Boolean Index Expressions for Information Retrieval

B.C.M. Wondergem, P. van Bommel, and Th.P. van der Weide
Computing Science Institute, University of Nijmegen
Toernooiveld 1, NL-6525 ED, Nijmegen, The Netherlands
Tel: +31 24 3653147, Fax: +31 24 3553450
E-mail: bernd@cs.kun.nl

Keywords: Information Retrieval, Characterisation and Query Languages, Boolean Operators, Normalisation.

Technical Report CSI-R9827

Abstract

Keywords still seem to form the basis for document content and query representation. Approaches to use more advanced linguistic structures, such as noun phrases, still are in an experimental phase. In addition, Boolean descriptor languages have often been applied for Information Retrieval. However, the synthesis of logic and linguistics in one descriptor language still is an open issue. In this paper, Boolean index expressions, combining Boolean logic and linguistic structure, are proposed as a good balance between expresiveness and practical issues. Boolean index expressions are obtained by augmenting regular index expressions with logical operators for disjunction, conjunction, and negation. Boolean index expressions are more expressive than both index expressions and the Boolean query language based on keywords. They allow a compact representation of logical combinations of index expressions. In addition, Boolean index expressions are still efficiently parsible and their meaning can be determined through their structure. It is shown how Boolean index expressions can be brought into normal form, allowing fast numerical matching. Matching strategies for Boolean index expressions are obtained by adapting matching strategies for index expressions by providing a case for negations. Our implementation of Boolean index expressions illustrates mentioned issues.

1 Introduction

Boolean descriptor languages, supporting logical structure in queries by using operators, have often been applied for Information Retrieval. On the other hand, query languages that are linguistically motivated, such as noun-phrases and index expressions, have proven of value for IR as well. However, the synthesis of logic and linguistics in one descriptor language still is an open issue.

In this article, we present *Boolean index expressions*, which combine the advantageous features of index expressions, that capture some of the structure of natural languages, with the well-known logical operators for disjunction, conjunction, and negation. Boolean index expressions are proposed as a good balance between expressiveness and practical issues.

The advantage for IR of Boolean index expressions over index expressions is threefold. First, they allow Boolean structure by using logical operators. This allows, for example, simple inclusion of synonyms (disjunctions) and phrase-like constructs (conjunctions). Furthermore, high standards of performance are achievable through the logical operators ([Salton et al., 1983]).

Second, Boolean index expressions serve as a compact representation of (logical) combinations of index expressions. This aspect exploits nested occurrences of logical operators. The compact representation may serve as a basis for efficient filtering ([Wondergem et al., 1998a]).

Third, Boolean index expressions serve as a more expressive query and indexing language. They allow the semantic content of queries or documents to be described more precisely, while preserving practical usability. For instance, Boolean index expressions can easily be matched numerically. This is guaranteed by the introduction of a normal form that presents Boolean index expressions in an elementary representation.

The augmentation of index expressions with logical operators also preserves other favourable features of index expressions such as parsebility and the possibility to interpret their meaning linguistically. This ensures that Boolean index expressions, despite their enlarged expressiveness, can still be practically applied for many IR tasks.

We also report on our implementation in the functional language Clean ([Brus et al., 1987]). It illustrates the practical applicability of Boolean index expressions. Functional languages are highly suitable for fast prototyping, support case distinction by pattern recognition, and allow auxiliary functions to be readily designed and incorporated. The ready-to-use code for all issues discussed in this article is provided in one of the appendices.

This article is structured as follows. Section 2 introduces Boolean index expressions by their syntax and illustrates their parsing. In section 3, the intended semantics of Boolean index expressions are examined and defined. In section 4, a procedure to bring them into normal form is devised. Section 5 exploits this normal form for defining equivalent Boolean index expressions. In addition, we provide a scheme to design numerical similarity measures exploiting available similarity measures for regular index expressions. Related work is discussed in section 6. Our concrete implementation is provided in appendix C. This article

does not elaborate on aspects of linguistic nature such as extracting Boolean index expressions from text and does not provide large experiments.

2 Boolean Augmentation

The language of *Boolean index expressions* allows Boolean combinations of index expressions that can contain nested logical operators. In this section, their syntax and parsing are described. First, regular index expressions are illustrated.

2.1 Index Expressions for Information Retrieval

Index expressions (see e.g. [Bruza and Weide, 1992, Wondergem et al., , Bruza, 1993]) have proven of great value for Information Retrieval (IR). As simple yet powerful descriptors, they have been applied for query formulation ([Bruza, 1990]), document indexing ([Bruza and Weide, 1991]), matching ([Wondergem et al., 1998b]), and the construction of hyperindices ([Bruza and Weide, 1990]). Index expressions capture some of the linguistic structure of natural languages. At the same time, they are easily parseble and allow for fast matching. In addition, index expressions provide a useful simplification of noun-phrases ([Winograd, 1983]).

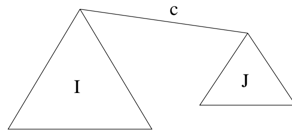

Figure 1: Structure of composed index expression.

Index expressions are inductively defined based on a set of terms and a set of connectors. Index expressions are either single terms or composed index expressions $\mathsf{add}(I, c, J)$, where I and J are index expressions and c is a connector. Figure 1 illustrates the structure of composed index expressions.

In this article, a more intuitive textual representation is used in examples. The composed index expression $\mathsf{add}(\mathsf{augmentation}, \mathsf{with}, \mathsf{logic})$ is textually represented as augmentation with logic. Brackets are only used if necessary to avoid ambiguity. For example, augmentation of (expressions) with (logic) is a different index expression than augmentation of (expressions with (logic)).

2.2 Syntax

The language of Boolean index expressions is given by the following inductive definition.

Definition 2.1

Let T be a set of terms and C be a set of connectors. Boolean index expressions are defined by

1. *if $t \in T$ is a term, it is also a Boolean index expression,*
2. *if I and J are Boolean index expressions and $c \in C$ is a connector, then $\mathsf{add}(I, c, J)$ is a Boolean index expression,*
3. *if I and J are Boolean index expressions, then $I \vee J$, $I \wedge J$, and $\neg I$ are Boolean index expressions as well, and*
4. *no other Boolean index expressions exist.*

□

As shown by definition 2.1, five types of Boolean index expressions exist. Cases 1 and 2 of definition 2.1 generate the the regular language of index expressions which consists of single terms and composed index expressions. The third case of definition 2.1 introduces logical augmentations comprising disjunctions, conjunctions, and negations of Boolean index expressions, respectively. Boolean index expressions form a strict superset of index expressions.

The only additional Boolean index expression, the empty one, denoted by ϵ, cannot occur within non-empty Boolean index expressions. Note that the logical operators may occur nested in Boolean index expressions. This ensures that we do not simply look into logical combinations of index expressions, but that logical operators are first-class citizens resulting in a more compact representation.

The example below provides several instances that illustrate the expressive power of Boolean index expressions.

Example 2.1

The Boolean index in this example are not represented as in definition 2.1 but in a slightly more intuitive fashion.

Using Boolean index expressions, one is able to specify negations of terms, e.g. ¬ apples *and composed index expressions* ¬ (apples with worm). *Furthermore, the negations may occur nested as in* apples with ¬ worm.

In addition, logical combinations of index expressions as walking ∨ cycling *may be readily specified. Furthermore, composed heads, e.g.* (walking ∧ cycling) in Holland, *and composed nested subexpressions, e.g.* walking in (Holland ∨ Belgium) *may occur.*

Logical operators may co-occur leading to more complex Boolean index expressions such as for instance (walking in (Holland ∨ Belgium)) ∨ (cycling through ¬ mountains) □

2.3 Parsing

The abstract grammar given in figure 2 describes the essence of Boolean index expressions. It is a grammar of trees rather than a string grammar. It does not aim at providing a grammar that can be directly used for creating a parser. This is to be done by another concrete grammar (see figure 3), which provides a concrete linear notation. A transducer, i.e., a string-to-string translator, can

be made to connect the two. An advantage of abstract grammars is that they are more suited for semantic processing than concrete grammars.

BoolExpr	$\rightarrow$	ϵ
	$\mid$	Term
	$\mid$	BoolExpr {Connector BoolExpr}*
	$\mid$	BoolExpr $\vee$ BoolExpr
	$\mid$	BoolExpr $\wedge$ BoolExpr
	$\mid$	$\neg$ NBoolExpr
Term	$\rightarrow$	t, t $\in T$
Connector	$\rightarrow$	c, c $\in C$

Figure 2: Abstract Grammar of Boolean Index Expressions

The abstract grammar gives an overview of the different types of Boolean index expressions, denoted by BoolExpr: the empty one, single terms, composed ones, disjunctions, conjunctions, and negations. In the case of composed Boolean index expressions, the expression {Connector BoolExpr}* allows for zero or more subexpressions.

The concrete grammar, as shown in figure 3, is based on the definition of index expressions as presented in [Wondergem et al.,]. The formalisation given there does not allow an empty index expression to occur in a non-empty one. In the concrete grammar, NBoolExpr stands for non-empty Boolean index expression. The primed forms ensure that priorities of the logical operators are correctly parsed. The concrete grammar is only right-recursive and, therefore, implementable in a linear-time parser.

3 Semantics

An important next step is to describe the intended semantics of Boolean index expressions. The semantics of Boolean index expressions that do not contain nested occurrences of the logical operators is provided in section 3.1. In section 3.2, the mutual influence of (logical) operators and (linguistic) connectors is investigated.

3.1 Pure Logical Combinations

The semantics of purely logical combinations of index expressions adheres to the standard semantics of the logical operators. These rules only deal with logical operators, not with their combination with linguistic connectors. That is, we include the standard rules for idempotency, double negations, commutativity, distributivity, DeMorgan, and assocativity. See figure 4 for an overview of these rules.

Example 3.1 provides example Boolean combinations of index expressions and illustrates some applications of the standard rules.

BoolExpr	$\rightarrow$	ϵ
	$\mid$	NBoolExpr
NBoolExpr	$\rightarrow$	NBoolExpr $\vee$ NBoolExpr'
	$\mid$	NBoolExpr'
NBoolExpr'	$\rightarrow$	NBoolExpr' $\wedge$ NBoolExpr"
	$\mid$	NBoolExpr"
NBoolExpr"	$\rightarrow$	$\neg$ IndExpr
	$\mid$	IndExpr
IndExpr	$\rightarrow$	Head {Connector NBoolExpr}*
Head	$\rightarrow$	(NBoolExpr)
	$\mid$	Term
Term	$\rightarrow$	t, t $\in T$
Connector	$\rightarrow$	c, c $\in C$

Figure 3: Concrete Grammar of Boolean Index Expressions

Example 3.1

Note that the examples below do not contain logical operators that are nested within index expressions. The first equivalent pair of Boolean index expressions results from commutativity, the second pair is obtained by distributivity, and the third by DeMorgan's laws.

information $\wedge$ systems
$\equiv$ systems $\wedge$ information

train $\vee$ (transportation on land $\wedge$ rails)
$\equiv$ (train $\vee$ transportation on land) $\wedge$ (train $\vee$ rails)

$\neg$ (fog $\wedge$ pollution by (metals))
$\equiv$ $\neg$ fog $\vee$ $\neg$ pollution by (metals)

□

3.2 Logical Operators vs. Linguistic Connectors

In this section, we investigate the semantics of Boolean index expressions in which logical operators occur nested with respect to linguistic connectors. That is, we investigate the mutual influence of (logical) operators and (linguistic) connectors. First, disjunctions and conjunctions are examined, followed by an investigation of negations.

3.2.1 Disjunction and Conjunction

This section elaborates on the intended semantics of conjunctions and disjunctions in the presence of connectors. We will investigate the semantics of Boolean index expressions in the context of the important IR task of query processing.

$I \wedge I$	$\equiv$	I	idempotent
$I \vee I$	$\equiv$	I	idempotent
$\neg\neg I$	$\equiv$	I	double negation
$I_1 \wedge (I_2 \vee I_3)$	$\equiv$	$(I_1 \wedge I_2) \vee (I_1 \wedge I_3)$	distributivity
$I_1 \vee (I_2 \wedge I_3)$	$\equiv$	$(I_1 \vee I_2) \wedge (I_1 \vee I_3)$	distributivity
$I_1 \wedge I_2$	$\equiv$	$I_2 \wedge I_1$	commutativity
$I_1 \vee I_2$	$\equiv$	$I_2 \vee I_1$	commutativity
$\neg(I_1 \wedge I_2)$	$\equiv$	$\neg(I_1) \vee \neg(I_2)$	DeMorgan
$\neg(I_1 \vee I_2)$	$\equiv$	$\neg(I_1) \wedge \neg(I_2)$	DeMorgan
$I_1 \wedge (I_2 \wedge I_3)$	$\equiv$	$(I_1 \wedge I_2) \wedge I_3$	associativity
$I_1 \vee (I_2 \vee I_3)$	$\equiv$	$(I_1 \vee I_2) \vee I_3$	associativity

Figure 4: Standard logical transformations.

Our point of departure is: suppose a document is characterised by Boolean index expression d and the user query is Boolean index expression q, do d and q match equivalently? Example 3.2 illustrates the interpretation of nested logical operators.

Example 3.2

This example provides two elementary cases of nested logical operators. In the first, the only subexpression consists of a disjunction. In the second, the head is a conjunction.

walking in (Holland ∨ Belgium)
≡ walking in Holland ∨ walking in Belgium

(cycling ∧ hiking) in mountains
≡ cycling in mountains ∧ hiking in mountains

□

Nesting logical operators with connectors may result in ambiguity. This ambiguity occurs when conjunctions and disjunctions co-occur in a Boolean index expression and are connected through at least one connector. The ambiguity involves the order of evaluation, or *priorities*, of the logical operators.

(cycling ∧ hiking) in (Belgium ∨ France)	(Head-∧)
(cycling ∨ hiking) in (Belgium ∧ France)	(Nested-∧)
hiking in (sun ∧ March) to (flat ∨ bar)	(Both nested)

Figure 5: Three minimal examples.

Figure 5 provides examples of the three minimal Boolean index expressions with a conjunction and a disjunction that are connected through at least one connector. The first two index expressions contain the logical operators at different depths, whereas in the last, the logical operators reside at the same depth. The minimal cases will be examined in detail below.

Head-∧.

Figure 6 provides the two possible interpretations of the first minimal example, Head-∧. In order to obtain a well-defined semantics, one of these possibilities should be chosen as the correct interpretation. The key issue here is whether the disjunction is to be evaluated first, as denoted by ∨-first, or the conjunction, denoted by ∧-first. In other words, how are the priorities of the operators defined?

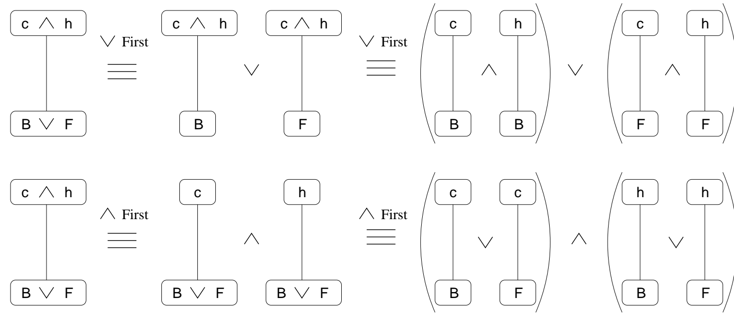

Figure 6: Two interpretations of the first example.

Returning to the query processing point of view, suppose the query is (cycling ∧ hiking) in (Belgium ∨ France) and the document under consideration is characterised by cycling in France ∧ hiking in Belgium. According to the case ∨-first, the two do not match equivalently. That is, the document is not relevant to the query. On the contrary, case ∧-first considers the query and document characterisation equivalent.

We believe the linguistically most natural interpretation of Head-∧ is represented by *(cycling and hiking) in either Belgium or France*, stating both activities should take place together in at least one of the mentioned countries. The reason for this is the intuition stating that, within the query, cycling and hiking are connected stronger than Belgium or France. This, in turn, results from the idea that disjunctions have lower priority than conjunctions. Note that this conforms to the priorities in case of purely logical combinations.

Nested-∧.

Arguments similar to the those for Head-∧ lead to the same conclusion for Nested-∧: the disjunction should have lower priority than the conjunction. In other words, the disjunction should be evaluated first. The combination

of Head-∧ and Nested-∧ seems to indicate that the depth of the operators is irrelevant for their evaluation order.

Both nested.

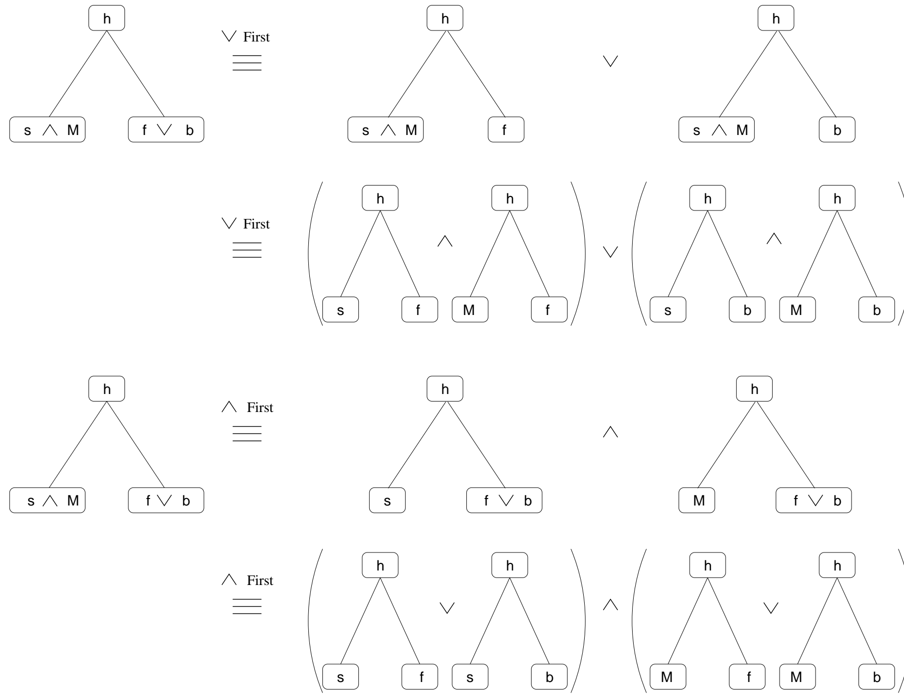

Figure 7: Two interpretations of third example.

See figure 7 for a schematic overview of Both nested. Again, we argue for the case ∨-first in which the disjunction is evaluated first.

To illustrate this, consider the query hiking in (sun ∧ March) to (home ∨ bar) and a document characterisation hiking in March to home ∧ hiking in sun to bar. Evaluating the disjunction first leads to invalidity of equivalence whereas the second case of figure 7, where the conjunction is evaluated first, states both Boolean index expressions are equivalent. Intuitively, the example Boolean index expressions should not be equivalent since hiking in (sun ∧ March) is considered to be the main concept which can be augmented with either one of the directions to home or to bar.

The priorities for disjunctions and conjunctions in co-existence of connectors should be applied on a general level. This is illustrated by figures 8 and 9. Thus disjunctions should *always* be evaluated before conjunctions.

Consider the example of figure 8. The Boolean index expressions that are obtained after steps ∨-first or ∧-first are semantically different, even though in the

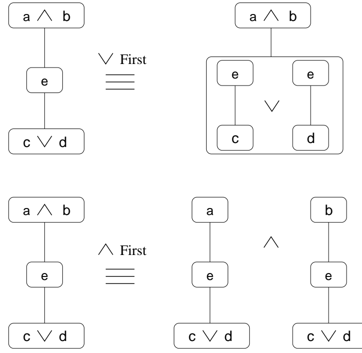

Figure 8: Conjunction and disjunction with additional connector.

initial expression there seems to be no direct interference between the disjunction and the conjunction. However, the first evaluation affects the semantics permanently since the order of evaluation influences the binding of terms to operators.

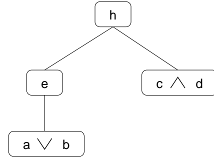

Figure 9: Nested disjunction.

In addition, consider the Boolean index expression of figure 9 where, again, no direct interference seems to occur. However, the order of evaluation of the logical operators again leads to different semantics. The issue here is that if the disjunction is evaluated first, it moves up in the Boolean index expression and causes a case similar to the third minimal example.

To conclude, our intuition states that in the co-existence of connectors, conjunctions bind stronger than disjunctions. In other words, disjunctions have lower priority than conjunctions. Note that this conforms to proposition logic and to the concrete grammar. As observed, disjunctions should always be evaluated before evaluating conjunctions.

This means that no simple rules can be given that, without additional coordination amongst them, describe correct equivalences. Therefore, we will provide a different way to describe equivalent Boolean index expressions. This is the topic of section 4. First, we investigate the influence of negations on the semantics of Boolean index expressions.

3.2.2 Negation

We now investigate the semantics of negations in co-existence of connectors. The grammar of Boolean index expressions allows for four variants of negations in cooking for singles:

¬ (cooking	for	singles)
¬ (cooking)	for	singles
cooking	for	¬ singles
¬ (cooking)	for	¬ singles

Our intuition states that, considered as a query, the first Boolean index expression is satisfied with every document that is not about cooking for singles. The second is satisfied by anything that is for singles, exept cooking. The third is satisfied by documents about cooking for others than singles. Finally, the fourth is satisfied by documents that describe something other than cooking *for* something else than singles.

The above examples show that the combination of negations with connectors does not define patterns of equivalent Boolean index expressions. That is, no normalisations can take place that consist of lifting a negation over a connector. This means that negations can only be normalised in the context of other logical operators, which is done via the standard logical interpretation of negation as aimed at in section 3.1.

4 Normal Forms

The construction of normal forms for Boolean index expressions is done in two stages. First, the disjunctions and conjunctions within the Boolean index expression are *zipped up*, pushed upward. Of course, this process has to conform to the intended semantics as described in the previous section.

After zipping the logical operators, their intended semantics have been evaluated. In the second stage, the zipped Boolean index expression can be brought into any normal form. This can be done by procedures known from normal forms of, for instance, propositional logic. For example, a conjunctive normal form can readily be obtained by the procedure given in appendix A.

The rest of this section focusses on the first stage, i.e., that of zipping logical operators in Boolean index expressions.

4.1 Zipping Logical Operators

We provide a procedure for zipping logical operators that results in a disjunction of conjunctions of so called *atomic* Boolean index expressions. In other words, the way in which we zip the logical operators ensures that the result is in disjunctive normal form modulo atomic Boolean index expressions.

Atomic Boolean index expression do not contain disjunctions and conjunctions. However, negations may appear in atomic Boolean index expressions. Atomic Boolean index expressions can be seen as (logical) atoms in a formula.

More formally, we say a Boolean index expressions is in disjunctive normal form (DNF) iff it is expressed as a sum of products:

$$\mathsf{DNF} : \bigvee_{i=1}^{k} (\bigwedge_{j=1}^{l_i} I_{i,j})$$

where $I_{i,j}$ are atomic Boolean index expressions.

We provide a constructive description of procedure called Zip for bringing Boolean index expressions into disjunctive normal form. The previous section showed us that disjunctions have lower priority than conjunctions. They, thus, have to be evaluated first:

$$\mathsf{Zip}(I) = \mathsf{ZipAnd}(\mathsf{ZipOr}(I))$$

The function Zip returns its argument in zipped form. It first zips all disjunctions by calling function ZipOr. This function is defined inductively as follows:

$$\begin{array}{lcl}
\mathsf{ZipOr}(t) & = & t \\
\mathsf{ZipOr}(\mathsf{add}(I, c, J)) & = & \mathsf{OrCons}(\mathsf{ZipOr}(I), c, \mathsf{ZipOr}(J)) \\
\mathsf{ZipOr}(I \vee J) & = & \mathsf{ZipOr}(I) \vee \mathsf{ZipOr}(J) \\
\mathsf{ZipOr}(I \wedge J) & = & \mathsf{OrProd}(\mathsf{ZipOr}(I), \mathsf{ZipOr}(J)) \\
\mathsf{ZipOr}(\neg I) & = & \mathsf{DNF}(\mathsf{NegProd}(\mathsf{ZipAnd}(\mathsf{ZipOr}(I))))
\end{array}$$

Terms do not contain disjunctions and therefore cannot be zipped. In the contructor case $\mathsf{add}(I, c, J)$, disjunctions in both head I and subexpression J are zipped first, resulting in disjunctive representations $\mathsf{ZipOr}(I)$ and $\mathsf{ZipOr}(J)$. From these, an overall disjunction is constructed by OrCons:

$$\mathsf{OrCons}(\bigvee_{1 \le i \le k} I_i,\ c,\ \bigvee_{1 \le j \le l} J_j) = \bigvee_{1 \le i \le k,\ 1 \le j \le l} \mathsf{add}(I_i, c, J_j)$$

In the case of a disjunction $I \vee J$, the disjunctive representation consists of the disjunction of the disjunctive representations $\mathsf{ZipOr}(I)$ and $\mathsf{ZipOr}(J)$. Another auxiliary routine, OrProd is used in the case of a conjunction $I \wedge J$ to ensure that a disjunctive result is produced.

$$\mathsf{OrProd}(\bigvee_{1 \le i \le k} I_i,\ \bigvee_{1 \le j \le l} J_j) = \bigvee_{1 \le i \le k,\ 1 \le j \le l} (I_i \wedge J_j)$$

In the case of a negation $\neg I$ more work has to be done. In I, disjunctions are zipped first, followed by zipping conjunctions. The function that zips conjunctions, ZipAnd, is given below. Zipping conjunctions as well, here, is necessary since the function that is applied next, NegProd, assumes its argument is in completely zipped form. This function, which is provided in figure 10, ensures the negation is taken into account properly. Finally, the zipped Boolean index expression is brought into disjunctive normal form by a call to DNF since the result of ZipOr is to be a disjunction. This property is assumed by OrCons. For a description of DNF we kindly refer the reader to appendix B.

The second fase in zipping logical operators is taken care of by function ZipAnd which zips all remaining conjunctions.

$$\begin{array}{lcl} \mathsf{NegProd}(t) & = & \neg t \\ \mathsf{NegProd}(\mathsf{add}(I,c,J)) & = & \neg\mathsf{add}(I,c,J) \\ \mathsf{NegProd}(I \vee J) & = & \mathsf{NegProd}(I) \wedge \mathsf{NegProd}(J) \\ \mathsf{NegProd}(I \wedge J) & = & \mathsf{NegProd}(I) \vee \mathsf{NegProd}(J) \\ \mathsf{NegProd}(\neg I) & = & I \end{array}$$

Figure 10: Function NegProd

$$\begin{array}{lcl} \mathsf{ZipAnd}(t) & = & t \\ \mathsf{ZipAnd}(\mathsf{add}(I,c,J)) & = & \mathsf{AndCons}(\mathsf{ZipAnd}(I), c, \mathsf{ZipAnd}(J)) \\ \mathsf{ZipAnd}(I \vee J) & = & \mathsf{ZipAnd}(I) \vee \mathsf{ZipAnd}(J) \\ \mathsf{ZipAnd}(I \wedge J) & = & \mathsf{ZipAnd}(I) \wedge \mathsf{ZipAnd}(J) \\ \mathsf{ZipAnd}(\neg I) & = & \neg I \end{array}$$

Similar to ZipOr, the result of zipping a term is that term itself. In the constructor case $\mathsf{add}(I,c,J)$, all conjunctions in I and J are zipped first by recursive calls to ZipAnd. After that, the function AndCons constructs the final conjunction:

$$\mathsf{AndCons}(\bigwedge_{1\leq i\leq k} I_i,\ c,\ \bigwedge_{1\leq j\leq l} J_j) = \bigwedge_{1\leq i\leq k,\ 1\leq j\leq l} \mathsf{add}(I_i, c, J_j)$$

In the case of a disjunction $I \vee J$, conjunctions in I and J are to be zipped locally. That is, the disjunctive structure is not to be altered. Rather, only in the non-disjunctive elements of the disjunction may conjunctions be zipped. Therefore, the ZipAnds are pushed downward without altering the disjunctive structure.

Conjunctions $I \wedge J$, are rewritten to the conjunction of the zipped versions of I and J.

Since all disjunctions and conjunctions are already zipped in the first stage by ZipOr, only negated atomic Boolean index expressions can occur for ZipAnd. Thus, in the case of a negation $\neg I$, ZipAnd may simply return its argument.

5 Matching

Normal forms ease certain computations on Boolean index expressions. In this section we show that, exploiting the disjunctive normal form, equivalence of Boolean index expressions and numerical similarity measures can easily be specified.

5.1 Equivalence of Boolean Index Expressions

We define two index expressions to be equivalent iff their normal forms are equal modulo the order of the atomic Boolean index expressions that constitute it.

Definition 5.1

Two Boolean index expressions I and J are equivalent, denoted $I \equiv J$, iff $\mathsf{Zip}(I) \doteq \mathsf{Zip}(J)$. □

The relation $\doteq$ only has to take associativity of disjunctions and conjunctions into account since both arguments are in disjunctive normal form. The relation $\doteq$ is specified in a functional programming language in appendix C.

Example 5.1

The Boolean index expressions of examples 3.1 and 3.2 provide instances of the equivalence relation. □

5.2 Similarity Measures

The fact that we can bring arbitrary Boolean index expressions into normal form allows us to exploit matching functions for normal index expressions with only minor modifications.

The availability of numerical similarity measures for Boolean index expressions enables their use in many important IR tasks. For instance, IR tasks that involve similarity measures are document ranking, classification, routing, and clustering.

The DNF normal form delivers a (logical) sum of products of atomic index expressions. The sum and product are directly translated to mathematical sum and product and do not involve any particular matching function. This implies that only atomic index expressions have to be directly compared, meaning that similarity measures for regular index expressions only have to be extended for the case of negations.

The new similarity measure for Boolean index expressions, denoted by $\mathsf{sim}_{\mathsf{Bool}}$, requiring its arguments are in disjunctive normal form, is defined as

$$\mathsf{sim}_{\mathsf{Bool}} = \mathsf{sim}_{\vee}(\mathsf{Zip}(I), \mathsf{Zip}(J))$$

We first exploit the structure of DNF by dealing with the disjunctions $\mathsf{Zip}(I) = \bigvee_{i=1}^{k} I_i$ and $\mathsf{Zip}(J) = \bigvee_{j=1}^{l} J_j$.

$$\mathsf{sim}_{\vee}(I, J) = \Sigma_{i=1}^{k} \Sigma_{j=1}^{l} \mathsf{sim}_{\wedge}(I_i, J_j)$$

Next, observe that the arguments of $\mathsf{sim}_{\wedge}(I, J)$ are conjunctions $I = \bigwedge_{i=1}^{k} I_i$ and $J = \bigwedge_{j=1}^{l} J_j$ of atomic Boolean index expressions. This implies that we can compute their similarity as a double product of elementary similarities for atomic index expressions. We denote the similarity measure for regular index expressions by sim.

$$\mathsf{sim}_{\wedge}(I, J) = \Pi_{i=1}^{k} \Pi_{j=1}^{l} \mathsf{sim}(I_i, J_j)$$

The functions $\mathsf{sim}_{\vee}$ and $\mathsf{sim}_{\wedge}$ cater for disjunctions and conjunctions, respectively. Therefore, the basic similarity measure for index expressions sim only

$$\mathsf{sim}(\neg I, J) = 1 - \mathsf{sim}(I, J)$$

$$\mathsf{sim}(I, \neg J) = 1 - \mathsf{sim}(I, J)$$

Figure 11: Additional rules for negations.

needs to be augmented for negations. We do this by adding two simple rules as provided in figure 11.

Several ways of matching index expressions are described. See for instance [Wouda, 1997, Berger, 1998] for techniques based on flattening index expressions. Since these measures are not based on case distinction by pattern recognition, they are not directly applicable in this context.

Suitable matching functions for index expressions were described and analysed in [Wondergem et al., 1998b]. All of these can be transformed into similarity measures for Boolean index expressions by the process described below for the example measure. As an example, we examine the embedded-content measure in the next section.

5.3 Embedded-Content

The *embedded-content* similarity measure for index expressions is shown in figure 12. There, $\mathsf{sim}_T : T \times T \to [0..1]$ delivers the similarity of terms and $\mathsf{sim}_C : C \times C \to [0..1]$ does the same for connectors. The resulting similarity function sim delivers the (maximal) degree of embedding of its leftmost argument in its rightmost argument. It takes the order of subexpressions into account.

We will briefly describe the workings of this similarity measure by explaining what is meant by embedding. The first case of the algorithm, i.e., Terms, states that term similarity gives their degree of embedding. The case Term-Comp states that the degree of embedding of a term t in a composed index expression $\mathsf{add}(I, c, J)$ is equal to the largest similarity of t with any term from $\mathsf{add}(I, c, J)$. The case Composed deals with the embedment of composed index expressions. The maximum of three subcases is computed: (left): $\mathsf{add}(K, d, L)$ is completely embedded in K, (right): the same for L, or (spreaded): the leftmost subexpressions I and K are embedded, connectors c and c are equal, and the rightmost subexpressions J and L are embedded. The last case, Comp-Term, is based on the idea that a composed index expressions can never be fully embedded in a single term. It gives a penalty relative to the size of the composed index expression.

The workings of the complete embedded-content similarity measure for Boolean index expressions are illustrated in example 5.2.

Example 5.2

The similarity between I = surfing on Internet *and* J = surfing in ($\neg$

$$\begin{array}{ll} & \mathsf{sim}(t, t') \qquad \text{(Terms)} \\ = & \mathsf{sim}_T(t, t') \end{array}$$

$$\begin{array}{ll} & \mathsf{sim}(t, \mathsf{add}(I, c, J)) \qquad \text{(Term-Comp)} \\ = & \mathtt{max}\{\mathsf{sim}(I, t), \mathsf{sim}(J, t)\} \end{array}$$

$$\begin{array}{lll} & \mathsf{sim}(\mathsf{add}(I, c, J), \mathsf{add}(K, d, L)) & \text{(Comp)} \\ = & \mathtt{max}\{\mathsf{sim}(\mathsf{add}(I, c, J), K), & \text{left} \\ & \quad \mathsf{sim}(\mathsf{add}(I, c, J), L), & \text{right} \\ & \quad \mathsf{sim}(I, K) \times \mathsf{sim}_C(c, d) \times \mathsf{sim}(J, L)\} & \text{spreaded} \end{array}$$

$$\begin{array}{ll} & \mathsf{sim}(\mathsf{add}(I, c, J), t) \qquad \text{(Comp-Term)} \\ = & \frac{\mathsf{sim}_T(\mathsf{Head}(I), t)}{|\mathsf{Terms}(\mathsf{add}(I, c, J))|} \end{array}$$

Figure 12: Embedded-Content for Index Expressions

Holland) $\wedge$ surfing on WWW, *according to the embedded-content measure is* 0.8.

This is computed as follows. Assume a similarity between connectors in *and* on *of* 0.8 *and a similarity of* 0.9 *between* Internet *and* WWW. *For all other terms, assume their similarity is their binary string equality.*

Since no disjunctions exist, we directly focus on the case of conjunctions. There, $\mathsf{sim}_\wedge(I, J) =$ sim(surfing on Internet, surfing in ($\neg$ Holland)) $\times$ sim(surfing on Internet, surfing on WWW). *The first part, resulting in a similarity of* 0.8, *is covered by case* Comp *of the algorithm. There, the maximal value is obtained for the* spreaded *subcase:* surfing *is maximally embedded in* surfing, *the connectors have a similarity of* 0.8 *and* sim($\neg$Holland, Internet) $= 1 -$ sim(Holland, Internet) $= 1 - 0 = 1$. *For the second part, the* spreaded *subcase of* Comp *delivers the similarity of* 0.9. *The resulting similarity value therefore is* $0.8 \times 0.9 = 0.72$. □

We have explained a simple mechanism to apply matching functions for index expressions for Boolean ones with only minor changes. Although the given mechanism is simple, it results in easily computable similarity measures and clearly shows the essence of adapting similarity measures for Boolean index expressions.

6 Related Work

This section describes related approaches to our work. Coordinated concepts and description logics involve structured queries and logical combinations thereof. However, whereas these techniques roughly aim at modeling relations between

concepts, our approach deals with linguistic contents and meaning. This shows, for example, in their need for ontologies.

Another, largely orthogonal approach is that of Extended Boolean retrieval. This approach does not consider structure within the elements with which logical combinations are formed. Thus, we are into new ground with Boolean index expressions.

6.1 Coordinated Concepts

In [Vet and Mars, 1998, Vet and Mars, 1996], van der Vet and Mars describe a descriptor language consisting of Boolean combination of concepts. Concepts, which are defined by a structured ontology, can either be simple (terms) or coordinated. Coordinated concepts are constructed by relating two or more concepts through a coordinator (relation). As an example, consider `cures(aspirin, headache)`, where the coordinator `cures` disambiguates the relation between `aspirin` and `headache`.

Their main concern is to raise precision while maintaining recall. A number of differences with our approach can readily be observed. In stead of concepts, we work with structured descriptors (index expressions). Observe that the above coordinated example could be described by the index expression aspirin as cure for headache. In addition, the Boolean operators in the coordinated concepts approach cannot be nested within concepts but define a layer on top of them. Finally, their approach makes use of an ontology, putting demands on user skills (experts), restricted domains, and construction and maintenance.

6.2 Description Logic

In [Meghini et al., 1993], Meghini *et al* describe the use of a *terminological logic* or *description logic* for IR. Terminological logics are specially designed to deal with complex terms, the formal equivalent of natural language noun phrases. Those complex terms can express the structure, layout, and semantic content of documents and queries.

In essence, this approach shows great resemblance to the abovementioned approach with coordinated concepts. A major difference, as mentioned in [Vet and Mars, 1996], is supposedly the query processing time of systems based on description logics:

> [their] performance on large concept hierarchies turned out to be insufficiently for deployment in real-life IR systems.

However, in [Meghini et al., 1993] the authors claim that most of the work can be done at construction time of the knowledge base,

> thus reducing the query-time theorem proving operation to table-lookup.

Next to the problems mentioned for the previously described approach, an obvious problem with description logics is how to write automatic indexing procedures that produce document or request representations in terms of them.

6.3 Extended Boolean IR

In [Salton et al., 1983], Salton *et al* propose the *extended Boolean IR* model, which represents a generalisation of both structured queries in Boolean retrieval where the logical connectives have a strict interpretation, and the vector space model, where no structure exists in queries. This is done by attaching a weight p to the standard Boolean operators and applying the theory of vector p-norms. Value $p = 1$ results in the vector space model and $p = \infty$ leads to standard Boolean retrieval. The resulting extended Boolean IR model combines favourable aspects of Boolean retrieval (structured queries) and the vector space model (ranking, term weighting, and output size control).

Their approach is largely orthogonal to ours, since they do not consider structure within the atoms with which logical combinations are formed. It would be interesting to develop Boolean index expressions with the "soft" logical operators obtained by including the p-norms. For instance, differently valued p's may serve as depth-factors. In order to accomplish this, terms with a higher depth-factor can be assigned lower p values.

7 Conclusions

This article presented Boolean index expressions for Information Retrieval as an augmentation of index expressions with logical operators. In this way, Boolean logic and some of the structure of natural languages are combined in one expressive descriptor language. We defined the syntax of Boolean index expressions, analysed their intended semantics, and provided a mechanism to bring them into disjunctive normal form. We exploited this normal form to enhance practical applicability of Boolean index expressions, as illustrated by an equivalence function and a numerical similarity measure. Throughout this article, we provided an implementation of Boolean index expressions in a functional language. We also compared our work to other approaches that elaborate on logical descriptor languages for IR.

Further research can be directed towards a comparison with noun-phrases. For instance, it would be interesting to see if the difficulties encountered when using noun phrases for query formulation and document indexing also hold for Boolean index expressions, which are less complex than noun phrases.

Furthermore, the use of Boolean index expressions for efficient information filtering is of interest. The logical operators compress the representation of expressions, which can be exploited to minimize the number of needed similarity computations. To illustrate the compactness of representation, consider a Boolean index expression consisting of k logical operators and n keywords. This can be described by a logical combination of atomic index expressions, but one would need 2^k atomic expressions of at most $n - k$ terms each. Research will have to be done into which (normal) form of Boolean index expressions suits best.

In addition, the construction and maintenance of user profiles based on

Boolean index expressions may be looked into in the context of user relevance feedback. Positive feedback might be incorporated by conjunctions or disjunctions, and negative feedback by negations.

Further research may also be devoted to develop other similarity measures for Boolean index expressions. When that has been done, experimental or theoretical evaluation of the different measures will also be appropriate.

A Conjunctive Normal Form

The expression $\mathsf{CNF}(I)$ provides the conjunctive normal form of Boolean index expression I. It must be remarked that the procedure below requires its argument to be in zipped form.

$$
\begin{aligned}
\mathsf{CNF}(t) &= t \\
\mathsf{CNF}(\mathsf{add}(I,c,J)) &= \mathsf{add}(I,c,J) \\
\mathsf{CNF}(I \vee J) &= \mathsf{AndProd}(\mathsf{CNF}(I), \mathsf{CNF}(J)) \\
\mathsf{CNF}(I \wedge J) &= \mathsf{CNF}(I) \wedge \mathsf{CNF}(J) \\
\mathsf{CNF}(\neg I) &= \mathsf{NegCNF}(I) \\
\\
\mathsf{NegCNF}(t) &= \neg t \\
\mathsf{NegCNF}(\mathsf{add}(I,c,J)) &= \neg\mathsf{add}(I,c,J) \\
\mathsf{NegCNF}(I \vee J) &= \mathsf{NegCNF}(I) \wedge \mathsf{NegCNF}(J) \\
\mathsf{NegCNF}(I \wedge J) &= \mathsf{AndProd}(\mathsf{NegCNF}(I), \mathsf{NegCNF}(J)) \\
\mathsf{NegCNF}(\neg I) &= \mathsf{CNF}(I)
\end{aligned}
$$

B Disjunctive Normal Form

The expression $\mathsf{DNF}(I)$ provides the disjunctive normal form of Boolean index expression I. Again, the procedure below requires its argument to be in zipped form.

$$
\begin{aligned}
\mathsf{DNF}(t) &= t \\
\mathsf{DNF}(\mathsf{add}(I,c,J)) &= \mathsf{add}(I,c,J) \\
\mathsf{DNF}(I \vee J) &= \mathsf{DNF}(I) \vee \mathsf{DNF}(J) \\
\mathsf{DNF}(I \wedge J) &= \mathsf{OrProd}(\mathsf{DNF}(I), \mathsf{DNF}(J)) \\
\mathsf{DNF}(\neg I) &= \mathsf{NegDNF}(I) \\
\\
\mathsf{NegDNF}(t) &= \neg t \\
\mathsf{NegDNF}(\mathsf{add}(I,c,J)) &= \neg\mathsf{add}(I,c,J) \\
\mathsf{NegDNF}(I \vee J) &= \mathsf{OrProd}(\mathsf{NegDNF}(I), \mathsf{NegDNF}(J)) \\
\mathsf{NegDNF}(I \wedge J) &= \mathsf{NegDNF}(I) \vee \mathsf{NegDNF}(J) \\
\mathsf{NegDNF}(\neg I) &= \mathsf{DNF}(I)
\end{aligned}
$$

C Implementation of Boolean Index Expressions

This section provides an implementation of Boolean index expressions in the functional programming language Clean (see [Brus et al., 1987]). Boolean index expressions are defined as the algebraic type `BoolExpr`. Constructors identify the different cases of definition 2.1. Terms and connectors are both represented as strings.

```
::BoolExpr = Term String
           | Add (BoolExpr) String (BoolExpr)
           | Dis (BoolExpr) (BoolExpr)
           | Con (BoolExpr) (BoolExpr)
           | Neg (BoolExpr)
```

The equivalence relation, as aimed at in section 5.1, is defined as an instance of the overloaded equivalence operator ==. In this way, generic functions which use the equivalence operator are also defined for Boolean index expressions. For example, the function `isMember` which checks if an element occurs in a list, can now be applied to lists of Boolean index expressions.

```
// Equivalence of Boolean Index Expressions
instance == BoolExpr
where
  (==) :: !BoolExpr !BoolExpr -> Bool
  (==) (Term s) (Term t)        = s == t
  (==) (Add i c j) (Add k d l)  = i == k && c == d && j == l
  (==) (Dis i j) (Dis k l)      = (i == k && j == l) || (i == l && j == k)
  (==) (Con i j) (Con k l)      = (i == k && j == l) || (i == l && j == k)
  (==) (Neg i) (Neg j)          = i == j
  (==) _ _                      = False
```

The auxiliary functions `Head` and `Terms` that are used in the embedded-content similarity measure are specified below.

```
Head :: BoolExpr -> BoolExpr
Head (Term s)          = Term s
Head (Add i c j)       = Head i
Head i                 = i

Terms :: BoolExpr -> BoolExpr
Terms (Term s)         = [s]
Terms (Add i c j)      = Terms i ++ Terms j
Terms (Dis i j)        = Terms i ++ Terms j
Terms (Con i j)        = Terms i ++ Terms j
Terms (Neg i)          = Terms i
```

The procedure for zipping logical operators, which is given later, uses several auxiliary functions to flatten the structure of Boolean index expressions

(Dis2List and Con2List) and build the structure up again (List2Dis and List2Con).

```
Dis2List :: BoolExpr -> [BoolExpr]
Dis2List (Dis i j)      = Dis2List i ++ Dis2List j
Dis2List i              = [i]

Con2List :: BoolExpr -> [BoolExpr]
Con2List (Con i j)      = Con2List i ++ Con2List j
Con2List i              = [i]

List2Dis :: [BoolExpr] -> BoolExpr
List2Dis [x:[x2:xs]]    = Dis x (List2Dis [x2:xs])
List2Dis [x:[]]         = x

List2Con :: [BoolExpr] -> BoolExpr
List2Con [x:[x2:xs]]    = Con x (List2Con [x2:xs])
List2Con [x:[]]         = x
```

The production and construction functions for disjunctions and conjunctions make use of list comprehensions. The function OrProd, for example, computes a list of conjunctions $a \wedge b$, where a and b are generated from the list of disjunctions from i and j, respectively. The resulting list of disjunctions is formed into a Boolean index expression by List2Dis. The other functions follow a similar line of working.

```
OrProd :: BoolExpr BoolExpr -> BoolExpr
OrProd i j = List2Dis [(Con a b) \\ a <- (Dis2List i), b <- (Dis2List j)]

AndProd :: BoolExpr BoolExpr -> BoolExpr
AndProd i j = List2Con [(Dis a b) \\ a <- (Con2List i), b <- (Con2List j)]


OrCons :: BoolExpr [Char] BoolExpr -> BoolExpr
OrCons i c j = List2Dis [(Add a c b) \\ a <- (Dis2List i), b <- (Dis2List j)]

AndCons :: BoolExpr [Char] BoolExpr -> BoolExpr
AndCons i c j = List2Con [(Add a c b) \\ a <- (Con2List i), b <- (Con2List j)]
```

The function to bring zipped Boolean index expressions into disjunctive normal form is a direct translation of the code in appendix B.

```
DNF :: BoolExpr -> BoolExpr
DNF (Term s)            = Term s
DNF (Add i c j)         = Add i c j
DNF (Dis i j)           = Dis (DNF(i)) (DNF(j))
DNF (Con i j)           = OrProd (DNF(i)) (DNF(j))
```

```
DNF (Neg i)              = NegDNF(i)


NegDNF :: BoolExpr -> BoolExpr
NegDNF (Term s)          = Neg (Term s)
NegDNF (Add i c j)       = Neg (Add i c j)
NegDNF (Dis i j)         = OrProd (NegDNF(i)) (NegDNF(j))
NegDNF (Con i j)         = Dis (NegDNF(i)) (NegDNF(j))
NegDNF (Neg i)           = DNF(i)
```

The last auxiliary function, `NegProd`, also readily follows from the functional specification given in section 4.

```
NegProd :: BoolExpr -> BoolExpr
NegProd (Term t)         = Neg (Term t)
NegProd (Add i c j)      = Neg (Add i c j)
NegProd (Dis i j)        = Con (NegProd i) (NegProd j)
NegProd (Con i j)        = Dis (NegProd i) (NegProd j)
NegProd (Neg i)          = i
```

The actual zipping functions can now be specified.

```
Zip :: BoolExpr -> BoolExpr
Zip i = ZipAnd (ZipOr i)

ZipOr :: BoolExpr -> BoolExpr
ZipOr (Term t)           = Term t
ZipOr (Add i c j)        = OrCons (ZipOr i) c (ZipOr j)
ZipOr (Dis i j)          = Dis (ZipOr i) (ZipOr j)
ZipOr (Con i j)          = OrProd (ZipOr i) (ZipOr j)
ZipOr (Neg i)            = DNF (NegProd (ZipAnd (ZipOr i)))

ZipAnd :: BoolExpr -> BoolExpr
ZipAnd (Term t)          = Term t
ZipAnd (Add i c j)       = AndCons (ZipAnd i) c (ZipAnd j)
ZipAnd (Dis i j)         = AndProd (ZipAnd i) (ZipAnd j)
ZipAnd (Con i j)         = Con (ZipAnd i) (ZipAnd j)
ZipAnd (Neg i)           = NegProd (ZipAnd i)
```

The embedded-content similarity function `sim`, as described in section 5, is readily translated into the Clean syntax. Note that the extra lines for negations and the type definitions for the primitive similarity functions for terms `TermSim` and connectors `ConnSim` are included. The primitive similarity measures can be implemented in numerous ways.

```
sim :: BoolExpr BoolExpr -> Real
sim (Term s) (Term t)
```

```
        = TermSim (Term s) (Term t)

sim (Term s) (Add i c j)
        = maxList (map (TermSim (Term s)) (Terms (Add i c j)))
        / toReal (length (Terms (Add i c j)))

sim (Add i c j) (Term s)
        = maxList (map (TermSim (Term s)) (Terms (Add i c j)))

sim (Add i c j) (Add k d l)
        = maxList [sim i k * ConnSim c d * sim j l
                  ,sim i (Add k d l)
                  ,sim j (Add k d l)
                  ]

sim (Neg i) j
        = 1.0 - (sim i j)
sim i (Neg j)
        = 1.0 - (sim i j)

TermSim :: BoolExpr BoolExpr -> Real

ConnSim :: String String -> Real
```

The total similarity function `SimBool` for Boolean index expressions can now be specified.

```
SimBool :: BoolExpr BoolExpr -> Real
SimBool i j = SimOr (Zip i) (Zip j)

SimOr :: BoolExpr BoolExpr -> Real
SimOr i j  = sum  [ SimAnd x y \\ x <- (Dis2List i), y <- (Dis2List j)]

SimAnd :: BoolExpr BoolExpr -> Real
SimAnd i j = prod [ sim x y \\ x <- (Con2List i), y <- (Con2List j)]
```

References

[Berger, 1998] Berger, F. (1998). *Navigational Query Construction in a Hypertext Environment.* PhD thesis, Department of Computer Science, University of Nijmegen.

[Brus et al., 1987] Brus, T., Eekelen, M. v., Leer, M. v., and Plasmeijer, M. (1987). Clean - A Language for Functional Graph Rewriting. In *Proceedings of the Third International Conference for Functional Programming Languages and Computer Architectures (FPCA '87)*, volume 274 of *Lecture Notes in Computer Science*, pages 364–384, Portland, Oregon, USA. Springer-Verlag.

[Bruza, 1990] Bruza, P. (1990). Hyperindices: A Novel Aid for Searching in Hypermedia. In Rizk, A., Streitz, N., and Andre, J., editors, *Proceedings of the European Conference on Hypertext - ECHT 90*, pages 109–122, Cambridge, United Kingdom. Cambridge University Press.

[Bruza, 1993] Bruza, P. (1993). *Stratified Information Disclosure: A Synthesis between Information Retrieval and Hypermedia.* PhD thesis, University of Nijmegen, Nijmegen, The Netherlands.

[Bruza and Weide, 1990] Bruza, P. and Weide, T. v. d. (1990). Two Level Hypermedia - An Improved Architecture for Hypertext. In Tjoa, A. and Wagner, R., editors, *Proceedings of the Data Base and Expert System Applications Conference (DEXA 90)*, pages 76–83, Vienna, Austria. Springer-Verlag.

[Bruza and Weide, 1991] Bruza, P. and Weide, T. v. d. (1991). The Modelling and Retrieval of Documents using Index Expressions. *ACM SIGIR FORUM (Refereed Section)*, 25(2).

[Bruza and Weide, 1992] Bruza, P. and Weide, T. v. d. (1992). Stratified Hypermedia Structures for Information Disclosure. *The Computer Journal*, 35(3):208–220.

[Meghini et al., 1993] Meghini, C., Sebastiani, F., Straccia, U., and Thanos, C. (1993). A Model of Information Retrieval based on a Terminological Logic. In *Proceedings of the 19th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 298–307, Pittsburgh, PA, USA.

[Salton et al., 1983] Salton, G., Fox, E., and WU, H. (1983). Extended Boolean information retrieval. *Communications of the ACM*, 26(12):1022–1036.

[Vet and Mars, 1996] Vet, P. v. d. and Mars, N. (1996). Coordination Recovered. In van der Meer, K., editor, *Informatiewetenschap 1996*, pages 139–151, Delft, The Netherlands.

[Vet and Mars, 1998] Vet, P. v. d. and Mars, N. (1998). Bottom-Up Construction of Ontologies. *IEEE Transactions on Knowledge and Data Engineering*, 10(4):513–526.

[Winograd, 1983] Winograd, W. (1983). *Language as a Cognitive Process.* Addison-Wesley Pub. Co., Reading MA, USA.

[Wondergem et al.,] Wondergem, B., Bommel, P. v., and Weide, T. v. d. Nesting and Defoliation of Index Expressions for Information Retrieval. *Knowledge and Information Systems.* To appear.

[Wondergem et al., 1998a] Wondergem, B., Bommel, P. v., and Weide, T. v. d. (1998a). Cumulative Duality in Designing Information Brokers. In *Proceedings of the 9th International Conference on Database and Expert Systems Applications (DEXA)*, Vienna, Austria.

[Wondergem et al., 1998b] Wondergem, B., Bommel, P. v., and Weide, T. v. d. (1998b). Matching Index Expressions for Information Retrieval. Technical Report CSI-R9826, University of Nijmegen, Nijmegen, The Netherlands.

[Wouda, 1997] Wouda, P. (1997). Similarity between Index Expressions. Master's thesis, University of Nijmegen, Nijmegen, The Netherladnds.