt{\#x}_1 \texttt{ \& !\#x}_2$

The last alternative fits the formal semantics of variables which occur in the scope of negation, e.g. in the formula $\texttt{!(f=\#x}_2\texttt{)}$, where $\texttt{\#x}_1$ stands for $\texttt{!(\#x}_2\texttt{)}$. Note that the substitution $\sigma\texttt{(\#x}_2\texttt{)}$ ignores whether $\texttt{\#x}_2$ occurs in a positive or in a negative context.

4.1 Feature Values

4.1.1 Disjunctive Normal Form:

Double negation elimination

$$\texttt{!(!d)} \quad \Longrightarrow \quad \texttt{d}$$

de Morgan rules

$$\texttt{!(d}_1 \texttt{ \& d}_2\texttt{)} \quad \Longrightarrow \quad \texttt{(!d}_1\texttt{) | (!d}_2\texttt{)}$$

$$\texttt{!(d}_1 \texttt{ | d}_2\texttt{)} \quad \Longrightarrow \quad \texttt{(!d}_1\texttt{) \& (!d}_2\texttt{)}$$

[1]This name has been adopted from [1] although the interpretation may be different from the one in [1].

Distribution rule

$$\texttt{(d}_1 \texttt{ | d}_2\texttt{) \& d}_3 \implies \texttt{(d}_1 \texttt{ \& d}_3\texttt{) | (d}_2 \texttt{ \& d}_3\texttt{)}$$

4.1.2 Disjunction reduction:

$$\texttt{c | c} \implies \texttt{c}$$

$$\texttt{c | (!c)} \implies \top$$

$$\texttt{(!c) | (!c)} \implies \texttt{(!c)}$$

$$\texttt{c | t} \implies \texttt{t} \quad \text{if } \texttt{c} \in [\![\texttt{t}]\!]$$

$$\texttt{c | (!t)} \implies \begin{cases} \top & \text{if } [\![\texttt{t}]\!] = \{\texttt{c}\} \\ \texttt{(!t)} & \text{if } \texttt{c} \notin [\![\texttt{t}]\!] \end{cases}$$

$$\texttt{(!c) | t} \implies \begin{cases} \top & \text{if } [\![\texttt{t}]\!] = \{\texttt{c}\} \\ \texttt{(!c)} & \text{if } \texttt{c} \notin [\![\texttt{t}]\!] \end{cases}$$

$$\texttt{(!c) | (!t)} \implies \begin{cases} \texttt{(!c)} & \text{if } \texttt{c} \in [\![\texttt{t}]\!] \\ \top & \text{if } \texttt{c} \notin [\![\texttt{t}]\!] \end{cases}$$

$$\texttt{t}_1 \texttt{ | t}_2 \implies \begin{cases} \texttt{t}_2 & \text{if } [\![\texttt{t}_1]\!] \subseteq [\![\texttt{t}_2]\!] \\ \texttt{t}_1 & \text{if } [\![\texttt{t}_2]\!] \subset [\![\texttt{t}_1]\!] \end{cases}$$

$$\texttt{t}_1 \texttt{ | (!t}_2\texttt{)} \implies \begin{cases} \top & \text{if } [\![\texttt{t}_2]\!] \subseteq [\![\texttt{t}_1]\!] \\ \texttt{(!t}_2\texttt{)} & \text{if } [\![\texttt{t}_1]\!] \cap [\![\texttt{t}_2]\!] = \emptyset \end{cases}$$

$$\texttt{(!t}_1\texttt{) | (!t}_2\texttt{)} \implies \begin{cases} \texttt{(!t}_1\texttt{)} & \text{if } [\![\texttt{t}_1]\!] \subseteq [\![\texttt{t}_2]\!] \\ \texttt{(!t}_2\texttt{)} & \text{if } [\![\texttt{t}_2]\!] \subset [\![\texttt{t}_1]\!] \\ \top & \text{if } [\![\texttt{t}_1]\!] \cap [\![\texttt{t}_2]\!] = \emptyset \end{cases}$$

4.1.3 Consistency check:

Constant-constant merge

$$\texttt{c}_1 \texttt{ \& c}_2 \implies \begin{cases} \texttt{c}_1 & \text{if } \texttt{c}_1 = \texttt{c}_2 \\ \bot & \text{otherwise} \end{cases}$$

$$\texttt{c}_1 \texttt{ \& (!c}_2\texttt{)} \implies \begin{cases} \texttt{c}_1 & \text{if } \texttt{c}_1 \neq \texttt{c}_2 \\ \bot & \text{otherwise} \end{cases}$$

$$\texttt{(!c) \& (!c)} \implies \texttt{(!c)}$$

Type check

$$\texttt{c \& t} \implies \begin{cases} \texttt{c} & \text{if } \texttt{c} \in [\![\texttt{t}]\!] \\ \bot & \text{otherwise} \end{cases}$$

$$\texttt{c \& (!t)} \implies \begin{cases} \texttt{c} & \text{if } \texttt{c} \notin [\![\texttt{t}]\!] \\ \bot & \text{otherwise} \end{cases}$$

$$\texttt{(!c) \& t} \implies \texttt{t} \quad \text{if } \texttt{c} \notin [\![\texttt{t}]\!]$$

$$\texttt{(!c) \& (!t)} \quad \Longrightarrow \quad \texttt{(!t)} \qquad \text{if } \texttt{c} \in [\![\texttt{t}]\!]$$

Type-type merge

$$\texttt{t}_1 \texttt{ \& } \texttt{t}_2 \quad \Longrightarrow \quad \begin{cases} \texttt{t}_1 & \text{if } [\![t_1]\!] \subseteq [\![t_2]\!] \\ \texttt{t}_2 & \text{if } [\![\texttt{t}_2]\!] \subset [\![\texttt{t}_1]\!] \\ \perp & \text{otherwise} \end{cases}$$

$$\texttt{t}_1 \texttt{ \& (!t}_2\texttt{)} \quad \Longrightarrow \quad \begin{cases} \texttt{t}_1 & \texttt{if } [\![\texttt{t}_2]\!] \cap [\![\texttt{t}_1]\!] = \emptyset \\ \perp & \texttt{if } [\![\texttt{t}_1]\!] \subseteq [\![\texttt{t}_2]\!] \end{cases}$$

$$\texttt{(!t)}_1 \texttt{ \& (!t}_2\texttt{)} \quad \Longrightarrow \quad \begin{cases} \texttt{(!t}_2\texttt{)} & \text{if } [\![\texttt{t}_1]\!] \subseteq [\![\texttt{t}_2]\!] \\ \texttt{(!t}_1\texttt{)} & \text{if } [\![\texttt{t}_2]\!] \subset [\![\texttt{t}_1]\!] \end{cases}$$

Lemma 1 (Completeness)

If $[\![\texttt{d}]\!] = \emptyset$ *then* $\texttt{d} \stackrel{*}{\Longrightarrow} \perp$

Proof: (sketch)

Assume that there exists a case where $[\![\texttt{d}]\!] = \emptyset$ but not $\texttt{d} \stackrel{*}{\Longrightarrow} \perp$.

However, all syntactic combinations (of Boolean operators and types of subformulas) have been covered by the reduction rules. Furthermore, whenever a lefthand side of a reduction rule is inconsistent, its lefthand side is an inconsistent formula, as well. In particular, a clash $\perp$ will be propagated in the cases $\texttt{(c \& }\perp\texttt{)}$, $\texttt{(c \& (!}\top\texttt{))}$, $\texttt{((!c) \& }\perp\texttt{)}$, $\texttt{((!c) \& (!}\top\texttt{)}$, $\texttt{(}\perp\texttt{ \& t)}$, $\texttt{(t \& (!}\top\texttt{)}$, $\texttt{((!}\top\texttt{) \& (!t))}$, $\texttt{(}\perp \texttt{ | } \perp\texttt{)}$, $\texttt{(}\perp\texttt{ | (!}\top\texttt{))}$, $\texttt{((!}\top\texttt{) | (!}\top\texttt{))}$ by the corresponding constant-constant merge rule, type check rule, or disjunction reduction rule. Hence, we get a contradiction to the assumption.

Lemma 2 (Soundness)

If $\texttt{d} \stackrel{*}{\Longrightarrow} \perp$ *then* $[\![\texttt{d}]\!] = \emptyset$

Proof: (sketch)

Assume that $\texttt{d} \stackrel{*}{\Longrightarrow} \perp$ but not $[\![\texttt{d}]\!] = \emptyset$.

However, in every reduction rule, the denotation of the lefthand side equals the denotation of the righthand side, therefore this assumption cannot hold. For example, the treatment of $\texttt{(t}_1\texttt{ \& (!t}_2\texttt{))}$ is sound, since types are organized in a hierarchy. There are three cases:

1. $[\![\texttt{t}_2]\!] \cap [\![\texttt{t}_1]\!] = \emptyset$:
   $(C \setminus [\![\texttt{t}_2]\!]) \cap [\![\texttt{t}_1]\!] = C \cap [\![\texttt{t}_1]\!] = [\![\texttt{t}_1]\!]$
2. $[\![\texttt{t}_1]\!] \subseteq [\![\texttt{t}_2]\!]$:
   $(C \setminus [\![\texttt{t}_2]\!]) \cap [\![\texttt{t}_1]\!] = (C \setminus [\![\texttt{t}_2]\!]) \cap [\![\texttt{t}_2]\!] = \emptyset$

3. $[\![\mathtt{t}_2]\!] \subset [\![\mathtt{t}_1]\!]$ cannot be reduced to a more concise form

Lemma 3

$[\![\mathtt{d}]\!] = \emptyset \quad \textit{iff} \quad \mathtt{d} \overset{*}{\Longrightarrow} \bot$

4.2 Feature Constraints

The function `evalre(z)` takes a regular expression `z` and returns a disjunction of constants ($\mathtt{c}_1$|...|$\mathtt{c}_n$). As an 'internal' representation, we admit the formula `!#x`.

4.2.1 Disjunctive Normal Form:

Feature Value Disjunction

$$\mathtt{f = (d_1 \mid d_2)} \quad \Longrightarrow \quad \mathtt{(f = d_1) \mid (f = d_2)}$$

$$\mathtt{f = \#x{:}(d_1 \mid d_2)} \quad \Longrightarrow \quad \mathtt{(f = \#x{:}d_1) \mid (f = \#x{:}d_2)}$$

Negated feature reduction ($\mathtt{t}_1$ domain of $\mathtt{f}$)

$$\mathtt{!(f = d)} \quad \Longrightarrow \quad \mathtt{(!t_1) \mid (f = !d)}$$

$$\mathtt{!(f = \#x_2)} \quad \Longrightarrow \quad \mathtt{(!t_1) \mid (f = !\#x_2)}$$

$$\mathtt{!(f = \#x{:}d)} \quad \Longrightarrow \quad \mathtt{!((f = \#x) \;\&\; (f = d))}$$

$$\mathtt{!(f\ != d)} \quad \Longrightarrow \quad \mathtt{!(f = !d)}$$

$$\mathtt{!(f = /z/)} \quad \Longrightarrow \quad \mathtt{!(f = evalre(z))}$$

$$\mathtt{!(f\ != /z/)} \quad \Longrightarrow \quad \mathtt{!(f = !evalre(z))}$$

Double negation elimination

$$\mathtt{!(!k)} \quad \Longrightarrow \quad \mathtt{k}$$

de Morgan rules

$$\mathtt{!(k_1 \;\&\; k_2)} \quad \Longrightarrow \quad \mathtt{(!k_1) \mid (!k_2)}$$

$$\mathtt{!(k_1 \mid k_2)} \quad \Longrightarrow \quad \mathtt{(!k_1) \;\&\; (!k_2)}$$

Distribution rule

$$\mathtt{(k_1 \mid k_2) \;\&\; k_3} \quad \Longrightarrow \quad \mathtt{(k_1 \;\&\; k_3) \mid (k_2 \;\&\; k_3)}$$

4.2.2 Disjunction reduction:

$$\bot \mid \mathtt{k} \quad \Longrightarrow \quad \mathtt{k}$$

$$\mathtt{t}_1 \mid \mathtt{t}_2 \quad \Longrightarrow \quad \begin{cases} \mathtt{t}_2 & \text{if } [\![\mathtt{t}_1]\!] \subseteq [\![\mathtt{t}_2]\!] \\ \mathtt{t}_1 & \text{if } [\![\mathtt{t}_2]\!] \subset [\![\mathtt{t}_1]\!] \end{cases}$$

$$\mathtt{t}_1 \mid (!\mathtt{t}_2) \quad \Longrightarrow \quad \begin{cases} \top & \text{if } [\![\mathtt{t}_2]\!] \subseteq [\![\mathtt{t}_1]\!] \\ (!\mathtt{t}_2) & \text{if } [\![\mathtt{t}_1]\!] \cap [\![\mathtt{t}_2]\!] = \emptyset \end{cases}$$

$$(!\mathtt{t}_1) \mid (!\mathtt{t}_2) \quad \Longrightarrow \quad \begin{cases} (!\mathtt{t}_1) & \text{if } [\![\mathtt{t}_1]\!] \subseteq [\![\mathtt{t}_2]\!] \\ (!\mathtt{t}_2) & \text{if } [\![\mathtt{t}_2]\!] \subset [\![\mathtt{t}_1]\!] \\ \top & \text{if } [\![\mathtt{t}_1]\!] \cap [\![\mathtt{t}_2]\!] = \emptyset \end{cases}$$

The subsequent rules apply to the individual disjuncts of the Disjunctive Normal Form. For the evaluation of each disjunct, there are fresh stores Σ, Θ, Δ.

4.2.3 Feature normalization:

$$\mathtt{f = !\#x_2} \quad \Longrightarrow \quad \mathtt{f = \#x_1}$$

$$\Delta \;\mathtt{:=}\; \Delta \cup \{\langle \mathtt{\#x_1}, \mathtt{\#x_2} \rangle\},\ \mathtt{\#x_1} \text{ fresh variable}$$

$$\mathtt{f = d} \quad \Longrightarrow \quad \mathtt{f = \#x} \qquad \Sigma \;\mathtt{:=}\; \Sigma \cup \{\langle \mathtt{\#x}, \mathtt{d} \rangle\},\ \mathtt{\#x} \text{ fresh variable}$$

$$\mathtt{f = \#x} \quad \Longrightarrow \quad \mathtt{t_1\ \&\ f{=}\#x}$$

$$\Sigma \;\mathtt{:=}\; (\Sigma \setminus \{\langle \mathtt{deref(\#x)}, \mathtt{d} \rangle\}) \cup \{\langle \mathtt{deref(\#x)}, (\mathtt{Constant\ \&\ t_2\ \&\ d}) \rangle\}$$

$$\mathtt{t}_1 \text{ domain}\mathtt{,}\ \mathtt{t}_2 \text{ range of } \mathtt{f}$$

$$\mathtt{f = \#x{:}d_1} \quad \Longrightarrow \quad \mathtt{f = \#x}$$

$$\Sigma := (\Sigma \setminus \{\langle \mathtt{deref(\#x)}, \mathtt{d_2} \rangle\}) \cup \{\langle \mathtt{deref(\#x)}, (\mathtt{d_1\ \&\ d_2}) \rangle\}$$

$$\mathtt{f\ !=\ d} \quad \Longrightarrow \quad \mathtt{f = !d}$$

$$\mathtt{f = /z/} \quad \Longrightarrow \quad \mathtt{f = evalre(z)}$$

$$\mathtt{f\ !=\ /z/} \quad \Longrightarrow \quad \mathtt{f = !evalre(z)}$$

4.2.4 Consistency check:

Constraint store

$$\mathtt{k} \quad \Longrightarrow \quad \bot \qquad \text{if } \langle \mathtt{\#x}, \bot \rangle \in \Sigma$$

Dif store

$$\mathtt{k} \quad \Longrightarrow \quad \bot$$

$$\langle \mathtt{\#x_1}, \mathtt{\#x_2} \rangle \in \Delta;\ \langle \mathtt{deref(\#x_1)}, \mathtt{d_1} \rangle, \langle \mathtt{deref(\#x_2)}, \mathtt{d_2} \rangle \in \Sigma$$

$$\text{if} \quad (\mathtt{d_1\ \&\ !d_2}) \quad \Longrightarrow \quad \bot$$

Failure propagation

$$\bot\ \mathtt{\&}\ \mathtt{k} \quad \Longrightarrow \quad \bot$$

Type-type merge

$$\texttt{t}_1 \ \texttt{\&} \ \texttt{t}_2 \quad \Longrightarrow \quad \begin{cases} \texttt{t}_1 & \text{if } [\![t_1]\!] \subseteq [\![t_2]\!] \\ \texttt{t}_2 & \text{if } [\![\texttt{t}_2]\!] \subset [\![\texttt{t}_1]\!] \\ \bot & \text{otherwise} \end{cases}$$

$$\texttt{t}_1 \ \ \texttt{\& (!t}_2\texttt{)} \quad \Longrightarrow \quad \begin{cases} \texttt{t}_1 & \texttt{if } [\![\texttt{t}_2]\!] \cap [\![\texttt{t}_1]\!] = \emptyset \\ \bot & \texttt{if } [\![\texttt{t}_1]\!] \subseteq [\![\texttt{t}_2]\!] \end{cases}$$

$$\texttt{(!t)}_1 \ \ \texttt{\& (!t}_2\texttt{)} \quad \Longrightarrow \quad \begin{cases} \texttt{(!t}_2\texttt{)} & \text{if } [\![\texttt{t}_1]\!] \subseteq [\![\texttt{t}_2]\!] \\ \texttt{(!t}_1\texttt{)} & \text{if } [\![\texttt{t}_2]\!] \subset [\![\texttt{t}_1]\!] \end{cases}$$

Feature-feature merge

$$\begin{aligned} \texttt{(f = \#x}_1\texttt{) \& (f = \#x}_2\texttt{)} \quad &\Longrightarrow \quad \texttt{f = \#x}_1 \\ \Sigma &:= (\Sigma \setminus \{\langle \texttt{deref(\#x}_1\texttt{)}, \texttt{d}_1\rangle, \langle \texttt{deref(\#x}_2\texttt{)}, \texttt{d}_2\rangle\}) \\ &\quad \cup \{\langle \texttt{deref(\#x}_1\texttt{)}, \texttt{(d}_1 \ \texttt{\&} \ \texttt{d}_2\texttt{)}\rangle\} \\ \Theta &:= \Theta \cup \{\langle \texttt{deref(\#x}_2\texttt{)}, \texttt{deref(\#x}_1\texttt{)}\rangle\} \end{aligned}$$

For the subsequent proofs, let `k'` be a single disjunct of the disjunctive normal form, i.e. a disjunction-free formula.

Lemma 4 *(Completeness, single disjunct)*

If $\quad [\![\texttt{k'}]\!] = \emptyset \quad$ *then* $\quad \langle \texttt{k'}, \Sigma, \Theta, \Delta \rangle \stackrel{*}{\Longrightarrow} \langle \bot, \Sigma', \Theta', \Delta' \rangle$

Proof: (sketch)

Similar argument as in the proof sketch of Lemma 1. Note that the reduction rules also reduce the formulas in the constraint store Σ.

In addition, one has to check that the constraint store and the store of variable renamings are handled in such a manner that all inconsistencies will be discovered:

A chain of variable renamings denotes an equivalence class of variables. The final element (the `deref`-value) of such a chain is the representative of the equivalence class.

There is at most one constraint per variable in the constraint store:

For example, the rule for feature-feature merge erases the constraints for the 'dereferenced variables' $\texttt{\#x}_1$ and $\texttt{\#x}_2$ from the constraint store and adds the accumulated constraints for $\texttt{\#x}_1$. The fact that $\texttt{\#x}_1$ and $\texttt{\#x}_2$ belong to the same equivalence class of variables is realized by the appropriate extension of the store of variable renamings.

Furthermore, the termination property must be checked:

It is sufficient to apply the type check for features only once to each feature in the feature constraint.

It is sufficient to carry out an inconsistency check of the dif store each time the Σ- or Θ-information on one of the variables contained in the dif store has been extended. The inconsistency check of the dif store will not create new dif constraints, since the updated Σ-constraint $\texttt{(d}_1\texttt{\&!d}_2\texttt{)}$ does not contain any variables.

Lemma 5 *(Soundness, single disjunct)*

If $\quad \langle \texttt{k'}, \Sigma, \Theta, \Delta \rangle \stackrel{*}{\Longrightarrow} \langle \bot, \Sigma', \Theta', \Delta' \rangle \quad$ *then* $\quad [\![\texttt{k'}]\!] = \emptyset$

Proof: (sketch)

Similar argument as in the proof sketch of Lemma 2. Check for example the equivalence of the lefthand side and righthand side of the first rule for negated feature reduction, cf. [3].

$$
\begin{aligned}
&[\![\texttt{!(f = d)}]\!] \\
&= \mathcal{F} \setminus [\![\texttt{f = d}]\!] \\
&= \mathcal{F} \setminus ([\![\texttt{t}_1]\!] \cap [\![\texttt{f = d}]\!]) \\
&= (\mathcal{F} \setminus [\![\texttt{t}_1]\!]) \cup (\mathcal{F} \setminus [\![\texttt{f = d}]\!]) \\
&= (\mathcal{F} \setminus [\![\texttt{t}_1]\!]) \cup ((\mathcal{F} \setminus [\![\texttt{t}_1]\!]) \setminus [\![\texttt{f = d}]\!]) \\
&= [\![\texttt{!t}_1]\!] \cup [\![\texttt{f = !d}]\!]
\end{aligned}
$$

The rule for $\texttt{!(f = \#x}_2\texttt{)}$ works together with the check of the dif store, which guarantees that the denotation of the value of feature f will be a subset of $\mathcal{F} \setminus [\![\texttt{\#x}_2]\!]$.

Lemma 6

$[\![\texttt{d}]\!] = \emptyset \quad$ *iff* $\quad \texttt{d} \stackrel{*}{\Longrightarrow} \bot$

Proof: (sketch)

Completeness:

Show that all feature constraints will be reduced to disjunctive normal form (DNF). By Lemma 4, the calculus is complete wrt. an individual disjunct of the DNF. The rules of type disjunction and disjunction reduction will reduce an inconsistent DNF to $\bot$.

Soundness:

Show that for each DNF rule, the lefthand side is equivalent to its righthand side. For example:

$$
\begin{aligned}
&[\![\texttt{f = (d}_1\texttt{| d}_2\texttt{)}]\!] \;=\; [\![\texttt{f = \#x:(d}_1\texttt{| d}_2\texttt{)}]\!] \;=\; [\![\texttt{(f = \#x:d}_1\texttt{) | (f = \#x:d}_2\texttt{)}]\!] \\
&= [\![\texttt{f = \#x:d}_1]\!] \cup [\![\texttt{f = \#x:d}_2]\!] \;=\; [\![\texttt{f = d}_1]\!] \cup [\![\texttt{f = d}_2]\!] \;=\; [\![\texttt{(f = d}_1\texttt{) | (f = d}_2\texttt{)}]\!]
\end{aligned}
$$

4.3 Node Descriptions

Subsequently, we assume that constant node identifiers occur only in corpus graphs.

4.3.1 Disjunctive Normal Form:

Node description normalization

$$\texttt{[k]} \implies \texttt{[\#x:k]} \qquad \texttt{\#x} \text{ fresh variable}$$

$$\texttt{[\#x}_2\texttt{:k]} \implies \texttt{\#x}_1\texttt{:[\#x}_2\texttt{:k]} \qquad \texttt{\#x}_1 \text{ fresh variable}$$

$$\texttt{\#x}_1\texttt{:[k]} \implies \texttt{\#x}_1\texttt{:[\#x}_2\texttt{:k]} \qquad \texttt{\#x}_2 \text{ fresh variable}$$

Feature constraint disjunction

$$\texttt{\#x}_1\texttt{:[\#x}_2\texttt{:(k}_1 \texttt{ | k}_2\texttt{)]} \implies \texttt{\#x}_1\texttt{:[\#x}_2\texttt{:k}_1\texttt{] | \#x}_1\texttt{:[\#x}_2\texttt{:k}_2\texttt{]}$$

The remaining rules for node descriptions apply to the individual disjuncts of the Disjunctive Normal Form. We assume fresh stores Σ, Θ, Δ for each disjunct.

4.3.2 Node description normalization:

$$\texttt{[ ]} \implies \texttt{\#x} \qquad \texttt{\#x} \text{ fresh variable}$$

$$\texttt{c:[k]} \implies \texttt{\#x}_1 \qquad \texttt{\#x}_1,\ \texttt{\#x}_2 \text{ fresh variables}$$

$$\Sigma := \Sigma \cup \{\langle \texttt{\#x}_1, \langle \texttt{c}, \texttt{\#x}_2\rangle\rangle, \langle \texttt{\#x}_2, \texttt{k}\rangle\}$$

$$\texttt{\#x:[ ]} \implies \texttt{\#x}$$

$$\texttt{\#x}_1\texttt{:[\#x}_2\texttt{]} \implies \texttt{\#x}_1 \qquad \langle \texttt{deref(\#x}_1\texttt{)}, \langle \texttt{c}_3, \texttt{\#x}_3\rangle\rangle \in \Sigma$$

$$\Sigma := (\Sigma \setminus \{\langle \texttt{deref(\#x}_2\texttt{)}, \texttt{k}_2\rangle, \langle \texttt{\#x}_3, \texttt{k}_3\rangle\}) \cup \{\langle \texttt{\#x}_3, (\texttt{k}_2 \ \& \ \texttt{k}_3)\rangle\}$$

$$\Theta := \Theta \cup \{\langle \texttt{deref(\#x}_2\texttt{)}, \texttt{\#x}_3\rangle\}$$

$$\texttt{\#x}_1\texttt{:[\#x}_2\texttt{:k}_2\texttt{]} \implies \texttt{\#x}_1\texttt{:[\#x}_2\texttt{]}$$

$$\Sigma := (\Sigma \setminus \{\langle \texttt{deref(\#x}_2\texttt{)}, \texttt{k}_3\rangle\}) \cup \{\langle \texttt{deref(\#x}_2\texttt{)}, (\texttt{k}_2 \ \& \ \texttt{k}_3)\rangle\}$$

4.3.3 Node guess:

Here, we need the failure symbol $\perp$ as a kind of 'node identifier' - in the case that $\texttt{c}_1$ and $\texttt{c}_2$ are not the same constants.

$$\texttt{\#x}_1 \implies \texttt{\#x}_2$$

for some $\texttt{\#x}_2$ in the corpus graph

$$\begin{aligned}\Sigma := (\Sigma \ \setminus \ & \{\langle \texttt{deref(\#x}_1\texttt{)}, \langle \texttt{c}_1, \texttt{\#x}_3\rangle\rangle, \langle \texttt{\#x}_2, \langle \texttt{c}_2, \texttt{\#x}_4\rangle\rangle, \\ & \langle \texttt{\#x}_3, \texttt{k}_3\rangle, \langle \texttt{\#x}_4, \texttt{k}_4\rangle\}) \\ & \cup \ \{\langle \texttt{\#x}_2, \langle \texttt{c}_1 \ \& \ \texttt{c}_2, \texttt{\#x}_4\rangle\rangle, \langle \texttt{\#x}_4, \texttt{k}_3 \ \& \ \texttt{k}_4\rangle\}\end{aligned}$$

$$\Theta := \Theta \cup \{\langle \texttt{deref(\#x}_1\texttt{)}, \texttt{\#x}_2\rangle, \langle \texttt{deref(\#x}_3\texttt{)}, \texttt{\#x}_4\rangle\}$$

4.4 Graph Descriptions

Subsequently, we admit the failure symbol $\perp$ as a graph description.

4.4.1 Disjunctive Normal Form:

Distribution rule

$(g_1 \mid g_2)\ \&\ g_3 \quad \Longrightarrow \quad (g_1\ \&\ g_3) \mid (g_2\ \&\ g_3)$

4.4.2 Disjunction reduction:

$\perp \mid g \quad \Longrightarrow \quad g$

4.4.3 Graph description normalization:

Elimination of trivial node relations

$\#x\ \&\ g \quad \Longrightarrow \quad g$

Node relation normalization

$\#x_1 > \#x_2 \quad \Longrightarrow \quad \#x_1 >1\ \#x_2$

$\#x_1\ .\ \#x_2 \quad \Longrightarrow \quad \#x_1\ .1\ \#x_2$

$\#x_1 >l\ \#x_2 \quad \Longrightarrow \quad (\#x_1 >l\ \#x_2)\ \&\ (\#x_1 >1\ \#x_2)$

$\#x_1\ \$.*\ \#x_2 \quad \Longrightarrow \quad (\#x_1\ \$\ \#x_2)\ \&\ (\#x_1\ .*\ \#x_2)$

Corpus graph transitive closure

$(\#x_1 >a\ \#x_2)\ \&\ (\#x_2 >b\ \#x_3) \quad \Longrightarrow$

$\qquad (\#x_1 >a\ \#x_2)\ \&\ (\#x_2 >b\ \#x_3)\ \&\ (\#x_1 >(a+b)\ \#x_3)$

$(\#x_1\ .a\ \#x_2)\ \&\ (\#x_2\ .b\ \#x_3) \quad \Longrightarrow$

$\qquad (\#x_1\ .a\ \#x_2)\ \&\ (\#x_2\ .b\ \#x_3)\ \&\ (\#x_1\ .(a+b)\ \#x_3)$

$\#x \quad \Longrightarrow \quad$ `tokenarity(#x,b)`

where `b` is the number of terminal leaves dominated by `#x`

The following rules apply to the individual disjuncts of the disjunctive normal form (of the query), with fresh stores Σ, Θ, Δ for each disjunct.

4.4.4 Consistency check:

Constraint store

$$\texttt{g} \quad \Longrightarrow \quad \bot \qquad \text{if } \langle \texttt{\#x}, \bot\rangle \text{ or } \langle \texttt{\#x}_1, \langle \bot, \texttt{\#x}_2\rangle\rangle \in \Sigma$$

Failure propagation

$$\bot \ \texttt{\&}\ \texttt{g} \quad \Longrightarrow \quad \bot$$

Arity (`arity(#x,b')` in the corpus graph)

$$\texttt{arity(\#x,a)} \quad \Longrightarrow \quad \begin{cases} \texttt{\#x} & \text{if } \texttt{a} = \texttt{b'} \\ \bot & \text{if } \texttt{a} \neq \texttt{b'} \end{cases}$$

$$\texttt{arity(\#x,a,b)} \quad \Longrightarrow \quad \begin{cases} \texttt{\#x} & \text{if } \texttt{a} \leq \texttt{b'} \leq \texttt{b} \\ \bot & \text{if } \texttt{b'} < \texttt{a} \texttt{ or } \texttt{b'} > \texttt{b} \end{cases}$$

Token arity (`tokenarity(#x,b')` in the corpus graph)

$$\texttt{tokenarity(\#x,a)} \quad \Longrightarrow \quad \begin{cases} \texttt{\#x} & \text{if } \texttt{a} = \texttt{b'} \\ \bot & \text{if } \texttt{a} \neq \texttt{b'} \end{cases}$$

$$\texttt{tokenarity(\#x,a,b)} \quad \Longrightarrow \quad \begin{cases} \texttt{\#x} & \text{if } \texttt{a} \leq \texttt{b'} \leq \texttt{b} \\ \bot & \text{if } \texttt{b'} < \texttt{a} \texttt{ or } \texttt{b'} > \texttt{b} \end{cases}$$

Node relations

$$\texttt{\#x >a \#x} \quad \Longrightarrow \quad \bot$$

$$\texttt{\#x}_1 \texttt{ >1 } \texttt{\#x}_2 \quad \Longrightarrow \quad \texttt{\#x}_2$$

if $(\texttt{\#x}_1 \texttt{ >1 } \texttt{\#x}_2)$ in the corpus graph

$$\texttt{\#x}_1 \texttt{ >* } \texttt{\#x}_2 \quad \Longrightarrow \quad \texttt{\#x}_1$$

if $(\texttt{\#x}_1 \texttt{ >a } \texttt{\#x}_2)$ in the corpus graph

$$\texttt{\#x}_1 \texttt{ >a } \texttt{\#x}_2 \quad \Longrightarrow \quad \texttt{\#x}_1$$

if $(\texttt{\#x}_1 \texttt{ >a } \texttt{\#x}_2)$ in the corpus graph

$$\texttt{\#x}_1 \texttt{ >a,b } \texttt{\#x}_2 \quad \Longrightarrow \quad \texttt{\#x}_1$$

if $(\texttt{\#x}_1 \texttt{ >a' } \texttt{\#x}_2)$ in the corpus graph, $\texttt{a} \leq \texttt{a'} \leq \texttt{b}$

$$\texttt{\#x}_1 \texttt{ >@l } \texttt{\#x}_2 \quad \Longrightarrow \quad \texttt{\#x}_1$$

if $\texttt{\#x}_2$ is left corner of $\texttt{\#x}_1$

$$\texttt{\#x}_1 \texttt{ >@r } \texttt{\#x}_2 \quad \Longrightarrow \quad \texttt{\#x}_1$$

if $\texttt{\#x}_2$ is right corner of $\texttt{\#x}_1$

$$\texttt{\#x . \#x} \quad \Longrightarrow \quad \bot$$

$$\texttt{\#x}_1 \texttt{ .* } \texttt{\#x}_2 \quad \Longrightarrow \quad \texttt{\#x}_1$$

if $(\texttt{\#x}_1 \texttt{ .a } \texttt{\#x}_2)$ in the corpus graph

$$\texttt{\#x}_1 \texttt{ .a } \texttt{\#x}_2 \quad \Longrightarrow \quad \texttt{\#x}_1$$

if $(\texttt{\#x}_1 \texttt{ .a } \texttt{\#x}_2)$ in the corpus graph

$$\texttt{\#x}_1 \texttt{ .a,b } \texttt{\#x}_2 \quad \Longrightarrow \quad \texttt{\#x}_1$$

if $(\#x_1\ .a'\ \#x_2)$ in the corpus graph, $a \leq a' \leq b$

$\#x_1\ \$\ \#x_2 \quad \Longrightarrow \quad \#x_1$

if $(\#x_0 > \#x_1)$, $(\#x_0 > \#x_2)$ in the corpus graph, $\#x_1 \neq \#x_2$

$\#x_1\ !{>}l\ \#x_2 \quad \Longrightarrow \quad \#x_2$

if $(\#x_1\ {>}l\ \#x_2)$ not in the corpus graph

$\#x_1\ !{>}\ \#x_2 \quad \Longrightarrow \quad \#x_2$

if $\forall l\ (\#x_1\ {>}l\ \#x_2)$ not in the corpus graph

$\#x_1\ !{>}{*}\ \#x_2 \quad \Longrightarrow \quad \#x_1$

if $\forall a\ (\#x_1\ {>}a\ \#x_2)$ not in the corpus graph

$\#x_1\ !{>}a\ \#x_2 \quad \Longrightarrow \quad \#x_1$

if $(\#x_1\ {>}a\ \#x_2)$ not in the corpus graph

$\#x_1\ !{>}a,b\ \#x_2 \quad \Longrightarrow \quad \#x_1$

if $(\#x_1\ {>}a'\ \#x_2)$ in the corpus graph, $a' < a$ or $b < a'$

$\#x_1\ !{>}@l\ \#x_2 \quad \Longrightarrow \quad \#x_1$

if $\#x_2$ is not left corner of $\#x_1$

$\#x_1\ !{>}@r\ \#x_2 \quad \Longrightarrow \quad \#x_1$

if $\#x_2$ is not right corner of $\#x_1$

$\#x_1\ !\$\ \#x_2 \quad \Longrightarrow \quad \#x_1$

if $\exists\ \#x_0$ s.t. $(\#x_0\ {>}1\ \#x_1)$ or $(\#x_0\ {>}1\ \#x_2)$ not in the corpus graph

$\#x_1\ !.\ \#x_2 \quad \Longrightarrow \quad \#x_1$ if $(\#x_1\ .\ \#x_2)$ not in the corpus graph

$\#x_1\ !.{*}\ \#x_2 \quad \Longrightarrow \quad \#x_1$

if $\forall a\ (\#x_1\ .a\ \#x_2)$ not in the corpus graph

$\#x_1\ !.a\ \#x_2 \quad \Longrightarrow \quad \#x_1$

if $(\#x_1\ .a\ \#x_2)$ not in the corpus graph

$\#x_1\ !.a,b\ \#x_2 \quad \Longrightarrow \quad \#x_1$

if $(\#x_1\ .a'\ \#x_2)$ in the corpus graph, $a' < a$ or $b < a'$

Bibliography

[1] F. Giannesini, H. Kanoui, R. Pasero, and M. van Caneghem. *Prolog*. InterÉdition, Paris, 1985.

[2] Esther König and Wolfgang Lezius. The TIGER language - a description language for syntax graphs. Part 1: User's guidelines. Technical report, IMS, University of Stuttgart, 2001.

[3] Gert Smolka. A feature logic with subsorts. Technical Report 33, IBM Deutschland, Institute for Knowledge-Based Systems, Stuttgart, Baden-Württemberg, 1988.