Efficient Negation Using Abstract Interpretation

Susana Muñoz, Juan José Moreno, Manuel Hermenegildo

Department of Computer Science, Technical U. of Madrid *

Abstract. While negation has been a very active area of research in logic programming, comparatively few papers have been devoted to implementation issues. Furthermore, the negation-related capabilities of current Prolog systems are limited. We recently presented a novel method for incorporating negation in a Prolog compiler which takes a number of existing methods (some modified and improved by us) and uses them in a combined fashion. The method makes use of information provided by a global analysis of the source code. Our previous work focused on the systematic description of the techniques and the reasoning about correctness and completeness of the method, but provided no experimental evidence to evaluate the proposal. In this paper, we report on an implementation, using the Ciao Prolog system preprocessor, and provide experimental data which indicates that the method is not only feasible but also quite promising from the efficiency point of view. In addition, the tests have provided new insight as to how to improve the proposal further. Abstract interpretation techniques are shown to offer important improvements in this application.
Keywords: Negation, Constraint Logic Programming, Program Analysis, Logic Programming Implementation, Abstract Interpretation.

1 Introduction

The fundamental idea behind Logic Programming (LP) is to use a computable subset of logic as a programming language. Probably, negation is the most significant aspect of logic that was not included from the start due to the significant additional complexity that it involves. However, negation has an important role for example in knowledge representation, where many of its uses cannot be simulated by positive programs. The different proposals differ not only in expressivity but also in semantics. Presumably as a result of this, implementation aspects have received comparatively little attention. A search on the The Collection of Computer Science Bibliographies [13] with the keyword "negation" yields nearly 60 papers, but only 2 include implementation in the keywords, and fewer than 10 treat implementation issues at all. Perhaps because of this, the negation techniques supported by current Prolog compilers are rather limited.

Our objective is to design and implement a practical form of negation and incorporate it into a Prolog compiler. In [19] we studied systematically what

* **Mail:** Facultad de Informática, Universidad Politécnica de Madrid, Campus de Montegancedo s/n, 28660, Madrid, SPAIN. **email:** `susana|jjmoreno|herme@fi.upm.es`. **voice:** +34-91-336-7455. **fax:** +34-91-336-7412.

we understood to be the most interesting existing proposals: negation as failure (*naf*) [8], use of delays to apply *naf* in a secure way [15], intensional negation [1],[2], and constructive negation [6],[7]. We could not find a single technique that offered both completeness and an efficient implementation. However, we proposed to use a combination of these techniques and that information from a static analysis of the program could be used to reduce the cost of selecting among techniques. We provided a coherent presentation of the techniques, implementation solutions, and a proof of correctness for the method, but we did not provide any experimental evidence to support the proposal. This is the purpose of this paper. One problem that we face is the lack of a good collection of benchmarks using negation to be used in the tests. One of the reasons has been discussed before: there are few papers about implementation of negation. Another fact is that negation is typically used in small parts of programs and it is very difficult to find it because it is not one of their main components. Additionally, the lack of sound implementations makes programmers avoid negations, even complicating the code or changing its semantics. We have had to collect a number of examples using negation from logic programming textbooks, research papers, and our own experience teaching Prolog.

We have tested these examples with all of our techniques in order to establish their efficiency. We have also measured the improvement of efficiency thanks to the use of the static analyzers. We have used the Ciao system [4] that is an efficient Prolog implementation and incorporates all the needed static analyses. However, it is important to point out that the techniques used are fairly standard, so they can be incorporated into almost any Prolog compiler.

In both cases the results have been very interesting. The comparison of the techniques has allowed us to improve the right order in which to apply them. Furthermore, we have learned that the impact of the use of the information from the analyzers is quite significant.

The rest of the paper is organized as follows. Section 2 presents more details on our method to handle negation and how it has been included in the Ciao system. Section 3 presents the evaluation of the techniques and how the results have helped us reformulate our strategy. The impact of the use of abstract interpretation is studied in 3.3.

2 Implementation of a Negation System

In this section we present shortly the techniques from the literature which we have integrated in a uniform framework. The techniques and the proposed combination share the following characteristics:

- We are interested in techniques with a single and simple semantics. The simplest alternative is to use the Closed Word Assumption (CWA) [8] by program completion and Kunen's 3-valued semantics [11]. These semantics will be the basis for soundness results.
- Another important issue is that they must be "constructive", i.e., program execution should produce adequate goal variable values for making a negated

goal false. Chan's constructive negation [6],[7] fulfills both objectives. However, it is difficult to implement and expensive in terms of execution resources. Our idea is to use the simplest technique for each particular case.
- The formulations need to be uniform in order to allow the mixture of techniques and to establish sufficient correctness conditions to use them.
- We also provide a Prolog implementation of each of the techniques so that they can be easily combined and we obtain a portable implementation.

2.1 Disequality constraints

An instrumental step in order to manage negation in a more advanced way is to be able to handle disequalities between terms such as $t_1 \neq t_2$. Prolog implementations typically include only the built-in predicate `/== /2` which can only work with disequalities if both terms are ground and simply succeeds in the presence of free variables. A "constructive" behavior must allow the "binding" of a variable with a disequality. On the other hand, the negation of an equation $X = t(\overline{Y})$ produces the universal quantification of the free variables in the equation, unless a more external quantification affects them. The negation of such an equation is $\forall\ \overline{Y}\ X \neq t(\overline{Y})$.

We have defined a predicate `=/= /2`, used to check disequalities, in a similar way to explicit unification (`=`). The main difference is that it incorporates negative normal form constraints instead of bindings and the decomposition step can produce disjunctions. When a universal quantification is used in a disequality (e.g., $\forall Y\ X \neq c(Y)$) the new constructor `fA/1` is used (e.g., `X / c(fA(Y))`).

2.2 Negation Techniques

- **Negation as failure and delays:** Typical Prolog systems implementation of `naf(Q)` is unsound unless the free variables of Q are ground. The sound version ensures that the call to `naf` is made only when the variables of the negated goal are ground (although it has the risk of floundering). It replaces a call to $\neg p(\overline{X})$ by: $\ldots, when(ground(\overline{X}), naf(p(\overline{X}))), \ldots$
- **Constructive negation for finite solutions:** We have implemented a Prolog predicate `cnegf(Q)` to implement finite constructive negation, that can be used if the number of solutions can be determined to be finite. It calculates the negation of the disjunction of all solutions of Q. It is a simple and efficient version of the constructive negation.
- **Intensional negation and universal quantification:** Intensional negation is a novel approach to obtain the program completion by transforming the original program into a new one that introduces the "only if" part of the predicate definitions (i.e., interpreting implications as equivalences). We reformulate the transformation by using a single constraint to express the complement of a term, instead of a set of terms. The transformation is fully formalized in [19].

– **General constructive negation:** Full constructive negation is needed when all the previous techniques are not applicable. While there are several papers treating theoretical aspects of it, we have not found papers with full descriptions of implementations. We have completed a simple implementation from scratch (`cneg/1`) which is complete, although it can certainly be improved.

2.3 Strategy

Our starting point is a (pseudo)predicate `neg/1` which computes constructively the negation of any Prolog (sub)goal $\neg G(\overline{X})$, selecting the most appropriate technique at run-time. However, the program is analyzed and optimized at compile-time to generate specialized versions of `neg` for each negated literal in the program (each call to `neg`), using only the simplest technique required. The basic decision steps are:

1. Groundness of $\overline{X}$ is checked before the call to G. If proved true statically, then simple negation as failure is applied, i.e., $\neg G(\overline{X})$ is compiled to `naf(`$G(\overline{X})$`)`.[1]
2. Otherwise, a new program is generated replacing the goal $\neg G(\overline{X})$ by `when(ground(`$\overline{X}$`), naf(`$G(\overline{X})$`))` and the "elimination of delays" technique is applied to it. If the analysis and the program transformation are able to remove the delay (perhaps moving the goal) the resulting program is used.[2]
3. Otherwise, if the finiteness analysis over $G(\overline{X})$ succeeds, then finite constructive negation can be used, transforming the negated goal into `cnegf`$(G(\overline{X}))$.
4. Otherwise, the intensional negation approach is tried by generating the corresponding negated predicates and replacing the goal by `call_not(`$G(\overline{X})$`, S)` that will call `not_G(`$\overline{X}$`)`. During this process, new negated goals can appear and the same compiler strategy is applied to each of them. If `S` is bound to `success` or `fail` then negation is solved, otherwise we continue.
5. If everything fails, full constructive negation must be used and the executed goal is `cneg(`$G(\overline{X})$`)`.

The strategy is complete and sound with respect to Kunen 3-valued semantics. This follows from the soundness of the negation techniques, the correctness of the analysis, and the completeness of constructive negation.

Let us illustrate the behavior of the method by using some simple examples. Consider the following program:

[1] Since floundering is undecidable, the analysis only provides an approximation of the cases where negation as failure can be applied safely. This means that in some cases the technique will not be applied even it might provide a sound result.

[2] Again, the approximations made during analysis could result in the method not being applied in some cases in which it might still provide a sound result.

```
less(0, s(Y)).                          member(X, [X|L]).
less(s(X), s(Y)) :- less(X, Y).         member(X, [Y|L]) :- member(X, L).

p1(X) :- member(X, [0, s(0)]),          p3(X) :- neg(less(X, s(s(0)))).
         neg(less(X, s(0))).            p4(X) :- neg(less(s(0), X)).
p2(X) :- neg(less(X, s(0))),            p5(X) :- neg(less(X, s(X))).
         member(X, [0, s(0)]).
```

Each of the $\mathtt{p}_i$ predicates requires a different variant. For `p1`, the groundness test for variable `X` succeeds and `naf/1` can be used, so it behaves as:

```
p1(X) :- member(X, [0, s(0)]),          ?- p1(X).
         naf(less(X, s(0))).               X = s(0)
```

Applying the "elimination of delays" analysis to the program:

```
p2(X) :- when(ground(X), naf(less(X, s(0)))),
         member(X, [0, s(0)]).
```

the delay can be eliminated, reordering the goals as follows:

```
p2(X) :- member(X, [0, s(0)]),          ?- p2(X).
         naf(less(X, s(0))).               X = s(0)
```

The case for `p3` is solved because the finiteness test can be proved to succeed, so the program is rewritten as:

```
p3(X) :- cnegf(less(X, s(s(0)))).       ?- p3(X).
                                           X / 0 ,  X / s(0)
```

`p4` needs intensional negation, so the generated program is:

```
not__less(W, Z) :- W =/= 0,             ?- p4(X).
        fA(X, W =/= s(X)),                 X = 0 ?;
        fA(Y, Z =/= s(Y)).                 X = s(0)
not__less(s(X), s(Y)) :-
        not__less(X, Y).
p4(X) :-
        not__less(s(0), X).
```

Finally, `p5` needs full constructive negation because the intensional approach is not able to give a result:

```
p5(X) :- cneg(less(X, s(X))).           ?- p5(X).
                                           no
```

3 Evaluating the strategy

3.1 Example programs

As mentioned earlier, one problem that we have faced is the lack of a good collection of benchmarks using negation to be used in the tests. We have, however, collected a number of examples using negation from logic programming textbooks, research papers, and our own experience teaching Prolog:

- **disjoint**: Code to verify that two lists have no common elements. Negation is used to check that elements of the first list are not in the second one.
- **jugs**: Classical jugs puzzle. A sequence of actions is planned that will produce 4 gallons of water in the larger jug. Negation is used to check that the state of the jugs is not repeated during the process.

- **robot**: Simulation of the behavior of a robot. Negation is used to check that possible new positions for the robot are not dangerous.
- **trie**: It finds the list of word-FileList couples that shows the sublist of files where each word appears (from an initial list of words and files). Negation is used when reading words to find the first non alphanumeric character.
- **numbers9**: It uses negation to detect impossible cases in balanced trees.
- **closure**: Transitive closure of a network. Negation is used to avoid infinite loops (detecting repeated nodes). From [16] page 169.
- **union**: It is used `neg(member (X, L1))` to check if an element X appears in both lists (for union of two lists without repetitions). From [16] page 154.
- **include**: $include(P, Xs, Ys)$ is true when Ys is the list of the elements of Xs such that $P(X)$ is true. Negation is used to detect elements that do not satisfy the property $P(X)$. From [16] page 227.
- **flatten**: Flattening a list using difference-lists. Negation is used to consider lists that are not empty. From [12] Program 915.2, page 241.
- **lessNodd**: Returns the list of odd natural numbers that are less than a number N. Negation is used to control that a number is not even.
- **friend**: Deduces the relationship between two people using the stored information from a database. Negation is used to exclude ancestors and descendants from the category of friends of a person.

3.2 Experimental results

We have first measured the execution times in milliseconds for the previous examples when using all the different (applicable) negation techniques that we have discussed, and also noted which technique is selected by our strategy (in boldface). A '–' in a cell means that the technique is not applicable. All measurements were made using Ciao Prolog[3] 1.5 on a Pentium II at 350 Mhz. Small programs were executed a sufficient number of times to obtain repeatable data. The results are shown in Table 1, where each column means:

- **const.** shows the time taken by general constructive negation (`cneg`).
- **naf/delay** uses either `naf` directly or within a delay directive. A 'D' is placed before the time in the second case.
- **fin.const.** is the time of the finite version of constructive negation, `cnegf`.
- **intens.** uses the `not_'p'` predicate from the intensional negation program transformation.
- **ratio** columns measure the speedup of the technique to their left w.r.t. constructive negation. An 'x' means the ratio is extremely high.

It is clear that the technique chosen by our strategy is always equal to or better than general constructive negation. In many cases, it is also the best possible of the examined techniques. We now study each technique separately:

[3] The negation system is coded as a library module ("package" [5]), which includes the corresponding syntactic and semantic extensions (i.e. Ciao's attributed variables). Such extensions apply locally within each module which uses this negation library.

programs	const.	naf/delay	ratio	fin.const.	ratio	intens.	ratio
disjoint1	7440	**780**	9.5	2740	2.7	-	-
disjoint2	3330	-	-	**1120**	2.9	-	-
jugs	8140	**859**	9.4	2175	3.7	<1	x
robot	4600	**1310**	3.5	1900	2.4	-	-
trie	8950	**1850**	4.8	2140	4.1	-	-
numbers9	286779	-	-	-	-	**25230**	11.3
closure1a	5100	**730**	6.9	1450	3.5	140	36.4
closure2a	3520	**560**	6.2	900	3.9	100	35.2
closure3a	10550	**1700**	6.2	2700	3.9	280	37.6
closure1b	26350	D**2240**	11.7	16460	1.6	8570	3.0
closure2b	17400	D**1500**	11.6	10580	1.6	5420	3.2
closure3b	16700	D**4510**	3.7	10120	1.6	16070	1.0
union1	1150	**300**	3.8	320	3.5	189	6.0
union2	20930	-	-	**9470**	2.2	2940	7.1
include1	9020	**1270**	7.1	2680	3.3	170	53.0
include2	9910	-	-	**2995**	3.3	-	-
flatten	32379	**8500**	3.8	12570	2.5	10	x
lessNodd1	58980	**4850**	12.1	17550	3.3	1270	46.4
lessNodd2	7750	**1490**	5.2	2700	2.8	-	-
lessNodd3	>3600000	-	-	-	-	**1540**	x
friend1a	16150	**2280**	7.0	-	-	39500	0.4
friend2a	17630	**<1**	x	-	-	10	x
friend3a	447200	D**4430**	100.9	-	-	43200	10.3
friend4a	>3600000	D**8750**	x	-	-	>3600000	x
friend1b	17350	**3020**	5.74	-	-	9	x
friend2b	17650	**<1**	x	-	-	10	x
friend3b	92500	D**3060**	30.2	-	-	43200	2.1
friend4b	>3600000	D**6050**	x	-	-	171290	x
average			13.0		2.9		18.3

Table 1. Comparing different negation techniques

- Using **naf** instead of **const.** results in speed-ups that range from 3.5 to 30.2. The average is more than 8.
- The **delay** technique, when applicable, has a considerable impact, speeding programs up to 100 times.
- The **fin.const.** technique is around 3 times faster than **const.**.
- **intens.** has a more random behavior. Very significant speed-ups are interleaved with more modest results and even some slow-down (*friend1a*).

The most surprising result is the efficiency of intensional negation. The transformational approach seems the most adequate in those cases, provided that we restrict the use of the technique to the case where there are no universal quantifications in the resulting program. On the other hand, it is possible that the intensional program may not be able to produce a result (wasting time) and its use is a dynamic decision. Although these problems do not arise often in practice, they are a serious risk. As a result we modified the strategy to use intensional negation as the preferable technique, but only when it can be used safely.

The overall conclusion is that, at least for the benchmarks studied, our strategy produces notable benefits. It preserves the completeness of general constructive negation but typically at a fraction of the cost.

3.3 Measuring the impact of abstract interpretation

As mentioned above, the selection strategy and the program optimizations performed make use of information from global program analysis. We have obtained the information and performed the transformations using the analyzers and specializers that are part of the Ciao system's preprocessor, CiaoPP [10]. In particular, from the analysis point of view, the *groundness* analysis has been performed using the domain and algorithms described in [14]. In order to *eliminate delays* a technique is used which, given a program with delays, tries to identify those that are not needed, perhaps after some safe reordering of literals, as described in [9, 17]. Finally, the upper bounds complexity and execution cost analysis [4] has been used to determine *finiteness in the number of solutions*.

The transformations have been implemented using the specializer in CiaoPP [18]. The source programs always make calls to a version of the generic predicate similar to the `neg` predicate presented in section 2. The specializer creates specialized versions of the generic predicate for each literal calling `neg` in which tests and clauses are eliminated as determined by the information available from the analyzers. For example, if the groundness test is proven true at compile-time, the specializer will eliminate the test and the rest of the clauses of `neg` and eventually even replace the literal calling `neg` with a direct call to `naf`. This is done automatically by CiaoPP without having to write any additional code.

In order to estimate the advantages obtained by using this approach we now present some experimental results comparing the execution time of the programs that might be generated without the help of the analyzers and the versions produced automatically by the Ciao preprocessor. In the first case, the calls to `neg` always call (a slightly modified version of) the full version of the `neg` predicate. Thus, for example, the groundness test is performed at execution time. The clause to check the finiteness of the goal and then call `cnegf` is removed since such checking cannot be made safely at run-time. Moreover, the delay technique is not used because, in general, it has the risk of floundering. In contrast, the version obtained with the help of the analyzers can remove the groundness check, use the reordering proposed by the elimination of delays, and use the information of the finiteness analysis to call `cnegf`.

Table 2 presents the results. We have also added for reference columns showing the execution time of using `naf` directly and a secure version of `naf`, i.e., checking groundness before. Finally, we have also added the time taken by CiaoPP to perform the analysis and transformation.

[4] Note that an upper bound cost that is not infinity implies a finite number of solutions (an alternative is [3].

program	with pp.	without pp.	ratio	naf	ratio	secure naf	ratio	prep.
disjoint1	1020	1700	1.66	780	0.76	1469	1.44	78
jugs	969	8419	8.68	859	0.88	1690	1.74	227
robot	1960	3100	1.58	1310	0.66	1800	0.91	700
trie	1890	2450	1.29	1850	0.97	1900	1.00	508
union1	300	350	1.16	230	0.76	300	1.00	119
closure1a	730	2600	3.56	730	1.00	900	1.23	257
closure2a	570	1970	3.45	560	0.98	670	1.17	257
closure3a	1710	5050	2.95	1700	0.99	2010	1.17	257
include1	1099	1180	1.07	1080	0.98	1270	1.15	178
flatten	8859	9300	1.04	8500	0.95	8080	0.91	168
lessNodd1	7310	8670	1.18	4850	0.66	6300	0.86	58
lessNodd2	1780	1830	1.02	1490	0.83	1590	0.89	58
friend1b	3220	3360	1.04	3020	0.93	3180	0.98	198
friend1a	2820	2860	1.01	2280	0.80	2840	1.00	198
average			2.33		0.86		1.10	
closure1b	610	8610	14.11	-	-	-	-	257
closure2b	570	5700	10.00	-	-	-	-	257
closure3b	1800	16300	9.05	-	-	-	-	257
friend3a	3100	43350	13.98	-	-	-	-	198
friend4a	6210	>3600000	x	-	-	-	-	198
friend3b	3100	43400	14.00	-	-	-	-	198
friend4b	6210	171495	27.61	-	-	-	-	198
average			14.79					
disjoint2	1125	3700	3.28	-	-	-	-	78
union2	9590	21010	2.19	-	-	-	-	119
include2	3070	10010	3.26	-	-	-	-	178
average			5.65					
average			2.37		0.86		1.10	

Table 2. Impact of program analysis

The table reveals that the impact of abstract interpretation is significant enough to justify its use. For those examples where `naf` is applicable, the analyzer is able to detect groundness statically in all the cases, so the call to `neg` is replaced by `naf`. It is worth mentioning that the implementation of the dynamic groundness test in Ciao is quite efficient (it is performed at a very low level, inherited from its &-Prolog origins). Even so, the speedup can reach a factor of over 8, and the average is 2.33. The impact of the elimination of delay is even better in general. Notice that if the delay technique is not used, intensional negation could be used instead, which in many cases is a very efficient approach. Even with this drawback, the use of abstract interpretation is helpful. When the finiteness analysis avoids the use of full constructive negation the speed-ups are greater than 3. The difference between the programs after preprocessing and the direct use of `naf` is negligible. The code produced by the preprocessor is better than the secure use of `naf` because of the elimination of groundness tests.

Acknowledgments

We are grateful to M. Carro and D. Cabeza for providing examples and to F. Bueno, and G. Puebla for their support while using the Ciao system preprocessor. This work was funded in part by MCyT project EDIPIA (TIC99-1151).

References

1. R. Barbuti, D. Mancarella, D. Pedreschi, and F. Turini. Intensional negation of logic programs. *Lecture notes on Computer Science*, 250:96–110, 1987.
2. R. Barbuti, D. Mancarella, D. Pedreschi, and F. Turini. A transformational approach to negation in logic programming. *JLP*, 8(3):201–228, 1990.
3. C. Braem, B. Le Charlier, S. Modart, and P. Van Hentenryck. Cardinality analysis of Prolog. In *ILPS*, pages 457–471. The MIT Press, 1994.
4. F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla. The Ciao Prolog System. Reference Manual. Technical Report CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM), August 1997. System and manual at `http://www.cliplab.org/Software/Ciao/`.
5. D. Cabeza and M. Hermenegildo. A New Module System for Prolog. In *CL2000*, number 1861 in LNAI, pages 131–148. Springer-Verlag, July 2000.
6. D. Chan. Constructive negation based on the complete database. In *Proc. Int. Conference on LP'88*, pages 111–125. The MIT Press, 1988.
7. D. Chan. An extension of constructive negation and its application in coroutining. In *Proc. NACLP'89*, pages 477–493. The MIT Press, 1989.
8. K. L. Clark. Negation as failure. In J. Minker H. Gallaire, editor, *Logic and Data Bases*, pages 293–322, New York, NY, 1978.
9. M. García , K. Marriott, and P. Stuckey. Efficient analysis of constraint logic programs with dynamic scheduling. In *ILPS*, pages 417–431. MIT Press, 1995.
10. M. Hermenegildo, F. Bueno, G. Puebla, and P. López-García. Program Analysis, Debugging and Optimization Using the Ciao System Preprocessor. In *1999 ICLP*, pages 52–66, Cambridge, MA, November 1999. MIT Press.
11. K. Kunen. Negation in logic programming. *JLP*, 4:289–308, 1987.
12. E. Shapiro L. Sterling. *The Art of Prolog*. The MIT Press, 1987.
13. The Collection of Computer Science Bibliographies. `http://liinwww.ira.uka.de/bibliography/LogicProgramming/index.html`.
14. K. Muthukumar and M. Hermenegildo. Compile-time derivation of variable dependency using abstract interpretation. *JLP*, 13(2/3):315–347, July 1992.
15. L. Naish. Negation and Control in Prolog. In *LNCS*, number 238. Springe, 1985.
16. R. A. O'Keefe. *The Craft of Prolog*. The MIT Press, 1990.
17. G. Puebla, M. García , K. Marriott, and P. Stuckey. Optimization of Logic Programs with Dynamic Scheduling. In *1997 International Conference on LP*, pages 93–107, Cambridge, MA, June 1997. MIT Press.
18. G. Puebla and M. Hermenegildo. Abstract Multiple Specialization and its Application to Program Parallelization. *JLP*, 41(2&3):279–316, November 1999.
19. J.J. Moreno S. Muñoz. How to incorporate negation in a prolog compiler. In V. Santos Costa E. Pontelli, editor, *2nd International Workshop PADL'2000*, volume 1753 of *LNCS*, pages 124–140, Boston, MA (USA), 2000. Springer.