Removing Unnecessary Variables from Horn Clause Verification Conditions

Emanuele De Angelis[*] and Fabio Fioravanti[*]
DEC, University "G. d'Annunzio" of Chieti-Pescara, Italy
{emanuele.deangelis, fabio.fioravanti}@unich.it

Alberto Pettorossi[*]
DICII, University of Rome Tor Vergata, Italy
adp@iasi.cnr.it

Maurizio Proietti
CNR-IASI, Rome, Italy.
proietti@iasi.cnr.it

Verification conditions (VCs) are logical formulas whose satisfiability guarantees program correctness. We consider VCs in the form of constrained Horn clauses (CHC) which are automatically generated from the encoding of (an interpreter of) the operational semantics of the programming language. VCs are derived through program specialization based on the unfold/fold transformation rules and, as it often happens when specializing interpreters, they contain *unnecessary* variables, that is, variables which are not required for the correctness proofs of the programs under verification. In this paper we adapt to the CHC setting some of the techniques that were developed for removing unnecessary variables from logic programs, and we show that, in some cases, the application of these techniques increases the effectiveness of Horn clause solvers when proving program correctness.

1 Introduction

Correctness of an imperative program P can be verified by: first, (i) generating *verification conditions* (VCs, for short) for the program P and the considered property, and then, (ii) using SMT solvers for checking the satisfiability of the VCs.

In this paper we consider VCs which are automatically derived by applying program specialization to a constrained Horn clause encoding of the operational semantics of the programming language. (In this paper we will use the notions of *Constrained Horn Clauses* (CHC) and *constraint logic programs* (CLP) interchangeably.) Program specialization is based on the application of semantics preserving unfold/fold transformation rules, guided by a strategy, called the *VCG strategy*, which has been specifically designed for VCs generation (see [4] for a detailed presentation). Other notable applications of CLP program specialization to the analysis of imperative or object-oriented programs can be found in [1, 11].

Given an imperative program P and a safety property, we introduce a CLP program I, which defines the nullary predicate `unsafe` such that P is safe if and only if the atom `unsafe` is not derivable from I or, equivalently, `unsafe` does not belong to the *least model* of I, denoted $\mathscr{M}(I)$.

The VCG strategy works by performing the so-called *removal of the interpreter*, that is, it removes the level of interpretation which is present in the initial CLP program I, where commands are encoded as CLP clauses and there are references to the operational semantics of the imperative programming language. The output of the VCG strategy is a program I_{sp} such that `unsafe` $\in \mathscr{M}(I)$ iff `unsafe` $\in \mathscr{M}(I_{sp})$. Moreover, due to the absence of the interpretative level, the test of whether or not `unsafe` belongs to $\mathscr{M}(I_{sp})$ is often easier than the test of whether or not `unsafe` belongs to $\mathscr{M}(I)$.

[*]Research associate at CNR-IASI, Rome, Italy.

The specialization-based approach for generating VCs is parametric with respect to: (i) the imperative program P, (ii) the operational semantics of the imperative language in which the program P is written, (iii) the property to be proved, and (iv) the logic used for specifying the property of interest (in this case, the reachability of an unsafe state).

One of the most significant advantages of this approach is that it enables the design of widely applicable VC generators for programs written in different programming languages, and for different operational semantics of languages with the same syntax, by making small modifications only [4].

2 Removing Unnecessary Variables

It is well known that program specialization and transformation techniques often produce clauses with more arguments than those that are actually needed [9, 12, 6]. Thus, it is not surprising to observe that such a side-effect also occurs when generating VCs via program specialization. Indeed, it is often the case that some of the variables occurring in the CLP program I_{sp}, which is generated by the VCG strategy, are not actually needed to check whether or not `unsafe` $\in \mathcal{M}(I_{sp})$. Avoiding those unnecessary variables, and thus deriving predicates with smaller arity, can increase the effectiveness and the efficiency of applying Horn clause solvers, and proving program correctness.

Now we present two transformation techniques which allow us to reduce the number of arguments of the predicates used in the VCs. These techniques extend to the case of CLP programs similar techniques that have been developed for logic programs [9, 12]. The first technique is a transformation strategy, called the *Non-Linking variable Removal strategy* (or the *NLR strategy*, for short) that removes variables occurring as arguments of an atom in the body of a clause, but that do not occur elsewhere in the clause. The second technique, called the *constrained FAR algorithm* (or the *cFAR algorithm*, for short), is a generalization of a liveness analysis, and removes arguments that are not actually used during program execution.

1. Non-Linking variable Removal Strategy. First, we consider the NLR strategy whose objective is to remove the *non-linking variables*. They are defined as follows.

Definition 1 (Linking variables [12]**)** Let C be the clause `H :- c, L, B, R`, where: `c` is a constraint, `L` and `R` are (possibly empty) conjunctions of atoms, and `B` is an atom. The set of *linking variables* of `B` in C, denoted by $linkvars(\mathtt{B}, C)$, is $vars(\mathtt{B}) \cap vars(\{\mathtt{H,c,L,R}\})$. The set of *non-linking variables* of `B` in C is $vars(\mathtt{B}) - linkvars(\mathtt{B}, C)$.

Before presenting the NLR strategy, we see it in action in an example. Let us consider the C program P in Figure 1. We want to verify the Hoare triple $\{\mathtt{x} \geq 0\}\ P\ \{\mathtt{y} \leq 0\}$. By applying the VCG strategy, we get the set of clauses $P1$ in Figure 1, where `unsafe` holds iff the Hoare triple is not valid. In $P1$ the non-linking variables have been underlined. Then, by applying the NLR strategy, we get the set of clauses $P2$ without non-linking variables (see Figure 2). $P1$ and $P2$ are equivalent with respect to the query `unsafe`, in the sense that `unsafe` $\in \mathcal{M}(P1)$ iff `unsafe` $\in \mathcal{M}(P2)$.

In particular, NLR replaces the predicates `newp1` and `newp2`, which are called with the non-linking variables `X2`, `Y1`, and `Y2` (see clauses 1 and 2 of $P1$ in Figure 1), with two new predicates `newp3` and `newp4`, respectively, whose arguments are linking variables only. Note that the removal of the two arguments `Y1` and `X2` of `newp1`, which are the non-linking variables in clause 1, determines in clause 2 the removal of the two arguments `Y1` and `X2`, which are *linking* variables of `newp2`. Thus, from `newp2` with six arguments in clause 2, by removing also the non-linking variable `Y2`, we get the predicate `newp4` in clauses 3' and 4' of program $P2$ with three arguments only (see Figure 2).

```
int x,y;
void main() {
  int z=x+1;
  while(z<=9)
      z=z+1;
  y=z;
}
```

The C program P

```
1. unsafe:- X1>=0, Y2=<0, newp1(X1,Y1,X2,Y2).
2. newp1(X1,Y1,X2,Z2):- Z1=X1+1,
       newp2(X1,Y1,Z1,X2,Y2,Z2).
3. newp2(X1,Y1,Z1,X2,Y2,Z2):- Z1=<9, Z3=Z1+1,
       newp2(X1,Y1,Z3,X2,Y2,Z2).
4. newp2(X1,Y1,Z1,X1,Y1,Z1):- Z1>=10.
```

Program $P1$: Verification Conditions obtained by VCG

Figure 1: Program $P1$ is the set of Verification Conditions VCs obtained by applying the VCG strategy starting from the C program P, the initial condition $\mathtt{x} \geq 0$ and the error property $\mathtt{y} \leq 0$.

The NLR strategy consists in a repeated application of the *unfolding*, *definition introduction*, and *folding* transformation rules [5].

We assume that the input of NLR is any CLP program *Prog*. To keep the notation simple, we will identify a tuple of variables with the set of variables occurring in it. The union of two tuples is constructed by erasing duplicate elements.

During the execution the NLR strategy maintains in a set *Defs* all the definitions that have been introduced so far. Every definition clause in *Defs* is unfolded with respect to the leftmost atom in its body, thereby producing a set S of clauses. Then every clause in S is folded (repeatedly, with respect all atoms in its body) by using either definitions that already occur in *Defs* or new definitions that are introduced in *Defs* for performing those folding steps.

The peculiarity of the NLR strategy lies in the careful management of the set of variables occurring in the head of the definition clauses.

Let C be a clause in S of the form: `H :- c, L, B, R`, where the predicate symbol of `B` occurs in *Prog*. If C cannot be folded with respect to the atom `B` using any clause in *Defs*, then we have to introduce a new definition clause as we now explain.

First, we consider a definition F whose head contains only the linking variables of the atom `B` in the clause C. Let F be `newp(V):-B`, where `newp` is a predicate symbol not occurring in the set $\mathit{Prog} \cup \mathit{Defs}$, and `V` is the set $\mathit{linkvars}(\mathtt{B}, C)$ of the linking variables of `B` in C.

If the set *Defs* contains a clause D of the form `newq(Q) :- S` such that, for some renaming substitution ϑ, $\mathtt{B}\vartheta = \mathtt{S}$, then we replace clause D in *Defs* with the clause `newp(L):-B`, where $\mathtt{L} = \mathtt{V}\vartheta \cup \mathtt{Q}$. Otherwise, we introduce the definition clause F and we add it to *Defs*.

The introduction of the definition F might seem to be the best choice in the sense that it contains exactly the head variables which are actually needed for folding clause C. However, (variants of) `B` may occur also in some other clauses to be folded. Thus, if we directly introduce definitions whose heads contain linking variables only, we run the risk of generating several definitions with the same atom in the body and different sets of variables in the head (modulo renaming).

In order to keep the number of definitions low (and this will often improve the ability of proving program correctness), instead of introducing multiple definitions containing the same atom in the body, by applying the NLR strategy, we merge them in a single definition whose set of head variables is the union of the head variables occurring in the merged definitions (modulo renaming).

The NLR strategy terminates when all clauses in *Defs* have been unfolded and no new definition need to be introduced for folding.

```
1'. unsafe:- X1>=0, Y2=<0, newp3(X1,Y2).
2'. newp3(X1,Z2):- Z1=X1+1, newp4(X1,Z1,Z2).
3'. newp4(X1,Z1,Z2):- Z1=<9, Z3=Z1+1,
            newp4(X1,Z3,Z2).
4'. newp4(X1,Z1,Z1):- Z1>=10.
```

*P*2: Verification Conditions obtained by NLR

```
1". unsafe:- X1>=0, Y2=<0, newp3(X1,Y2).
2". newp3(X1,Z2):- Z1=X1+1, newp4(Z1,Z2).
3". newp4(Z1,Z2):- Z1=<9, Z3=Z1+1,
        newp4(Z3,Z2).
4". newp4(Z1,Z1):- Z1>=10.
```

*P*3: Verification Conditions obtained by cFAR

Figure 2: Program *P*2 and Program *P*3 are the Verification Conditions VCs obtained by applying the NLR strategy and the cFAR algorithm, respectively.

Theorem 1 (Termination and Correctness of the NLR Strategy) *Given any CLP program Prog, the NLR strategy terminates and produces a CLP program $Prog'$ such that* `unsafe` $\in \mathscr{M}(Prog)$ *holds iff* `unsafe` $\in \mathscr{M}(Prog')$ *holds.*

2. Constrained FAR Algorithm (cFAR). Now we present an extension to constraint logic programs of the FAR algorithm presented in [9] for removing redundant arguments from logic programs. This extension will be called constrained FAR algorithm, or cFAR, for short. The objective of the FAR algorithm is to remove arguments that are not actually used during any computation of the program at hand. Indeed, it has been shown in [8] that the FAR algorithm (and thus, also the cFAR algorithm) can be seen as a generalization of the liveness analysis.

In Figure 2 we show the effect of applying the cFAR algorithm to the CLP program *P*2 obtained by the NLR strategy. The output of the algorithm is the CLP program *P*3. Note that in program *P*3 the predicate symbol `newp4` denotes a different relation with respect to the one in program *P*2, because in *P*3 it has arity 2 and not 3.

In order to define the constrained FAR algorithm we need to introduce some preliminary notions, some of which have been adapted from [9].

Definition 2 (Erasure, Erased Atom, Erased Clause, Erased Program) (i) *An* erasure *is a set of pairs each of which is of the form* $(\mathtt{p}, k)$*, where* `p` *is a predicate symbol of arity n and* $1 \le k \le n$.
(ii) *Given an erasure E and an atom* `A` *whose predicate symbol is* `p`*, the* erased atom $\mathtt{A}|_E$ *is obtained by dropping all the arguments that occur at position k, for some* $(\mathtt{p}, k) \in E$.
(iii) *Given an erasure E and a clause C* (*respectively, a CLP program Prog*)*, the* erased clause $C|_E$ (*respectively, the* erased program $Prog|_E$) *is obtained by replacing all atoms* `A` *in C* (*respectively, in Prog*) *by* $\mathtt{A}|_E$.

In order to avoid the risk of collisions between predicate symbols after erasing some arguments, we assume that *Prog* does not contain identical predicate symbols with different arity.

Obviously, we are interested in removing redundant arguments without altering the semantics of the original program, in the sense captured by the following definition.

Definition 3 (Correctness of Erasure) *An erasure E is correct for a program Prog if, for all atoms* `A`*, we have that*: $\mathtt{A} \in \mathscr{M}(Prog)$ *iff* $\mathtt{A}|_E \in \mathscr{M}(Prog|_E)$.

Since we are dealing with constraint logic programs, the notion of multiple occurrences of a variable which is used in the original formulation of FAR [9], needs to be generalized as follows. In this paper we assume that a constraint is a conjunction of h (≥ 0) atomic constraints in the theory $\mathscr{A}$ of the linear integer arithmetics with integer arrays.

Definition 4 (Variable Constrained to Another Variable) *Given two variables* $\mathtt{X}$ *and* $\mathtt{Y}$ *and a constraint* $\mathtt{c}$ *of the form* $\mathtt{c_1} \wedge \ldots \wedge \mathtt{c_h}$*, we say that* $\mathtt{X}$ *is* constrained to $\mathtt{Y}$ (*in* $\mathtt{c}$) *if there exists* $\mathtt{c_j}$*, with* $1 \leq j \leq h$*, such that either* (i) $\{\mathtt{X},\mathtt{Y}\} \subseteq vars(\mathtt{c_j})$*, or* (ii) *there exists a variable* $\mathtt{Z}$ *such that* (ii.1) $\{\mathtt{X},\mathtt{Z}\} \subseteq vars(\mathtt{c_j})$ *and* (ii.2) $\mathtt{Z}$ *is constrained to* $\mathtt{Y}$ (*in* $\mathtt{c}$).

Now we are ready to introduce the notion of *safe erasure* that will be used during the application of the constrained FAR algorithm.

Definition 5 (Safe Erasure) *Given a program Prog, an erasure E is a* safe erasure *if, for all* $(\mathtt{p},k) \in E$ *and clauses* $\mathtt{H} :- \mathtt{c},\mathtt{G}$ *in Prog, where* $\mathtt{H}$ *is of the form* $\mathtt{p(X1,...,Xn)}$ *and* $\mathtt{c}$ *is of the form* $\mathtt{c_1} \wedge \ldots \wedge \mathtt{c_h}$*, we have that:* (i) $\mathtt{X_k}$ *is a variable and* $\mathscr{A} \models \forall \mathtt{X_k}. \exists \mathtt{Y1}, \ldots, \mathtt{Ym}.\ \mathtt{c}$*, with* $\{\mathtt{Y1},\ldots,\mathtt{Ym}\} = vars(\mathtt{c}) - \{\mathtt{X_k}\}$*,* (ii) $\mathtt{X_k}$ *is not constrained to any other variable occurring in* $\mathtt{H}$*, and* (iii) $\mathtt{X_k}$ *is not constrained to any variable occurring in* $\mathtt{G}|_E$*.*

Similarly to what has been done in [9], it can be shown that if an erasure E is safe, then it is also correct.

The cFAR algorithm takes as input a CLP program *Prog*, computes a safe erasure E, and produces as output the program $Prog|_E$. The algorithm starts off by initializing the current erasure E to the *full erasure*, that is, the set of all pairs $(\mathtt{p},k)$, where p is a predicate of arity n occurring in *Prog* and $1 \leq k \leq n$. Then, while E contains a pair $(\mathtt{p},k)$ such that one of the conditions of Definition 5 is not satisfied, the pair $(\mathtt{p},k)$ is removed from E. The algorithm terminates when it is no longer possible to remove a pair $(\mathtt{p},k)$ from E, and thus E is a safe erasure.

The cFAR algorithm terminates and preserves the least-model semantics, as stated by the following theorem.

Theorem 2 (Termination and Correctness of the cFAR Algorithm) *Given any CLP program Prog, the cFAR algorithm terminates and produces a CLP program* $Prog|_E$ *such that* $\mathtt{unsafe} \in \mathscr{M}(Prog)$ *iff* $\mathtt{unsafe} \in \mathscr{M}(Prog|_E)$*.*

Finally, we would like to note that, even if the objectives of the NLR and cFAR transformations are similar, they work in a different way. While cFAR is goal independent, NLR starts from the predicate `unsafe` and proceeds by unfolding in a goal directed fashion, similarly to *redundant argument filtering* [9]. It can be shown that, in general, the NLR and cFAR transformations have incomparable effects.

3 Experimental evaluation

We have used the VeriMAP transformation and verification system [2, 3] for evaluating the techniques presented in this paper. We have considered 320 verification problems for C programs (227 of which were safe and the remaining 93 were unsafe). We have applied the VCG strategy for generating the Verification Conditions VCs using a multi-step semantics [4]. The C programs and the VCs we have generated are available at: `http://map.uniroma2.it/vcgen`. Then, we have checked the satisfiability of the VCs by giving them as input to the Z3 Horn solver [10] using default options[1] and the PDR engine. Finally, we have applied the NLR and cFAR transformations presented in Section 2 to evaluate the effect of these transformations in terms of efficiency and efficacy in the program verification tasks considered.

[1]Note that Z3, by default, runs the *slice* transformation for reducing the number of variables in the signature of a predicate.

		VCG ; Z3	VCG ; NLR ; Z3	VCG ; NLR ; cFAR ; Z3
c	Correct answers	196	7	9
s	- safe problems	144	3	7
u	- unsafe problems	52	4	2
to	Timeouts	124	117	108
n	Total problems	320	124	117
t_{VCG}	VCG time	40.65	20.48	4.57
t_{NLR}	NLR time	–	58.39	9.53
t_{cFAR}	cFAR time	–	–	304.84
st	Z3 solving time	2704.95	988.15	649.56
tt	Total time	2745.60	1067.02	968.50
at	Average time	14.01	152.43	107.61

Table 1: Verification results obtained by using Z3 on the output generated by applying VCG and the auxiliary transformations NLR and cFAR. The timeout limit time is 300 seconds. Times are in seconds.

Improving effectiveness of solving. In Table 1 we show the experimental results obtained by using VeriMAP and Z3. Column 'VCG ; Z3' reports the results obtained by applying the VCG strategy and then the Z3 solver. Column 'VCG ; NLR ; Z3' reports the results obtained, for the problems *not solved* by 'VCG ; Z3', by applying VCG, followed by the NLR transformation, and then Z3. Column 'VCG ; NLR ; cFAR ; Z3' reports the results obtained, for the problems *not solved* by 'VCG ; NLR ; Z3', by applying VCG, followed by NLR, then cFAR, and finally Z3. Lines (t_{VCG}), (t_{NLR}), and (t_{cFAR}) report the time taken by the execution of the VCG, NLR, and cFAR transformations, respectively, to produce the verification conditions for which Z3 was able to return the correct answers (that is, to show the satisfiability or the unsatisfiability of the clauses). Line (*st*) reports the time taken by Z3 to produce the correct answers.

The NLR transformation enables Z3 to prove 7 additional verification problems. In particular, it allows Z3 to prove the program `ntdrvsimpl-cdaudio_simpl1_unsafeil.c`, which is the largest program in the benchmark set (2.1 KLOC). Concerning the time required for executing the NLR transformation in this example, we want to point out that this program takes 91% of the total NLR time (t_{NLR}), that is 53.04 seconds. Therefore, the remaining 6 programs only require 5.35 seconds to be transformed. The cFAR transformation allows Z3 to prove 9 additional verification problems. In this case, about 89% of the total cFAR time (t_{cFAR}), that is, 271.62 seconds, is required for specializing two programs whose size is about 1 KLOC each, namely `ntdrvsimpl-diskperf_simpl1_safeil.c` (98.82 seconds) and `ntdrvsimpl-floppy_simpl3_safeil.c` (172.80 seconds).

4 Conclusions

In this paper we have shown that the effectiveness of Horn clause solvers for proving the satisfiability of VCs can be improved by the use of program transformations that remove unnecessary variables.

As future work, we would like to investigate in more depth how the structure of the VCs influences the heuristics adopted by Horn solvers. Hopefully, this would allow us to tune the VCG strategy for generating VCs that are easier to be proved. Also, it would be interesting to study the effect of the NLR and cFAR transformations on the VCs generated by other tools like, for example, SeaHorn [7].

Acknowledgements

The authors would like to thank the GNCS - INdAM for the Research Grant 2016 "Verifica Automatica di Proprietà Relazionali di Programmi".

References

[1] E. Albert, M. Gómez-Zamalloa, L. Hubert & G. Puebla (2007): *Verification of Java Bytecode Using Analysis and Transformation of Logic Programs*. In M. Hanus, editor: *Practical Aspects of Declarative Languages*, Lecture Notes in Computer Science 4354, Springer, pp. 124–139, doi:10.1007/978-3-540-69611-7_8.

[2] E. De Angelis, F. Fioravanti, A. Pettorossi & M. Proietti (2014): *Program Verification via Iterated Specialization*. *Science of Computer Programming* 95, Part 2, pp. 149–175, doi:10.1016/j.scico.2014.05.017.

[3] E. De Angelis, F. Fioravanti, A. Pettorossi & M. Proietti (2014): *VeriMAP: A Tool for Verifying Programs through Transformations*. In: *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '14*, Lecture Notes in Computer Science 8413, Springer, pp. 568–574, doi:10.1007/978-3-642-54862-8_47. Available at: http://www.map.uniroma2.it/VeriMAP.

[4] E. De Angelis, F. Fioravanti, A. Pettorossi & M. Proietti (2015): *Semantics-based generation of verification conditions by program specialization*. In: *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming, Siena, Italy, July 14-16, 2015*, ACM, pp. 91–102, doi:10.1145/2790449.2790529.

[5] S. Etalle & M. Gabbrielli (1996): *Transformations of CLP Modules*. *Theoretical Computer Science* 166, pp. 101–146, doi:10.1016/0304-3975(95)00148-4.

[6] J. P. Gallagher & B. Kafle (2014): *Analysis and Transformation Tools for Constrained Horn Clause Verification*. *Theory and Practice of Logic Programming* 14(4-5), pp. 90–101. Supplementary Materials.

[7] A. Gurfinkel, T. Kahsai, A. Komuravelli & J.A. Navas (2015): *The SeaHorn Verification Framework*. In: *Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015*, Springer, pp. 343–361, doi:10.1007/978-3-319-21690-4_20.

[8] K. S. Henriksen & J. P. Gallagher (2006): *Abstract Interpretation of PIC Programs through Logic Programming*. In: *Proceedings of the 6th IEEE International Workshop on Source Code Analysis and Manipulation, SCAM '06*, pp. 103–179, doi:10.1109/SCAM.2006.1.

[9] M. Leuschel & M. H. Sørensen (1996): *Redundant Argument Filtering of Logic Programs*. In J. Gallagher, editor: *Logic Program Synthesis and Transformation, Proceedings LOPSTR '96, Stockholm, Sweden*, Lecture Notes in Computer Science 1207, Springer-Verlag, pp. 83–103, doi:10.1007/3-540-62718-9_6.

[10] L. M. de Moura & N. Bjørner (2008): *Z3: An Efficient SMT Solver*. In: *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '08*, Lecture Notes in Computer Science 4963, Springer, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[11] J. C. Peralta, J. P. Gallagher & H. Saglam (1998): *Analysis of Imperative Programs through Analysis of Constraint Logic Programs*. In G. Levi, editor: *Proceedings of the 5th International Symposium on Static Analysis, SAS '98*, Lecture Notes in Computer Science 1503, Springer, pp. 246–261, doi:10.1007/3-540-49727-7_15.

[12] M. Proietti & A. Pettorossi (1995): *Unfolding-Definition-Folding, in this Order, for Avoiding Unnecessary Variables in Logic Programs*. *Theoretical Computer Science* 142(1), pp. 89–124, doi:10.1016/0304-3975(94)00227-A.