Automating the Computation of Nested Abnormality Theories

Zhendong Patrick Su

Advised by

Professor Vladimir Lifschitz

Read by

Professor Robert Boyer

Professor Alan Cline

Abstract

Nested Abnormality Theories (NATs) were introduced by Lifschitz to use circumscription for representing knowledge. We describe a program CS (Circumscription Simplifier) for computing NATs. The main part of CS is SCAN, developed by Engel and Ohlbach for the elimination of second-order quantifiers. Our addition to SCAN is a new interface that allows us to use it to simplify NATs. CS takes a NAT as input and if it terminates, the output is a set of formulas whose conjunction is logically equivalent to the NAT we used as input. In many cases, the resulting formulas are first-order.

1 Introduction

To formalize common sense remains a major challenge to the logical approach to Artificial Intelligence. In attempting to solve this problem, several non-monotonic formalisms emerged including circumscription. Circumscription, proposed by McCarthy, is a syntactic transformation of formulas to formalize the kind of conjectural reasoning used often by humans: The objects that they can determine to have certain properties or relations are the only objects that do. The methodology for representing defaults developed by McCarthy [1986] involves the use of an "abnormality predicate" and the application of circumscription to minimize its extent. McCarthy explored several possible strategies to perform the circumscription of abnormality, but none of them turned out to be completely satisfactory.

Lifschitz [1995] proposed a new approach to the use of circumscription for representing knowledge. This new framework is called Nested Abnormality Theories (NATs). The main feature of this formalization is that it may have a nested structure. Each level of nesting gives a "block" that corresponds to one application of the circumscription operator. As a result, the circumscriptions involved become rather simple. In many examples, the second order quantifiers can be eliminated using the existing techniques.

Two methods for computing circumscriptions exist. One of them is a generalization of predicate completion [Lifschitz, 1993]. The other is based on the SCAN algorithm [Gabbay and Ohlbach, 1992]. In this paper, we focus on the application of the SCAN algorithm to computing circumscription. An implementation of the algorithm developed by Engel and Ohlbach is based on the OTTER theorem prover [McCune, 1990] developed at Argonne National Laboratory. We have written a program called CS which uses the SCAN program to automatically compute a set of first-order formulas

that are equivalent to a given NAT. It has worked successfully on several examples. In the next few sections, we'll review some of the syntax and semantics of NATs and introduce the CS program. Also, directions of future work will be discussed.

2 Using Circumscription for Representing Knowledge

2.1 Why Non-monotonic Reasoning

Representing common sense knowledge is of major concern to AI researchers. Classical logic is insufficient to solve this problem. It is monotonic in the following sense: As the set of beliefs grows, so does the set of conclusions that can be drawn from those beliefs. While much human reasoning is monotonic, some important human common sense reasoning is not. We reach conclusions from certain premises that we would not reach if certain other sentences were included in our premises. For example, if you were told that "Birds fly and Tweety is a bird," then you can conclude that "Tweety flies". But if you were told, in addition, that "Tweety is dead", then you couldn't conclude that he flies this time. The inference here is non-monotonic. Without the knowledge that Tweety is an atypical bird, you assume that it's typical. But once you learn something new about Tweety (for instance, that he is dead), you will need to revise your conclusion that he flies. To formalize this kind of reasoning, several formalisms have emerged, including circumscription.

2.2 Circumscription

Circumscription was introduced by McCarthy [1980, 1986]. The main idea of circumscription is to consider only the models of a given axiom set that satisfy a certain *minimality condition*, rather than to consider arbitrary models of the axioms. Mathematically, circumscription is defined as a syntactic transformation of logical formulae. It transforms a sentence A to a stronger sentence A^*, such that the models of A^* are precisely the minimal models of A.

Here is the formal definition of circumscription taken from [Lifschitz, 1993]. Let $A(P, Z_1, \ldots, Z_m)$ be a sentence containing a predicate constant P and object, function, and/or predicate constants $Z_1, \ldots, Z_m$ (and possibly other object, function, and predicate constants). The *circumscription of P in A with varied $Z_1, \ldots, Z_m$* is the sentence

$$A(P, Z_1, \ldots, Z_m) \wedge \neg \exists p z_1, \ldots, z_m [A(p, z_1, \ldots, z_m) \wedge p < P],$$

where

$$p < P$$

in the formula stands for

$$\forall x[p(x) \supset P(x))] \wedge \neg \forall x[p(x) \equiv P(x)].$$

Here p is a predicate variable of the same arity as P; if Z_i is an object constant, then z_i is an object variable, and if Z_i is a function/predicate constant, then z_i is a function/predicate variable of the same arity. If we denote the tuple $Z_1, \ldots, Z_m$ by Z, and $z_1, \ldots, z_m$ by z, then the definition of circumscription above can be written as

$$A(P, Z) \wedge \neg \exists pz[A(p, z) \wedge p < P].$$

This formula is denoted by $CIRC[A;P;Z]$. If Z is empty, the formula is then denoted as $CIRC[A;P]$. It's the basic circumscription of P in A.

Let's look at a few examples taken from [Lifschitz, 1993].

Example 1. Let A be $P(a)$, where a is an object constant. Z is empty. Therefore, it is a basic circumscription. $CIRC[P(a);P]$ is

$$P(a) \wedge \neg \exists p[p(a) \wedge p < P].$$

If we interpret P as a set and a as an object, then this formula expresses that a belongs to the set denoted by P, and there doesn't exist a proper subset of P that a is a member. Which is equivalent to saying that P is a singleton set. We can conclude that

$$CIRC[P(a); P] \equiv \forall x[P(x) \equiv x = a].$$

Example 2. Let A be $\forall x[Q(x) \supset P(x)]$. With Z empty, circumscribing P in A transforms the implication into an equivalence:

$$CIRC[A; P] \equiv \forall x[Q(x) \equiv P(x)].$$

Example 3. A is the same as in Example 2. With Q varied this time, we get a stronger formula:

$$CIRC[A; P; Q] \equiv \forall x \neg Q(x) \wedge \forall x \neg P(x).$$

This is because any model of $\forall x[Q(x) \supset P(x)]$ in which the extent of P is not empty can be "improved" by making both P and Q empty.

2.3 Nested Abnormality Theories (NATs)

Nested Abnormality Theories (NATs) were proposed by Lifschitz [1995] as a new approach to the use of circumscription for representing knowledge. They are similar to simple abnormality theories introduced by McCarthy [1986], except that their axioms may have a nested structure, with each level corresponding to another application of the circumscription operator. This new style of applying circumscription sometimes leads to more economical and elegant formalization.

2.3.1 Definitions

Consider a second-order language L that does *not* include Ab among its symbols. For every natural number k, by L_k we denote the language obtained from L by adding Ab as a k-ary predicate constant.

Blocks: For any k and any list of function and/or predicate constants[1] $C_1, \ldots, C_m$ ($m \geq 0$) of L, if each of $A_1, \ldots, A_n$ ($n \geq 0$) is a formula of L_k or a *block*, then $\{C_1, \ldots, C_m : A_1, \ldots, A_n\}$ is a *block*. The last expression reads: $C_1, \ldots, C_m$ are such that $A_1, \ldots, A_n$. About $C_1, \ldots, C_m$ we say that they are *described* by this block.

Note that, according to this definition, if A_i and A_j are formulas, and Ab occurs in both, then it is used in both with the same number of arguments; if, however, A_i or A_j is itself a block, then this is not guaranteed.

Nested Abnormality Theory (NAT): A NAT is a set of blocks, called its *axioms*. Note that each axiom is a finite string of symbols, but there may be infinitely many axioms in a NAT.

The semantics of NATs is characterized by a map φ that translates blocks into sentences of L. It is convenient to make φ defined also on formulas of the languages L_k. If A is such a formula, then φA stands for the universal closure of A. For blocks we define, recursively:

$$\varphi\{C_1, \ldots, C_m : A_1, \ldots, A_n\} = \exists ab F(ab),$$

where

$$F(Ab) = \mathrm{CIRC}[\varphi A_1 \wedge \ldots \wedge \varphi A_n; Ab; C_1, \ldots, C_m].$$

A sentence A of L will be identified with the block $\{: A\}$. We will call such blocks *trivial*. It is easy to see that $\varphi\{: A\}$ is equivalent to A.

[1] This includes function constants of arity 0 (object constants) and predicate constants of arity 0 (propositional constants).

For any NAT T, φT stands for $\{\varphi A \mid A \in T\}$. Thus φT is a second-order theory in the language L. A *model* of T is a model of φT in the sense of classical logic. A *consequence* of T is a sentence of L that is true in all models of T.

If a block A is an axiom of T, then inserting an additional formula in A may result in losing some of the consequences of T. In this sense, the formalism defined here is non-monotonic. But adding more axioms to a NAT can only make the set of its consequences larger.

2.3.2 Example

Let's look at an example of using NATs to represent knowledge. We'll take the first example from [Lifschitz, 1995], Section 3.

Example 4.
It's a standard example: *objects normally don't fly; birds normally do; canaries are birds; Tweety is a canary.* These assertions can be formalized as the NAT whose only axiom is

$$\begin{array}{l}\{Flies:\\ \quad Flies(x) \supset Ab(x),\\ \quad \{Flies:\\ \qquad Bird(x) \wedge \neg Ab(x) \supset Flies(x),\\ \qquad Canary(x) \supset Bird(x),\\ \qquad Canary(Tweety)\\ \quad \}\\ \}.\end{array} \tag{1}$$

The outer block describes the ability of objects to fly; the inner block gives more specific information about the ability of *birds* to fly.

In order to apply φ to (1), we first to apply φ to the inner block. It is easy to check, using the methods of [Lifschitz, 1993], Section 3, that the result is equivalent to the conjunction of (the universal closures of) the formulas

$$Bird(x) \supset Flies(x), \tag{2}$$

$$Canary(x) \supset Bird(x) \tag{3}$$

and

$$Canary(Tweety). \tag{4}$$

Using this technique again, we conclude that φ applied to (1) is equivalent to the conjunction of (3), (4) and

$$Bird(x) \equiv Flies(x).$$

3 Using CS

You can use CS on Sun OS, since the SCAN program is compiled under this environment. To use it type:

```
CS input_file output_file m n
```

where m and n are optional. They denote the times in seconds used for simplifying the generated clauses.[2] Their default value is 1. If they are 0, no simplification will be done on the generated clauses.

3.1 Formula Syntax

The formula syntax used by CS is the same as the SCAN/OTTER syntax [McCune, 1990].

Names are alphanumeric strings. A name may contain up to 50 characters. Names are used as constant symbols, function symbols, predicate symbols, propositional variables, and regular variables. White space (spaces, tabs, newlines) can occur anywhere except within names and between a function or predicate symbol and the opening parenthesis. Most terms and atomic formulas (*atoms*) are written in prefix form as is usual in logic. In addition, every name is also an atom (a propositional symbol), and expressions (t_1 `=` t_2) and (t_1 `!=` t_2) are atoms (`=` is the equality predicate, and `!=` is the inequality predicate). White space is required around `=` and `!=`, and parentheses are required.

Formulas

- Atoms are formulas.
- If F and G are formulas, then (F `<->` G) and (F `->` G) are formulas.
- If $F_1, \ldots, F_n$ are formulas, then (F_1 `|` $\ldots$ `|` F_n) and (F_1 `&` $\ldots$ `&` F_n) are formulas.

[2] This simplification process is crucial in some cases. You may want to experiment with larger time limits to see if more simplified results can be obtained.

- The symbols `all` and `exists` are quantifiers. If $Q_1, \ldots, Q_n$ are quantifiers, $x_1, \ldots, x_n$ are names, and F is a formula, then $(Q_1 x_1 \ldots Q_n x_n\ F)$ is a formula.
- If F is a non-negated formula, then `-`F is a formula.

The symbols have their expected meaning: `-` means "not", `<->` means "if and only if", `->` means "implies", `|` means "or", and `&` means "and".

All parentheses are required, and white space is required around `<->`, `->`, `|`, and `&`, and after quantifiers and their associated variables.

3.2 Input File

The input file is a NAT, with each formula written in this syntax and followed by a period, and with no commas between formulas. Formulas in the input file should not have free variables.

For instance, the NAT from Example 4 should be represented like this:

```
{Flies:
   (all x (Flies(x) -> Ab(x))).
   {Flies:
      (all x ((Bird(x) & -Ab(x)) -> Flies(x))).
      (all x (Canary(x) -> Bird(x))).
      Canary(Tweety).
   }
}
```

3.3 Output

CS generates output to an output file specified by the user and to the standard output, the screen.

The following are directed to the screen:

- Error messages listed in Section 3.4
- Error and warning messages from SCAN/OTTER
- The simplification results
- Some statistical information

The following are directed to an output file:

- The simplification results
- Error and warning messages from SCAN/OTTER

Note that the set of formulas generated is sent to both the screen and an output file. SCAN/OTTER generates messages if there are errors in the formulas or if some of the limits are reached such as the time limit or the memory limit.

3.4 Error Messages

The following error messages may be generated if there are mistakes in the input file.

- `input ERROR: missing colon on line n`. This error message is generated if no colon follows the opening braces and the predicate list.
- `input ERROR: mismatched braces on line n`. This error message is generated if the number of opening and closing braces don't match.
- `input ERROR: missing period at the end of formula on line n`. This error message is generated if the period is missing after the formula on line n.
- `input ERROR: too many periods at the end of formula on line n`. This error message is generated if there are two or more periods at the end of the formula on line n.

4 Examples

We'll look at a few examples here.

Example 5. What's On The Table: *blocks are normally on the table; B1 and B2 are two blocks; B1 is not on the table.* This is an example from [Lifschitz, 1993]. The information above can be represented by the following NAT.

```
{Ontable:
   (all x ((Block(x) & -Ab(x)) -> Ontable(x))).
   -Ontable(B1).
   Block(B1).
   Block(B2).
   (B1 != B2).
}
```

We'll get the following as the result:

```
(all x0 (-Block(x0) | Ontable(x0) | x0=B1)).
-Ontable(B1).
Block(B1).
Block(B2).
(B1!=B2).
```

This first of these formulas can be written as:

$$Block(x) \wedge x \neq B1 \supset Ontable(x).$$

Example 6. Whether Birds Can Fly
Now, let's look at Example 4 from Section 2.3.2. Recall that it says *objects normally don't fly; birds normally do; canaries are birds; Tweety is a canary.* The input to CS is the same as the one we used as example in Section 3.

Invoking CS, we get the following as the output:

```
(all x0 (-Canary(x0) | Bird(x0))).
(all x1 (-Bird(x1) | Flies(x1))).
(all x2 (-Flies(x2) | Bird(x2))).
Canary(Tweety).
```

which can be simplified to the conjunction of the (universal closures of the) following formulas:

$$Bird(x) \equiv Flies(x).$$

$$Canary(Tweety).$$

$$Canary(x) \supset Bird(x).$$

Example 7. *Objects normally don't fly; birds normally do; objects that have wings normally are birds; Tweety has wings; Pegasus has wings; Pegasus is not a bird.* This is an enhancement of the second example from [Lifschitz, 1995].

Our input to CS is:

```
{Flies:
   (all x (Flies(x) -> Ab(x))).
   {Flies:
```

```
        (all x ((Bird(x) & -Ab(x)) -> Flies(x))).
        {Bird:
           (all x ((HasWings(x) & -Ab(x)) -> Bird(x))).
           HasWings(Tweety).
           HasWings(Pegasus).
           -Bird(Pegasus).
        }
     }
  }
```

We'll get the following as the result:

```
(all x0 (-HasWings(x0) | Bird(x0) | x0=Pegasus)).
(all x1 (-Bird(x1) | Flies(x1))).
(all x2 (-Flies(x2) | Bird(x2))).
HasWings(Tweety).
HasWings(Pegasus).
-Bird(Pegasus).
```

Example 8. Nixon Diamond Example.
Let's look at the famous Nixon Diamond example. We want to represent that *Quakers are normally pacifists; Republicans are normally hawks; a person cannot be both a pacifist and a hawk; pacifists and hawks are politically active.*

Our input to CS is:

```
{Pacifist,Hawk,PoliticallyActive:
   (all x ((Quaker(x) & -Ab(Aspect1,x)) -> Pacifist(x))).
   (all x ((Republican(x) & -Ab(Aspect2,x)) -> Hawk(x))).
   (all x (-(Pacifist(x) & Hawk(x)))).
   (all x (Pacifist(x) -> PoliticallyActive(x))).
   (all x (Hawk(x) -> PoliticallyActive(x))).
   (Aspect1 != Aspect2).
   (all x (x = x)).
}
```

The last formula `(all x (x = x))` was added since SCAN requires it for simplifying formulas with equalities.

We'll get the following as the result:

```
(all x0 (-Republican(x0) | Hawk(x0) |  -Quaker(x0) | Pacifist(x0))).
(all x1 (-Quaker(x1) | Pacifist(x1) | Republican(x1))).
(all x2 (-Republican(x2) | Hawk(x2) | Quaker(x2))).
(all x3 (-Pacifist(x3) |  -Hawk(x3))).
(all x4 (-Pacifist(x4) | PoliticallyActive(x4))).
(all x5 (-Hawk(x5) | PoliticallyActive(x5))).
(Aspect1!=Aspect2).
(all x6 (x6=x6)).
```

The formula `(Aspect1 != Aspect2)` in the input can be placed outside of the block. However, if we do that, it will not be used for simplifying the results.

Another example that we have looked at is the first example from [Kartha and Lifschitz, 1995]. It was an example used to show how to use the existing methods of computing circumscription to reason about actions. For this example, CS generated a long and complicated second-order formula. The latest version of the SCAN program which supports un-Skolemization can be used to produce a first-order formula. However, the formula is still very lengthy and complicated.

5 Program Structure for CS

5.1 Choice of The Programming Language

We have implemented CS using a script language called PERL, short for Practical Extraction and Report Language. CS contains several PERL scripts, and each of these scripts does a specific job for simplifying NATs. Because our interface with SCAN involves lots of character and string manipulations, and such facilities in PERL are very efficient, it was chosen as the programming language.

5.2 SCAN

The formula φT that defines the semantics of a NAT T (Section 2.3.1) involves second-order quantifiers. Eliminating these quantifiers is the main computational task of CS.

The main part of CS is the SCAN program. SCAN [Gabbay and Ohlbach, 1992] is an algorithm for eliminating second-order quantifiers over predicate variables in formulas of type $\exists P_1, \ldots, P_n F$ where F is any first-order formula. The resulting formula is equivalent to the original formula if the algorithm terminates.

The algorithm normalizes the first-order formula F into clausal form and generates all resolvents of the clauses with predicates P_i. It is shown that the subset of the

generated clauses not containing predicates P_i (maybe infinite) is equivalent to the original formula. Since $\forall P_1, \ldots, P_n F$ is logically equivalent to $\neg\exists P_1, \ldots, P_n \neg F$, we can also use this algorithm to eliminate second-order predicates from universally quantified formulas. An implementation of the SCAN algorithm has been developed by Engel and Ohlbach.

5.3 Outline of Process

CS consists of several parts. Each part does a specific job for the computation. The idea of CS is attacking a nontrivial innermost block first. It computes the results, and replaces the innermost block with the resulting formulas. This process is repeated until no more nontrivial blocks exist in the input file.

The main loop of CS is

while there are nontrivial blocks in the input file
1.extract the first nontrivial innermost block from the input file.
2. simplify this block.
3. replace it with the results from step 2.
end of while loop.

A single-level NAT has the form

$$\begin{array}{l}\{Q: \\ \quad A_1(Ab, Q), \\ \quad \ldots \\ \quad A_n(Ab, Q) \\ \}.\end{array}$$

The map φ translates this block into the formula

$$\exists ab F(ab), \tag{5}$$

where

$$\begin{array}{l} F(Ab) = A_1(Ab, Q) \wedge \ldots \wedge A_n(Ab, Q) \\ \qquad \wedge\neg\exists ab,q[A_1(ab, q) \wedge \ldots \wedge A_n(ab, q) \wedge ab < Ab]. \end{array} \tag{6}$$

SCAN has to be called twice, first to eliminate $\exists ab, q$ in (6), and then to eliminate $\exists ab$ in (5). Recall that there are two optional parameters `m` and `n` when invoking CS. They are parameters of the two calls to SCAN for eliminating redundant clauses. In the first call to SCAN, $A_1(Ab, Q) \wedge \ldots \wedge A_n(Ab, Q)$ are used to simplify generated clauses.

The procedures for computing a single level block are

1. form the input for the first call to SCAN.
2. call the SCAN program.
3. replace the Skolem constants in the resulting clauses
by existentially quantified variables.
4. form F(Ab).
5. call the SCAN program the second time

6 Conclusion

We have presented a program CS to automatically simplify NATs. A few examples have been used to demonstrate the efficacy of the program. In many cases, it generates a set of first-order formulas whose conjunction is equivalent to the second-order formula represented by the NAT used as input. This transformation is useful since automating reasoning for second-order predicate logic is much more difficult than for first-order predicate logic. After this transformation being done, we can then employ a first-order reasoning system to answer questions.

The program has some limitations. Some of these limitations are inherited from the SCAN/OTTER program.

- NATs can have function symbols "described" by the nested blocks. But the SCAN algorithm cannot eliminate function symbols from a set of formulas. Therefore, CS cannot simplify NATs with functions "described".

- The SCAN/OTTER program produces the equivalent first-order formula only if it terminates. Therefore it will be desirable to isolate the classes of second-order formulas that SCAN/OTTER terminates on.

- As demonstrated by the action example that we have tried in Section 4, the simplification ability of SCAN is still not powerful enough. More techniques should be devised to enhance this ability.

- The output of SCAN may include Skolem functions. CS turns them into second-order variables and then returns a second-order formula. Engel and Ohlbach have added un-Skolemization to the SCAN/OTTER program. They haven't made that available. The one that we have been using doesn't have this feature. This problem can be solved if the new version of SCAN/OTTER is available.

- In a NAT T, formulas outside a block can be used to simplify the block. Our current implementation of CS doesn't support this as noted in Example 8. We plan to modify the program to add this feature.

Acknowledgments

First, I would like to thank my advisor Professor Vladimir Lifschitz for his patience and helpful advice. Also I would like to thank Neelakantan Kartha, a recent Ph.D. from the University of Texas at Austin for his help on using the SCAN program. Special thanks to Thorsen Engel and Hans Juergen Ohlbach for making the implementation of SCAN available to us. Thanks also go to Professor Robert Boyer and Professor Alan Cline for reading this paper.

Appendix: A sample session of CS

The input file `nat` contains the NAT from Example 7.

CS was invoked by

```
CS nat out
```

It produced the following to the standard output (the screen).

```
Job run on 1995/5/23 13:43:24

Please be patient.  This will take a while.

Processing ...

DONE.

The output is left in file: out

Job finished on 1995/5/23 13:43:48
Time elapsed: 24 seconds.
========================================
              O U T P U T
```

```
=======================================

The inputed NAT is equivalent to the
CONJUNCTION of the following formula(s).
Hope that's what you have expected.

(all x0 (-HasWings(x0) | Bird(x0) | x0=Pegasus)).
(all x1 (-Bird(x1) | Flies(x1))).
(all x2 (-Flies(x2) | Bird(x2))).
HasWings(Tweety).
HasWings(Pegasus).
-Bird(Pegasus).
```

Time elapsed is the time between the job's beginning and its termination.[3]

References

[1] Dov Gabbay and Hans J. Ohlbach. Quantifier Elimination in Second-Order Predicate Logic. In Bernhard Nebel, Charles Rich, and William Swartout, editors, *Proc. of the Third Int'l Conf. on Principles of Knowledge Representation and Reasoning*, 1992.

[2] Matthew L. Ginsberg. AI and Nonmonotonic Reasoning. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *The Handbook of Logic in AI and Logic Programming*, volume 3, pages 1-33. Oxford University Press, 1993.

[3] G. Neelakantan Kartha and Vladimir Lifschitz. A Simple Formalization of Actions Using Circumscription. Submitted for publication, 1995.

[4] Vladimir Lifschitz. Circumscription. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *The Handbook of Logic in AI and Logic Programming*, volume 3, pages 298–352. Oxford University Press, 1993.

[5] Vladimir Lifschitz. Nested Abnormality Theories. To appear in *Artificial Intelligence*, 1995.

[6] John McCarthy. Circumscription—a form of non-monotonic reasoning. *Artificial Intelligence*, 13(1, 2):27–39,171–172, 1980.

[3] The number may vary due to load of the specific machine that CS is running on.

[7] John McCarthy. Applications of circumscription to formalizing common sense knowledge. *Artificial Intelligence*, 26(3):89–116, 1986.

[8] John McCarthy. Mathematical logic in Artificial Intelligence. *Dædalus*, pages 297–311, 1988.

[9] John McCarthy. *Formalizing common sense: papers by John McCarthy.* Ablex, Norwood, NJ, 1990.

[10] William W. McCune. OTTER 2.0 Users Guide. Argonne National Laboratory, 1990.

[11] David Poole. A logical framework for default reasoning. *Artificial Intelligence*, 36(1):27–48, 1988.

[12] Larry Wall. PERL manual. PERL – Practical Extraction and Report Language. URL: http://www.cis.ufl.edu/cgi-bin/plindex.