PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 18/06

Some Strategies for the Automatic Use of Hoare Logic

Juliana Carpes Imperial
Edward Hermann Haeusler

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900
RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 18/06 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena Julho, 2006

Some Strategies for the Automatic Use of Hoare Logic [1]

Juliana Carpes Imperial and Edward Hermann Haeusler

juliana@inf.puc-rio.br, hermann@inf.puc-rio.br

Abstract. This papers aims at showing some strategies to construct a proof of correctness of a program in such way to increase the automation of the process. To achieve this, the proposed strategies reduce drastically the search space and the need of user interaction.

Keywords: Hoare Logic, Program Correctness, Invariants of Loops, Proof Theory

Resumo. Este artigo tem como objetivo mostrar algumas estratégias para construir uma prova de correção de um programa de maneira a tornar o processo mais automático. Para conseguir isso, as estratégias propostas reduzem drasticamente o espaço de busca e a necessidade de interação com o usuário.

Palavras-chave: Cálculo de Hoare, Correção de Programas; Invariantes de *Loops*, Teoria da Prova

[1]Partially funded by CAPES, FAPERJ and CNPq (projects Universal – 471608/03-3 and VAS – 552192/02-3.

In charge for publications:
Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3114-1516 Fax: +55 21 3114-1530
E-mail: bib-di@inf.puc-rio.br
Web site: http://bib-di.inf.puc-rio.br/techreports/

1 Introduction

It is well-known that the task of proving a theorem is a hard one. Propositional theorem proving is CoNP-complete, while first-order theorem proving is undecidable. The former fact by itself is reason enough, unless CoNP = NP, to be desirable the interference of the human-being in the theorem proving process. In addition, the size of the proofs, again an essential problem, unless CoNP = NP, demands the use of automatic help or guidance. A good approach to the problem is to use a set of heuristics that, in most of the cases, minimizes the complexity of the whole process, namely, the size of the proof, the number of lemmas to be used, the number of times a human interference is needed and the total time to finish the proof. When dealing with first-order logic, to the addressed problem is added the task of stop, by human interference of course, the whole proving procedure since the theorem prover may be faced with a non-theorem. The propositional proof-procedure can as well use a kind of pruning or aborting heuristics, but in this situation a complete one, in the logical sense. When dealing with program's correctness proofs, the worst of the worlds is faced, as the first-order data-types theories involved carry the first-order and, obviously, the propositional problems. Furthermore, the very task of proving non-trivial properties about programs is undecidable (Rice's theorem) [1]. Concerning the specific case of Hoare Logic, the task of automatically finding invariants for the loops is, in general, undecidable. It is good to remind that all of the mentioned problems also happen if equational logic is used instead of first-order logic for the data-type specification.

Correctness proofs for programs, as opposed to other formal specifications, gained a new importance with the development of the proof-carrying code technology [3, 4]. The present work aims to show that one possible use of a semi-automatic Hoare Logic prover is to have PCC at a high level programming language. Faced with the facts mentioned in the previous paragraph, the aim is the design of strategies for a Hoare Logic prover in order to minimize the human interference, the number of calls to a first-order theorem prover and the time to conclude the proof. In this work, the size of the proof is not considered to be the most important issue.

Proof-theory is a well established subject. It has shown not only the complexity degree of some theories, by pointing out (upper-bound) ordinals for proving the consistency of each of them, but also has raised some quite important insights as the Curry-Howard isomorphism and the automatic proof construction from general observations on the format of a cut-free, or normal proof. Most of the automatic theorem provers conception are based at least on that. Just in this situation, proof-theoretical considerations on the kind of a constructed proof raised a lot of (complete) heuristics for theorem provers. The present work follows this approach when designing the strategies, namely, a strategy can be based on proof-theoretical considerations on the format of a possible proof and on some conservative transformations on this format.

In the next section it is shown how from proof-theoretical considerations some heuristics ideas are posed. Section 3 shows in detail each one of the strategies for Hoare Logic. Section 4 concerns the completeness and correctness of the presented strategies. Section 5 remarks some problems related with the arrays (data-type) and how they cross the bridge between the data-type specification and the language specification.

2 Normalization and Heuristics

In the deductive system of classical logic named natural deduction it is possible to normalize a proof using successive reductions [7, 8, 11, 14]. It is also possible to produce a normal proof directly by using heuristics. Therefore, the fact that there exists normalization for this logic makes the (automatic) construction of proofs easier. Moreover, this conclusion holds to any logic in which there is a normalization process [8].

What is intended in this work is to apply the same idea to program verification using Hoare Logic [6]. That is, it is desirable the creation of patterns to generate the proofs, so that the search space can become smaller. Consequently, they are constructed in a more efficient way.

In some inference rules of Hoare Logic, shown in the figure 1, there are first-order sentences (logical implications) which must be demonstrated, such as the *if-then* rule and the weakening and strengthening rules. Since it is syntactically possible to have many proofs of programs that satisfy the Hoare Logic rules but with some sentences which cannot be demonstrated, it is worth trying to find proofs with valid sentences avoiding the use of a theorem prover to demonstrate them if the program satisfies the specification given by a pair of pre-condition and post-condition [2].

It is desirable to obtain one proof with only demonstrable sentences without using a theorem prover since demonstrating the sentences is often inefficient. Furthermore, trying to do this to invalid sentences might lead the theorem prover to an infinite loop (after all, in most cases, knowing whether a first-order sentence is valid or not is an undecidable process).

So, when there is a program with a post-condition which is the result of the execution of this program with a certain pre-condition, a formal program verifier may answer different proofs in which their sentence sets can have all of them demonstrable or not if there is no guarantee that all of them are valid. Motivated by what is discussed above, it is desirable to analyze efficient strategies for a formal program verifier using Hoare Logic, so that it constructs only one proof with all its sentences being demonstrable, unless the program does not behave as desired, which means that the program does not satisfy its specification given by its pre-condition and post-condition.

The next section shows a set of strategies to derive a proof of correctness as discussed above in the case that the program does not deal with arrays (the problem with arrays is described later in this text). If the program does not satisfy its specification, there will be at least one invalid sentence.

3 The Strategies

A proof using Hoare Logic can be done from from the left to the right (that is, from the first command to the last one), from the right to the left (that is, from the last command to the first one) or using both of them. The Hoare Logic rules used are the ones below [6]:

3.1 From the Right to the Left

Considering the fact that in this work the loops' invariants are given by the user (finding them is an undecidable task) and if the program does not have arrays, it is straightforward

$$\frac{\{P \wedge B\}\ C_1\ \{Q\} \qquad \{P \wedge \neg B\}\ C_2\ \{Q\}}{\{P\}\ if\ B\ then\ C_1\ else\ C_2\ fi\ \{Q\}} \qquad \frac{\{P \wedge B\}\ C\ \{Q\} \qquad P \wedge \neg B \rightarrow Q}{\{P\}\ if\ B\ then\ C\ fi\ \{Q\}}$$

$$\frac{}{\{P(V/E)\}\ V := E\ \{P\}} \qquad \frac{\{P\}\ C_1\ \{R\} \qquad \{R\}\ C_2\ \{Q\}}{\{P\}\ C_1; C_2\ \{Q\}} \qquad \frac{P \rightarrow R \qquad \{R\}\ C\ \{Q\}}{\{P\}\ C\ \{Q\}}$$

$$\frac{}{\{P\}\ skip\ \{P\}} \qquad \frac{\{P \wedge B\}\ C\ \{P\}}{\{P\}\ while\ B\ do\ C\ od\ \{P \wedge \neg B\}} \qquad \frac{\{P\}\ C\ \{R\} \qquad R \rightarrow Q}{\{P\}\ C\ \{Q\}}$$

Figura 1: Hoare Logic Rules

to build the correctness proof from the right to the left. Obviously, the post-condition must be known to construct the proof this way, which is very similar to the Dijkstra's Weakest Pre-Condition [13].

In this process, if a pre-condition is found (that is, it is already determined), this pre-condition is weakened to the new one found when proving the correctness of the program. This can happen when the first command of the program is achieved (if the program's pre-condition is given), when the first command of the body of a loop is achived (its pre-condition is the invariant with the loop's test) or when the post-condition of a loop is achived (its invariant with the negation of the loop's test is the pre-condition of the command after the loop). The only places where weakenings are used are those and the ones to prove the if commands correctness.

To the following kinds of commands, it is only necessary to know how to apply the suitable Hoare Logic rule to prove their correctness: the sequence of commands (it is only necessary to use the pre-condition of a command as the post-condition of the previous one or weaken a loop post-condition if the previous command is a *while*), the *skip* command (the one that does nothing), the *while* command (the invariant, which is given by the user, must be known) and the attribution command (the pre-condition can be determined by substituting variables, as indicated by its rule) [2].

Nevertheless, it is not trivial to do what is explained above with the if rules, as the post-condition of this command may depend on its test, that is, it is possible that the assertion which is true after the if execution depends on the result of its test evaluation. To prove the *if-then-else* correctness, one must do the following [13]:

$$\frac{\dfrac{P \wedge B \rightarrow P_1 \qquad \{P_1\}\ C_1\ \{Q\}}{\{P \wedge B\}\ C_1\ \{Q\}} \qquad \dfrac{P \wedge \neg B \rightarrow P_2 \qquad \{P_2\}\ C_2\ \{Q\}}{\{P \wedge \neg B\}\ C_2\ \{Q\}}}{\{P\}\ if\ B\ then\ C_1\ else\ C_2\ fi\ \{Q\}}$$

where $P = (B \rightarrow P_1) \wedge (\neg B \rightarrow P_2)$ or $P = (B \wedge P_1) \vee (\neg B \wedge P_2)$ and P_1 is C_1's pre-condition and P_2 is C_2's pre-condition, which were found when proving their correctness from the right to the left.

In a similar way, the correctness of *if-then* can be proved, by putting the *skip* command instead of C_2 in the *if-then-else* proof.

3.2 From the Left to the Right

If the post-condition is not known, the proof cannot be done from the right to the left. In this situation, the new post-condition found after proving the correctness of each command must be as strong as possible, because a significant post-condition is desirable (*true* is always a valid post-condition, for instance, but it does not mean much).

One important thing is that, if there is a *while* command (and no arrays), its pre-condition and post-condition are known because the loops' invariants are given by the user. So, when doing the correctness proof for a program without a post-condition, only the commands after the loop need to have the correctness proof built from the left to the right. The commands inside the loop and the ones before it can have its correctness proved using the other way around. The whole program is not proved correct from the left to the right because the proofs done this way are usually much larger, as it can be seen later.

It is shown below how to find the strongest post-condition for the attribution and the *if* commands. The other commands are omitted: to the *skip* command it is only necessary to use its Hoare Logic rule, the sequence of commands is proved correct by using the first command's post-condition as the second command's pre-condition and *while* is always proved correct using the other way around because its post-condition is already known [2].

In what follows, it is still assumed that the program does not deal with arrays. In addition to this, when $P(x)$ is used, it means that the variable x, which is the left hand side of the attribution, does occur in $P(x)$. On the other hand, the use of P means that x does not occur in P. The same can be said about the expressions $E(x)$ and E, which are the new values of x. So, there are four cases to be analyzed. Moreover, Q is the unknown attribution post-condition.

The four cases are the following:

$\{\mathbf{P}\}\ \mathbf{x} := \mathbf{E}\ \{\mathbf{Q}\}$. In this situation, since P does not depend on x, the strongest post-condition is $Q = (P \wedge x = E)$. Therefore, supposing that $P \neq true$ and $P \neq (E = E)$, the verification proof is:

$$\frac{P \rightarrow (P \wedge E = E) \qquad \{P \wedge E = E\}\ x := E\ \{P \wedge x = E\}}{\{P\}\ x := E\ \{P \wedge x = E\}}$$

If $P = true$, the proof can be simplified, since the P's which are not alone in the assertions can be omitted. In addition to this, if $P = (E = E)$, not only can the first simplification be done, but the weakening also becomes not necessary.

$\{\mathbf{P}\}\ \mathbf{x} := \mathbf{E}(\mathbf{x})\ \{\mathbf{Q}\}$. Since P does not have any information related to x, it is not always possible to know the new value of x after the attribution, since $E(x)$ depends on x. In some cases, it is not possible even to know any property concerning x, so the post-condition will be just P. In other situations, $E(x)$ may have operations whose result does not depend on x. $E(x) = 0 \times x$ and $E(x) = x \vee true$ are examples of this kind of situation. If this happen, the strategy that must be applied is the one related to the case $\{P\}\ x := E\ \{Q\}$.

In other cases, although it is impossible to know the value of x in the post-condition, it is feasible to know some property about it. For instance, if $P = true$ and $E(x) = x \times x$, it can be said that $Q = (x \geq 0)$.

Consequently, when it is necessary to prove an attribution correctness in this situation, it can be assumed that x already have a value before the attribution, which is unknown.

Then, after the command execution, its actual value depends on its last value. Since it exists, the existential quantifier will be used to capture this idea. Therefore, $Q = (P \wedge \exists y(x = E(y)))$, where $y \notin FV(E)$. So,

$$\frac{P \rightarrow (P \wedge \exists y(E(x) = E(y)))\quad \{P \wedge \exists y(E(x) = E(y))\}\ x := E(x)\ \{P \wedge \exists y(x = E(y))\}}{\{P\}\ x := E(x)\ \{P \wedge \exists y(x = E(y))\}}$$

If $P = true$ or $P = \exists y(E(x) = E(y))$, the simplifications described in the last case can also be done in the proof above.

Obviously, the post-condition of the proof above can be weakened, so that it can be equal to the ones of the examples cited above.

$\{\mathbf{P(x)}\}\ \mathbf{x := E}\ \{\mathbf{Q}\}$. Supposing that $P(x)$ does not have contradictions (if it has contradictions, it derives $\perp$ and hence anything is a pre-condition and a post-condition) and that the value of x before the attribution can be found when analyzing the pre-condition, it can be said that $Q = (P(a) \wedge x = E)$, if the value of x is equal to a. That is, there is a proof, using logic and the theory of the data-types involved, which proves that $x = a$ and, consequently, that $P(x) = P(a)$. So, the proof is like that:

$$\frac{P(x) \rightarrow (P(a) \wedge E = E) \qquad \{P(a) \wedge E = E\}\ x := E\ \{P(a) \wedge x = E\}}{\{P(x)\}\ x := E\ \{P(a) \wedge x = E\}}$$

If $P(x)$ only has information concerning the value of x, the proof above can be simplified by removing the $P(a)$'s.

However, if the value of x before the attribution cannot be found or it is very difficult to know it, something similar to what is done in the last situation can be done, that is, the existential quantifier can be introduced, once x has an initial value, although it is unknown. It must be remembered that the bound variable introduced cannot be free in P. Therefore,

$$\frac{P(x) \rightarrow (\exists a P(a) \wedge E = E) \qquad \{\exists a P(a) \wedge E = E\}\ x := E\ \{\exists a P(a) \wedge x = E\}}{\{P(x)\}\ x := E\ \{\exists a P(a) \wedge x = E\}}$$

It can be noticed that the proofs above are very similar, except that the last one has an existential quantifier. This is so because the other one is a simplification of the last one.

$\{\mathbf{P(x)}\}\ \mathbf{x := E(x)}\ \{\mathbf{Q}\}$. In the proof below, the suppositions that must be done are the same of the first proof of the last case, with the restriction that a cannot have bound variables because a is used in E. In this situation, $Q = (P(a) \wedge x = E(a))$. So,

$$\frac{P(x) \rightarrow (P(a) \wedge E(x) = E(a))\quad \{P(a) \wedge E(x) = E(a)\}\ x := E(x)\ \{P(a) \wedge x = E(a)\}}{\{P(x)\}\ x := E(x)\ \{P(a) \wedge x = E(a)\}}$$

The same simplification done in the last case can also be done to the proof above.

Again, if there is trouble in finding the value of x, the existential quantifier must be introduced:

$$\frac{P(x) \rightarrow \exists a(P(a) \wedge E(x) = E(a))\quad \{\exists a(P(a) \wedge E(x) = E(a))\}\ x := E(x)\ \{\exists a(P(a) \wedge x = E(a))\}}{\{P(x)\}\ x := E(x)\ \{\exists a(P(a) \wedge x = E(a))\}}$$

In the last two cases, the program verifier which was implemented is able to find the value of x when $P(x)$ is a set of conjunctions of the form $A_1 \wedge ... \wedge A_n$, where one of the A_i is of the form $x = a$, $a = x$, x or $\neg x$. In the last two situations, x is a boolean variable which has the values $true$ and $false$, respectively.

It can be noticed that in the last three patterns, it is not always possible to find a "good" post-condition, that is, the one that would be obtained if the formal proof was done manually. If so, the existential quantifier would not be needed. Nevertheless, to always obtain a "good" post-condition, a complete and consistent arithmetic module should be part of the program verifier, which is impossible [4, 5].

One problem with the adopted approach is that the sentences and assertions can become much bigger than the ones which would be obtained if the proof was done manually. This turns the verification of the sentences more difficult to a theorem prover. On the other hand, the proof generation is done automatically, except for the verification of the sentences. This is a need for proof-carrying code purposes. For a general purpose program verifier the choice may be another.

To finish off the strategies, it must be shown how to prove the if correctness from the left to the right. Depending on the value of its test, the execution of this command can follow two different paths. When the pre-condition P has enough information to evaluate the test B, the pre-condition of one of the paths will have a contradiction (that is, one between $P \wedge B$ and $P \wedge \neg B$ will be false). Using the strategies already shown, this contradiction is taken to the end of the path, because it will not be removed. Therefore, it can be said that the if post-condition is the disjunction of the post-conditions of the two paths. As one of them is false and $A \vee \bot \equiv A$, the desired result is obtained.

When P does not have enough information to evaluate B, the disjunction of the post-conditions of the paths is indeed the if's post-condition. Below it can be seen how to prove the correctness of the *if-then-else* command (to prove the correctness of the *if-then*, it is just necessary to put the *skip* command instead of C_2):

$$\dfrac{\dfrac{\{P \wedge B\}\ C_1\ \{Q\} \qquad Q \rightarrow Q \vee R}{\{P \wedge B\}\ C_1\ \{Q \vee R\}} \qquad \dfrac{\{P \wedge \neg B\}\ C_2\ \{R\} \qquad R \rightarrow Q \vee R}{\{P \wedge \neg B\}\ C_2\ \{Q \vee R\}}}{\{P\}\ if\ B\ then\ C_1\ else\ C_2\ fi\ \{Q \vee R\}}$$

where Q is C_1's post-condition and R is C_2's post-condition, which were found when proving their correctness from the left to the right.

4 Other Considerations, Correctness and Completeness

When proving the correctness of a program from the right to the left, the process can be considered complete because it is based in a well-known process, the Dijkstra's Weakest Pre-Condition [13]. So if there is already a pre-condition to the program which reflects the program's behaviour together with its post-condition, it will always be possible to weaken the old pre-condition to the one found by proving the program correctness from the right to the left.

Moreover, the process is correct because it obeys the Hoare Logic rules and the sentences which appeared when proving the if commands correctness can be proved using Natural Deduction or Sequent Calculus.

Since no relevant information is discarded from the pre-condition or from the post-condition when proving a program correctness from the left to the right, and considering what is done when showing the strategies to prove the correctness of a program this way, it can be concluded that:

Lemma 1 *The strategies for the attribution produce the strongest post-condition.*

Lemma 2 *The strategies for the if commands produce the strongest post-condition.*

When using all the commands together in a program, it can be easily proved that:

Theorem 1 *Using the strategies for proving the correctness from the left to the right always produce the strongest post-condition for the program.*

Therefore, if there is a post-condition for the program not used to prove its correctness, which reflects the program's behaviour together with its pre-condition, it can be obtained by weakening the post-condition to the old one.

Furthermore, this process is also correct because it obeys the Hoare Logic rules and the sentences which appeared when proving the if and attribution commands correctness can be proved.

One thing that can be said about the strategies is that when using them it is only possible to have one proof of correctness. When proving the correctness of a command manually, one can weaken the pre-condition, strengthen the post-condition (if they are known) or apply a rule to the command being verified. However, when using the strategies, the correctness proof is constructed in a deterministic way, which reduces drastically the search space. It cannot be said that forcing a way will remove a valid proof because the weakenings can be put upwards or downwards the proof if they are not done to a *while* command [2]. Below it is shown how to put the weakenings upwards in the *if-then-else* command. A similar process can be done to the other Hoare Logic rules that are not axioms, except for the *while* command. It can be easily proved that the first proof is equivalent to the second and the third to the last one:

5 Problem with Arrays

The strategies of proof construction shown above turns out the proving of a program correctness automatic, except for the verification of sentences. However, it is known that they are valid if the program satisfies its specification. So, the proof will be correct unless the program does not satisfy it.

Nevertheless, when arrays are included in the language used by the program verifier, this does not seem to be automatically done anymore. Unfortunately, when proving the attributions correctness, the strategies shown above might not behave properly with arrays. The problem occurs when the variable which receives a new value in an attribution is a position of an array. So, the heuristics can be used if an array position does not appear in the left hand side of the attribution. Moreover, they can also be used if the pre-condition (when proving the correctness from the left to the right) or the post-condition (when proving the correctness from the right to the left) does not describe properties concerning the whole array.

$$\dfrac{\dfrac{\{R \wedge B\}\ C_1\ \{Q\} \qquad \{R \wedge \neg B\}\ C_2\ \{Q\}}{P \rightarrow R \qquad \{R\}\ if\ B\ then\ C_1\ else\ C_2\ fi\ \{Q\}}}{\{P\}\ if\ B\ then\ C_1\ else\ C_2\ fi\ \{Q\}}$$

$$\dfrac{\dfrac{P \wedge B \rightarrow R \wedge B \qquad \{R \wedge B\}\ C_1\ \{Q\}}{\{P \wedge B\}\ C_1\ \{Q\}} \quad \dfrac{P \wedge \neg B \rightarrow R \wedge \neg B \qquad \{R \wedge \neg B\}\ C_2\ \{Q\}}{\{P \wedge \neg B\}\ C_2\ \{Q\}}}{\{P\}\ if\ B\ then\ C_1\ else\ C_2\ fi\ \{Q\}}$$

$$\dfrac{\dfrac{\{P \wedge B\}\ C_1\ \{R\} \qquad \{P \wedge \neg B\}\ C_2\ \{R\}}{\{P\}\ if\ B\ then\ C_1\ else\ C_2\ fi\ \{R\}} \qquad R \rightarrow Q}{\{P\}\ if\ B\ then\ C_1\ else\ C_2\ fi\ \{Q\}}$$

$$\dfrac{\dfrac{\{P \wedge B\}\ C_1\ \{R\} \qquad R \rightarrow Q}{\{P \wedge B\}\ C_1\ \{Q\}} \quad \dfrac{\{P \wedge \neg B\}\ C_2\ \{R\} \qquad R \rightarrow Q}{\{P \wedge \neg B\}\ C_2\ \{Q\}}}{\{P\}\ if\ B\ then\ C_1\ else\ C_2\ fi\ \{Q\}}$$

Figura 2: Putting the weakening and strengthening upwards in the *if-then-else* rule

The matter is that the properties describing arrays are usually related with all its elements and not with a specific one. So, a theorem prover or a human being is needed to make certain inferences relating indexes, the array position being modified and properties concerning the whole array.

When proving the correctness of an attribution from the right to the left, the pre-condition will be the same as the post-condition if the assertion in the post-condition is concerning the entire array. For instance, in the proof below, the proof cannot be started from the post-condition because the pre-condition will be the same as the post-condition, which will generate an invalid sentence, which is $\forall j(j < i \rightarrow a[j] = j) \rightarrow \forall j(j < i + 1 \rightarrow a[j] = j)$. The right way of proving its correctness is shown below:

$$\dfrac{P \rightarrow P \wedge i = i \qquad \dfrac{\{P \wedge i = i\}\ a[i] := i\ \{P \wedge a[i] = i\} \qquad P \wedge a[i] = i \rightarrow Q}{\{P \wedge i = i\}\ a[i] := i\ \{Q\}}}{\{P\}\ a[i] := i\ \{Q\}}$$

where $P = \forall j(j < i \rightarrow a[j] = j)$ and $Q = \forall j(j < i + 1 \rightarrow a[j] = j)$.

Moreover, when trying to find the strongest post-condition of an attribution, inserting the existential can also cause trouble, because the property of the position of the array being modified cannot be easily found in the pre-condition. In addition, if it is discovered, the new pre-condition, which is a weaker version of the old one, is difficult to be obtained, as inferences with the indexes must be done.

Consequently, the solution found in the implementation of the program verifier to validate the strategies is asking the user a pre-condition or a post-condition, depending on which information is needed.

6 Conclusions

The set of strategies proposed here were based on proof-theoretic considerations. Due to intrinsic problems, as for example the closed connection between the array data-type and the programming language constructs, some cases failed to be complete and, as a consequence, a help or guidance from the user is asked. This might be expected, once in the very starting of the process of correctness the user is present. It is reminded that the user must provide the invariants (this can be done by annotations in the code). However, as it is explained in the introduction, any useful environment cannot refuse the human help, not only by reasons of finding a proof, but also because an enormous proof is the best thing an automatic prover can do, assuming $CoNP \neq NP$.

As the work presented here is a practical one, a prototype was build and implemented the strategies here presented. It was implemented in the SWI-Prolog v5.21 and the size of the program is about 3500 code lines. Some experiments, correctness proofs, were done and the general result was quite good. However, it is worthwhile to note that this research was guided by a strong theoretical background since its very beginning. This helped quite a lot in determining the quality of the strategies, concerning their contribution in making the search space smaller and their completeness. Thus, it was presented a completeness argument for the implementation, namely, when a proof for a triple $\{P\}\ C\ \{Q\}$ exists, the prototype shows one and it is known that every logical sentence in it is provable. The correctness of the prototype is a routine task, since it implements each Hoare Logic rule in a faithful way. Again it is stressed that the arguments provided here for the completeness were based on proof-theory approach.

Future steps in this work is the searching to provide a better treatment for arrays, which represent the kind of data-type that uses non-local references, and to provide a reasonable graphic output for the result; currently it is displayed in a file in text format. This last task is important to completely fulfil the interaction with a user.

Referências

[1] MACHTEY, M., YOUNG, P. **An Introduction to the General Theory of Algorithms**. Theory of Computation Series. The Computer Science Library, 1978. 264 p.

[2] IMPERIAL, J. C. **Techniques for the Use of Hoare Logic in PCC**. M. Sc. Dissertation. Departamento de Informática, PUC-Rio, Brazil. August, 14^{th} 2003. 137 p. In Portuguese.

[3] NECULA, G. C. Proof-Carrying Code. In: SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES (POPL'97). Paris, France, January, 1997. Proceedings... p 106 - 119.

[4] APPEL, A. W., FELTEN, E. W., SHAO, Z. Scaling Proof-Carrying Code to Production Compilers and Security Policies. Princeton University and Yale University, January, 1999. 19 p. It can be found in **http://www.cs.princeton.edu/sip/projects/pcc/whitepaper/**. Accessed on August, 07^{th} 2002.

[5] DENTON, W. Gödel's Incompleteness Theorem. July, 02^{nd} 2002. 4 p. It can be found in **http://www.miskatonic.org/godel.html**. Accessed on October, 03^{rd} 2002.

[6] HOARE, C. A. R. An Axiomatic Basis for Computer Programming. Communications of the ACM. October, 1969. p 576 - 580.

[7] MENEZES, P. B., HAEUSLER, E. H. **Category Theory for Computer Science**. Editora Sagra Luzzatto, Instituto de Informática, UFRGS, Porto Alegre, 2001. 324 p. In Portuguese.

[8] DALEN, D. V. **Logic and Structure**. Third Edition. Springer, Berlin, 1997. 217 p.

[9] PAPADIMITRIOU, C. H. **Computational Complexity**. First Edition. Addison-Wesley, November, 1993. 500 p.

[10] TROELSTRA, A. S., SCHWICHTENBERG, H. **Basic Proof Theory**. Cambridge University Press, August, 2002. 430 p.

[11] WAINER, S. S., WALLEN, L. A.: Basic Proof Theory. In: **Proof Theory**. Editors: ACZEL, Peter, SIMMONS, Harold, WAINER, Stanley. Cambridge University Press, 1993. 26 p.

[12] MANNA, Z. **Mathematical Theory of Computation**. Dover Pubns, December, 24^{th} 2003. 464 p.

[13] DIJKSTRA, E. W. **A Discipline of Programming**. Prentice Hall Series in Automatic Computation, Prentice-Hall. 1976

[14] HAEUSLER, E. H. **Proof of Theorems: an Abstract Approach**. phD Thesis. Departamento de Informática, PUC-Rio, Brazil. 1990. In Portuguese.