STRUCTURED PROGRAMMING (1992) 13

Hoare Logic, Executable Specifications, and Logic Programs

Norbert E. Fuchs

Department of Computer Science
University of Zurich
fuchs@ifi.unizh.ch

Starting from Hoare correctness formulae *{P} S {Q}* which define first-order predicates *S* by their pre- and postconditions *P* and *Q*, I formulate logic specifications $S \leftrightarrow P \wedge Q$ for the predicates. Subsequently, I discuss two methods to construct logic programs from logic specifications. First, I simply derive programs as the if-halves of their logic specifications. These programs are concise, readable, but often inefficient and should rather be considered executable specifications. In the second method, I derive more efficient programs by structural induction. These programs are recursive, and – though correct – not obviously logical consequences of the pre- and postconditions of the predicate.

1 Introduction

Recently there has been much interest in constructive methods to derive logic programs from their specifications (e.g. [Bundy et al. 90], [Flener 91], [Fribourg 90], [Deville 90], [Lau, Prestwich 91]).

Bundy and collaborators [Bundy et al. 90] in their Oyster system synthesize programs in first-order logic from non-executable specifications in typed constructive logic. The synthesis uses the proofs-as-programs technique which constructively proves a specification theorem: each proof step is associated with a program construction step, e.g. proof by induction with recursion. The synthesized logic program consists of a set of equivalences called tracts. In a second step, the synthesized logic program is compiled into a Prolog program.

Deville [Deville 90] formulates specifications of logic programs in natural language. In addition to a description of the relation to be specified, a specification contains information on the types and the directionality of the arguments, on possible restrictions, and on side-effects. Different parts of the specification are considered pre- and postconditions of the program. By structural induction and generalization, Deville derives from the specification a logic description of a predicate expressed as equivalences in untyped first-order predicate logic. In a further step, the logic description is transformed into a Prolog program.

In this paper, I present a new method that is based on non-executable logic specifications derived from Hoare correctness formulae for logic programs. Logic specifications specify the same programs, they are equivalent to correctness formulae.

Logic specifications can be used as the starting point for the construction of programs. In fact, I present two construction methods, one leading to executable specifications, the other one to efficient recursive programs.

Both Deville's logic descriptions [Deville 90] and Bundy's tracts [Bundy et al. 90] superficially resemble my logic specifications though their roles are completely different: logic specifications are the starting point for program synthesis, while logic descriptions and tracts are the result of program synthesis.

Hoare correctness formulae for logic programs were also proposed by other authors. Recently, Colussi and Marchiori [Colussi, Marchiori 91] defined an axiomatic semantics for Prolog programs with the specific intention to express Prolog's operational aspects. Colussi's and Marchiori's paper contains references to related earlier work.

In section 2 of this paper, I briefly recall Hoare logic, and in section 3 the relation between specifications and programs in logic. In section 4, I introduce logic specifications, first by means of an example, then formally. I also prove that the programs specified by logic specifications are correct and complete with respect to the least Herbrand model. In section 5, I derive executable specifications from logic specifications, and advocate coroutining as a complete and efficient execution method for executable specifications. In section 6, I briefly define structural induction, and demonstrate that by structural induction efficient programs can be derived from logic specifications. Finally in section 7, I summarize my results.

2 Hoare Logic

In his seminal paper Hoare [Hoare 69] shows how programs can be given an axiomatic basis. Programs in procedural languages acquire a semantics by pre- and postconditions expressed in first-order predicate logic. These pre- and postconditions can be considered as specifications of the programs. Hoare defines correctness formulae

$$\{P\}\ S\ \{Q\}$$

where S is a program in a procedural language, and the precondition P and post–condition Q are logical formulae describing properties of data manipulated by the program S. The correctness formula means that if execution of S is started in a state satisfying P and if the execution terminates then Q will be true. Often the postcondition Q involves both initial and final states.

Defining the semantics of individual statements of a procedural programming language by correctness formulae, and providing correct rules for the composition of statements to programs enables us to construct a verified program from its specifications. Though not actually applicable to realistic programs, this idea has turned out to be extremely fruitful for research on program verification and design, and has led to a large number of publications. Cousot [Cousot 90] gives a recent overview of the field and provides a comprehensive bibliography.

3 Specifications and Programs in Logic

In Hoare correctness formulae, specifications are expressed in logic while programs are expressed in a procedural language. This leads to the unfortunate complication that we have to establish relations between elements of two different languages, concretely that we have to prove that a program meets its specification.

In the context of logic programming, Kowalski [Kowalski 79] points out that logic should not only be considered as a specification language but also as a means to solve the specified problem. Kowalski further emphasizes that expressing specifications and programs in the same language eases the task of verification.

Hoare's axiomatic semantics abstracts from programs in procedural languages to specifications in first-order logic. Combining Hoare's and Kowalski's ideas, I complete the process of abstraction by letting the *S* in Hoare's correctness formula stand for a logic program, i.e. by expressing both the specification and the program in logic.

Kowalski [Kowalski 85] discusses this situation. He states that there is no syntactic difference between a program and its specification. Both specifications and programs can be executed by an interpreter based on automatic deduction – though programs are expected to be more efficient than specifications. Programs can be derived from their specifications by logical deduction, and can thus be considered computationally useful consequences of their specifications.

In the following, I will deduce logic programs from correctness formulae via intermediate expressions which I call logic specifications. Logic specifications capture the essence of correctness formulae and admit at least two different methods for program construction.

4 Logic Specifications

4.1 An Introductory Example

The program `sort(L,SL)` that sorts a list `L` can be specified as

```
{list(L)} sort(L,SL) {permutation(L,SL) ∧ ordered(SL)}
```

One possible implementation of the predicate `sort` is the quicksort algorithm defined by the Prolog program

```
sort([],[]).
sort([L|Ls],SL) :-
  partition(L,Ls,Smaller,Larger),
  sort(Smaller,SortedSmaller),
  sort(Larger,SortedLarger),
  append(SortedSmaller,[L|SortedLarger],SL).
```

To show that `sort` meets its specification we have to prove that `sort` is a logical consequence of its specification. This proof is by no means trivial.

Interestingly, logic programming admits quite another way to derive a program from the given specification. The idea is to define the program as implication of the conjunction of its pre- and postconditions. In the case of `sort` the derivation leads to the Horn clause

$$sort(L,SL) \leftarrow list(L) \wedge permutation(L,SL) \wedge ordered(SL)$$

This program `sort` trivially meets its specification.

I now assume that the elementary predicates `list`, `permutation` and `ordered` are available in executable form, and rewrite the second `sort` as the Prolog program

```
sort(L,SL) :-
  list(L),
  permutation(L,SL),
  ordered(SL).
```

Comparing the derived programs, we find that the first `sort` is an efficient implementation of its specification, but does not obviously meet it, while the second `sort` trivially meets its specification, but – being based on the generate-and-test technique – is very inefficient.

4.2 Formal Definition of Logic Specifications

Let me make the ideas of the preceding sections more precise. I express the relation between a logic program S and its pre- and postconditions P and Q by the correctness formula

$$\{P\}\ S\ \{Q\}$$

where P and Q are logical formulae, and S is an atomic logical formula. Restricting S to an atom means that the program implements a single predicate.

From the correctness formula I derive the *logic specification*

$$S \leftrightarrow P \wedge Q$$

for S. All variables of the logic specification are implicitly universally quantified in front of it.

Logic specifications are consistent if the pre- and postconditions are. Inconsistencies due to multiple assignments are impossible, because logical variables can be bound at most once.

It is interesting to note, that the logic specification of S – enhanced by an equality theory – is equivalent to Clark's completion of S ([Clark 78], [Lloyd 87]) which forms the basis for the declarative semantics of normal logic programs.

4.3 Correctness and Completeness of Logic Specifications

Let the definite logic program *S* implement the predicate *s*. The declarative semantics of *S* is its least Herbrand model

$\{H \mid H \in B_S \wedge S \models H\}$ *where* B_S *is the set of all ground instances of s*

Following Clark ([Clark 79], [Deville 90]), I relate this declarative semantics to the one expressed by the correctness formula

$\{P\}\ S\ \{Q\}$

with the help of correctness and completeness criteria. If A is an axiom system which formalizes s, S is correct with respect to the least Herbrand model iff

$A \models (P \wedge S \rightarrow Q)$

and complete iff

$A \models (P \wedge Q \rightarrow S)$

Identifying A with the logic specification for S

$A = (S \leftrightarrow P \wedge Q)$

it is trivial to prove that the program S as defined by the logic specification fulfills both of Clark's criteria.

I conclude, that the program specified by the logic specification is the same as the one specified by the correctness formula, i.e. that the logic specification and the correctness formula are equivalent ways of specification. Furthermore I conclude, that the specified program is correct and complete with respect to its least Herbrand model.

5 Executable Specifications Derived From Logic Specifications

5.1 Programs as If-Halves of Logic Specifications

The *if-half*

$S \leftarrow P \wedge Q$

of the logic specification is a Horn clause which constitutes a program for S.
In the general case, this Horn clause is in the extended form

$A \leftarrow W$

introduced by [Lloyd, Topor 84]. The head *A* is an atomic formula, while the body *W* is a first-order formula including the usual connectors and quantifiers.

In spite of their increased expressiveness, extended Horn clauses remain within the Horn clause subset of predicate logic. In fact, [Lloyd, Topor 84] demonstrated that extended Horn clauses can be transformed into equivalent normal Horn clauses provided that negation is safe. For example, the formula `∀X:(P→ Q)` is transformed into the Prolog literal `not(P,not Q)`.

As we see, the if-half of a logic specification is equivalent to a normal logic program. How about the correctness and completeness of this program with respect to the least Herbrand model? The declarative semantics of normal programs is based on Clark's completion which – as pointed out – is equivalent to the logic specification plus an equality theory. Disregarding the equality theory for the moment we see that the logic specification forms the basis of the semantics of the derived normal programs and that the correctness and completeness results of section 4.3 apply.

For two reasons, programs derived as the if-halves of their logic specifications should rather be considered executable specifications. First, the programs are concise, readable, and obviously fulfill their pre- and postconditions. Second, if the pre- and postconditions are available in executable form, the programs are executable, though not necessarily efficiently. As ([Fuchs 91], [Levi 86]) pointed out, pre- and postconditions defining predicates by their properties often lead to programs based on generate-and-test, i.e. on search.

5.2 The First Few Smallest Elements of a List

As an example, I will derive the program `smallest` which finds the `S` smallest elements in an unsorted list of `N ≥ S` elements. I specify `smallest` by

```
{list(List) ∧ length(List)=N ∧ 0<S≤N}
smallest(List,S,Smallest)
{Smallest⊆List ∧ length(Smallest)=S ∧
∀X,∀Y: (X∈(Smallest) ∧ Y∈(List-Smallest) → X≤Y)}
```

The predicate `smallest` is defined as the if-half of the logic specification, i.e. as implication of the conjunction of the pre- and postconditions.

```
smallest(List,S,Smallest) ←
  list(List) ∧ length(List)=N ∧ 0<S≤N ∧
  Smallest⊆List ∧ length(Smallest)=S ∧
  ∀X,∀Y: (X∈(Smallest) ∧ Y∈(List-Smallest) → X≤Y)
```

This logic program for `smallest` is in the form of an extended Horn clause. Lloyd-Topor transformations will transform the extended Horn clause into a normal Horn clause. Expressions in functional notation are replaced by equivalent expressions in relational notation, e.g. `length(List)=N` by `length(List,N)`, or `List-Smallest` by `difference(List,Smallest,List_Smallest)`. The transformation yields the Prolog program

```
smallest(List,S,Smallest) :-
  list(List),
  length(List,N),
  0 < S, S ≤ N,
  subset(Smallest,List),
  length(Smallest,S),
  difference(List,Smallest,List_Smallest),
  not((member(X,Smallest),member(Y,List_Smallest),X>Y)).
```

Given the elementary predicates in executable form, the program is executable. Being derived from a definition of properties, `smallest` resembles more an executable specification than a program: it obviously fulfills its specification, and is based on generate-and-test.

5.3 Complete Proof Procedures

Hoare's correctness formula

$$\{P\}\ S\ \{Q\}$$

does not mean that S will necessarily terminate if started in any state in which the precondition P holds. Termination is guaranteed only for the subset of states fulfilling the weakest precondition [Dijkstra 76]. The weakest precondition $wp(S,Q)$ defines the set of all states for which the program S terminates in a state fulfilling the postcondition Q. This means that the logic program

$$S \leftarrow wp(S,Q) \wedge Q$$

also terminates.

Unfortunately, this is not necessarily true for the Prolog program derived from it. For efficiency reasons, the Prolog interpreter processes conjunctions sequentially from left to right. Consequently, Prolog's SLD proof procedure is incomplete. To ensure termination, Prolog's proof procedure must be replaced by a complete one.

We get a complete proof procedure if we delay the execution of literals which cannot yet safely be executed. Specifically, we delay negated literals until they are ground. Mechanism for delayed execution were discussed by [Naish 86], and are available in a number of Prolog implementations, and in Gödel [Hill, Lloyd 91].

Additionally, many of these languages allow to explicitly delay the execution of literals until user-defined conditions are fulfilled, i. e. the languages support coroutining. Coroutining can turn inefficient programs – specifically executable specifications – into efficient ones. For a generate-and-test solution, coroutining can interleave the execution of the generator and tester. Consequently, failure branches of the proof tree are pruned much earlier, and the proof can become significantly more efficient than without coroutining.

Furthermore, a complete proof procedure allows us to take advantage of the relational nature of an executable specification [Kowalski 85], i.e. we can query a specification for various instantiations of its arguments.

6 Derivation of Logic Programs By Structural Induction

6.1 Structural Induction

The if-halves of logic specifications result in programs which are usually rather inefficient. To derive efficient programs from logic specifications we have to employ other methods. In the following, I will investigate structural induction which leads to recursive programs.

The induction principle [Deville 90] allows us to conclude that all members of a set S have a certain property P if S is well-founded, i.e. has a well-founded relation $<$, if P is true for the minimal element E of the set, and if we can prove that P holds for an element X if P holds for all elements $Y < X$.

Structural induction is based on the recursive form of standard data structures which suggests many well-founded relations. For lists we can use the relation *suffix*, i.e. we say that `L1 < L2` iff `L1` is a suffix of `L2`. The minimal element is the empty list.

Two examples will demonstrate the power of structural induction to generate efficient recursive programs. For a comparison, I will also derive executable specifications.

6.2 Removing Elements From a List

A well-known problem in the logic programming community is the removal of the first occurrence of a given element from a list.

I describe the problem by the correctness formula

```
{list(L) ∧ X∈L}
remove(X,L,L_X)
{∃As,Bs: (list(As) ∧ list(Bs) ∧ append(As,[X|Bs],L) ∧
append(As,Bs,L_X) ∧ ¬ X∈As)}
```

This yields the logic specification

```
remove(X,L,L_X) ↔
  list(L) ∧ X∈L ∧ ∃As,Bs: (list(As) ∧ list(Bs) ∧
  append(As,[X|Bs],L) ∧ append(As,Bs,L_X) ∧ ¬ X∈As)
```

from which I can derive an executable specification in Prolog notation

```
remove(X,L,L_X) :-
  append(As,[X|Bs],L),
  not member(X,As),
  append(As,Bs,L_X).
```

This executable specification reflects the correctness formula, but is rather inefficient since it is based on generate-and-test. Structural induction allows us to derive a more efficient recursive program from the logic specification. For the two problems that I will discuss, I use induction on the structure of the list `L`.

The precondition `X∈L` means that `L` cannot be the empty list `[]`. Therefore, I only have to reason about a non-empty list, i.e. `L=[H|T]`. There are two possibilities: `X` is the first element of the list `L`, or it is not.

The element `X` is the first element of `L`, i.e. `X=H`. We get

```
remove(X,L,L_X) ↔
  L=[H|T] ∧ X=H ∧ ∃As,Bs: (list(As) ∧ list(Bs) ∧
  append(As,[X|Bs],L) ∧ append(As,Bs,L_X) ∧ ¬ X∈As)
```

This is only possible if `As=[]`, `Bs=T`, and `L_X=T`. Thus

```
remove(X,L,L_X) ↔
  L=[H|T] ∧ X=H ∧ L_X=T
```

If the element `X` is not the first element of `L`, `X` can only be in `T`, and `L_X` must have the form `[H|T_X]`, where `T_X` is `T` without the first occurrence of `X`.

```
remove(X,L,L_X) ↔
  L=[H|T] ∧ X≠H ∧ X∈T ∧ L_X=[H|T_X] ∧
  ∃As,Bs: (list(As) ∧ list(Bs) ∧ append(As,[X|Bs],L_X) ∧
  append(As,Bs,T_X) ∧ ¬ X∈As)
```

With the help of the logic specification this can be simplified to

```
remove(X,L,L_X) ↔
  L=[H|T] ∧ X≠H ∧ L_X=[H|T_X] ∧
  remove(X,T,T_X)
```

Using only the if-halves of the results and performing all substitutions I derive the Prolog program

```
remove(X,[X|T],T).
remove(X,[H|T],[H|T_X]) :-
  X\=H,
  remove(X,T,T_X).
```

This recursive program is substantially more efficient than its executable specification. But it no longer bears any resemblance to the correctness formula.

6.3 Hamming Numbers

Hamming numbers are the ordered sequence of natural numbers which have as prime factors only 2, 3, and 5. Since there are infinitely many Hamming numbers, the sequence is not computable, but finite prefixes are.

I define the predicate `hamming(Limit,Hs)` which is true if `Hs` is the finite sequence of Hamming numbers smaller than or equal to `Limit`. I represent sequences as lists. `N` stands for the set of natural numbers. The predicate `hamming` is defined by the correctness formula

```
    {Limit∈N ∧ list(Hs)}
    hamming(Limit,Hs)
    {Hs={H | H∈N ∧ has_primefactors_2_3_5(H) ∧ H ≤ Limit}}
```

which leads to the logic specification

```
    hamming(Limit,Hs) ↔
      Limit∈N ∧ list(Hs) ∧
      Hs={H | H∈N ∧ has_primefactors_2_3_5(H) ∧ H ≤ Limit}
```

from which I can derive the executable specification [Fuchs 91]

```
    hamming(Limit,Hs) :-
      set(H,(natural_number(H,Limit), has_primefactors_2_3_5(H)),Hs).
```

The predicate `natural_number(H,Limit)` generates natural numbers `H` up to `Limit`, while `has_primefactors_2_3_5` tests for 'Hamming'-ness. Again, this executable specification is based on generate-and-test.

For a more efficient recursive program I use induction with the list `Hs` as induction parameter. If `Hs` is the empty list `[]`, we can only have `Limit=1`.

```
    hamming(Limit,Hs) ↔ Limit=1, Hs=[].
```

If `Hs` is the non-empty list `[X|Xs]` of Hamming numbers, there are two cases, depending on whether `Limit` is a Hamming number or not.

If `Limit` is a Hamming number, I let `Hs=[Limit|Xs]` where `Xs` is the list of Hamming numbers smaller than or equal to `Limit-1`.

```
    hamming(Limit,Hs) ↔
      Limit∈N ∧ has_primefactors_2_3_5(Limit) ∧
      Hs=[Limit|Xs] ∧ list(Xs) ∧ hamming(Limit-1,Xs)
```

If `Limit` is not a Hamming number all elements of `Hs` must be smaller than or equal to `Limit-1`.

```
hamming(Limit,Hs) ↔
  Limit∈N ∧ not has_primefactors_2_3_5(Limit) ∧
  hamming(Limit-1,Hs)
```

Using only the if-halves of the equivalences I translate them into Prolog clauses. I introduce an executable predicate `natural_number` which generates natural numbers, and perform all substitutions.

```
hamming(1,[]).
hamming(Limit,[Limit|Xs]) :-
  natural_number(Limit),
  has_primefactors_2_3_5(Limit),
  hamming(Limit-1,Xs).
hamming(Limit,Hs) :-
  natural_number(Limit),
  not has_primefactors_2_3_5(Limit),
  hamming(Limit-1,Hs).
```

Finally, I take into account that Hamming numbers are ordered by introducing the interface predicate `hamming_numbers` which calls `hamming` and sorts the result.

```
hamming_numbers(Limit,HammingNumbers) :-
  hamming(Limit,Hs),
  sort(Hs,HammingNumbers).
```

It is interesting to note that the derived logic program is an order of magnitude more efficient than its executable specification.

7 Conclusions

Starting with Hoare's correctness formulae for predicates, I define logic specifications for the predicates as if-and-only-if implications of the conjunction of the pre- and postconditions. Logic specifications and correctness formulae are equivalent, and the specified programs are correct and complete with respect to the least Herbrand model.

Logic specifications form the basis for two constructive methods to derive programs. On the one hand, the if-halves of logic specifications are – possibly extended – Horn clauses which are executable if the predicates of the pre- and postconditions are executable. On the other hand, recursive program for the predicate can be derived from logic specifications by structural induction.

Both methods have their advantages and disadvantages. The first method leads directly to concise programs which are obviously correct and reflect their correctness formulae. On the other side, conjunctions of pre- and postconditions often result in inefficient programs based on generate-and-test. Thus I call programs derived by the first method executable specifications. The second method generates programs that can be orders of magnitude more efficient than their executable specification counterparts, but – being recursive – the programs are no longer obvious logical consequences of their correctness formulae.

Acknowledgements

I would like to thank Yves Deville and Markus P. J. Fromherz for valuable comments on an earlier draft of this paper.

This research has been partially supported by the Swiss National Science Foundation under contract 2000-5.449.

References

[Bundy et al. 90] A.Bundy, A. Smaill, G. Wiggins, The Synthesis of Logic Programs from Inductive Proofs, in: J. W. Lloyd (ed.), Computational Logic, ESPRIT Symposium Proceedings, Brussels November 1990, Springer, 1990

[Clark 78] K. L. Clark, Negation As Failure, in: H. Gallaire, J. Minker (eds.), Logic and Data Bases, Plenum Press, 1978, pp. 293-322

[Clark 79] K. L. Clark, Predicate Logic As a Computational Formalism, Research Report 79/59, Imperial College, London, 1979

[Colussi, Marchiori 91] L. Colussi, E. Marchiori, Proving Properties of Logic Programs Using Axiomatic Semantics. Proceedings 8th International Conference on Logic Programming. MIT Press, 1991

[Cousot 90] P. Cousot, Methods and Logics for Proving Programs, in: J. van Leeuwen (ed.), Handbook of Theoretical Computer Science, Vol. B: Formal Models and Semantics, Elsevier, 1990

[Deville 90] Y. Deville, Logic Programming, Systematic Program Development, Addison-Wesley Publishing Company, 1990

[Dijkstra 76] E. W. Dijkstra, A Discipline of Programming, Prentice-Hall, 1976

[Flener 91] P. Flener, Towards Stepwise, Schema-Guided Synthesis of Logic Programs, in: K.-K. Lau, T. Clement (eds.), Proceedings of LOPSTR '91, Springer, 1992

[Fribourg 90] L. Fribourg, Extracting logic programs from proofs that use extended Prolog execution and induction, in: Proceedings 7th International Conference on Logic programming, MIT Press, 1990

[Fuchs 91] N. E. Fuchs, Specifications Are (Preferably) Executable, Software Engineering Journal, September 1992, also: Technical Report 91.10, Institut für Informatik, Universität Zürich, 1991

[Hill, Lloyd 91] P. M. Hill, J. W. Lloyd, The Gödel Report, TR-91-02, Computer Science Department, University of Bristol, March 1991

[Hoare 69]	C. A. R. Hoare, An axiomatic basis for computer programming, CACM 12, pp. 576-583, October 1969
[Kowalski 79]	R. A. Kowalski, Logic for Problem Solving, North-Holland, 1979
[Kowalski 85]	R. A. Kowalski, The relation between logic programming and logic specification, in: C. A. R. Hoare, J. C. Shepherdson (Eds.), Mathematical Logic and Programming Languages, Prentice-Hall International, 1985
[Lau, Prestwich 91]	K.-K. Lau, S. D. Prestwich, Synthesis of a Family of Recursive Sorting Procedures, in: V. Saraswat, K. Ueda (eds.), Proceedings of the 1991 International Symposium on Logic Programming, MIT Press, 1991
[Levi 86]	G. Levi, New Research Directions in Logic Specification Languages, in: H.-J. Kugler (Ed.), Information Processing 86 (IFIP), Elsevier Science Publisher, pp. 1005-1008, 1986
[Lloyd 87]	J. W. Lloyd, Foundations of Logic Programming,, Springer, 1987
[Lloyd, Topor 84]	J. W. Lloyd, R. W. Topor, Making Prolog More Expressive, Journal of Logic Programming, Vol. 1, No. 3, pp. 225-240, 1984
[Naish 86]	L. Naish, Negation and Control in Prolog, LNCS 225, Springer, 1986