Phil. Trans. R. Soc. Lond. A **312**, 345–361 (1984) [345]
Printed in Great Britain

The relation between logic programming and logic specification

By R. Kowalski
Department of Computing, Imperial College, 180 Queen's Gate, London SW7 2BZ, U.K.

Formal logic is widely accepted as a program specification language in computing science. It is ideally suited to the representation of knowledge and the description of problems without regard to the choice of programming language. Its use as a specification language is compatible not only with conventional programming languages but also with programming languages based entirely on logic itself. In this paper I shall investigate the relation that holds when both programs and program specifications are expressed in formal logic.

In many cases, when a specification *completely* defines the relations to be computed, there is no syntactic distinction between specification and program. Moreover the same mechanism that is used to execute logic programs, namely automated deduction, can also be used to execute logic specifications. Thus all relations defined by complete specifications are executable. The only difference between a complete specification and a program is one of efficiency. A program is more efficient than a specification.

Logic programming

The use of logic as a specification language may be more familiar than its use as a programming language. A short introduction may therefore be appropriate.

Logic programming (Kowalski 1974, 1983) is based upon (but not necessarily restricted to) the interpretation of rules of the form

A if B and C and ...

as procedures

to do A,
do B and do C and ....

This interpretation is equivalent to 'backwards reasoning' and is a special case of the resolution rule of inference (Robinson 1965). It is the basis also of the programming language PROLOG (Colmerauer *et al.* 1973; Clark & McCabe 1984).

Example 1.

(*a*) x is mortal if x is human.

This is interpreted as the procedure

'to show x mortal
show x human'

or

'to find x mortal
find x human'

depending whether x is given or not.

(*b*) Socrates is human.

This is interpreted as a procedure that solves problems without introducing further subproblems.

The notation of (*a*) and (*b*) is one we have used to simplify the syntax of logic for teaching children (Ennals 1982). This syntax is translated by a PROLOG program into more conventional symbolic notation:

Mortal(x) if Human(x)
Human(Socrates).

To find a mortal we pose the *query*

Find x where Mortal(x).

This is equivalent to proving the theorem:

there exists an x such that x is mortal, i.e.
$\exists$ x Mortal(x).

The procedural interpretation answers the query by reasoning backwards from the theorem to be proved by using 1 (*a*) to reduce the problem to the subproblem of showing

there exists an x such that x is human,

which is solved directly by 1 (*b*). As a byproduct, the solution

x = Socrates

can be extracted from the proof. In this way 1 (*a*) and 1 (*b*) behave as a program and backwards reasoning behaves like procedure invocation.

In this example we are given declarative statements of classical logic, and the procedural interpretation turns them into a program. The following example is more obviously related to computing.

A LOGIC SPECIFICATION OF THE SORTING PROBLEM

Example 2.

Sort(x y) if Permutation(x y)
and Ordered(y).

The rule can be read both declaratively:

'y is a sorted version of x
if y is a permutation of x
and y is ordered'

and procedurally:

'to sort x into y
find a permutation y of x and
show y is ordered'.

Given complete definitions of the lower-level Permutation and Ordered relations, the rule for Sort *completely* specifies the notion of sorting. The procedural interpretation turns the specification into an executable, although very inefficient, non-deterministic program.

Notice that the specification of sortedness can be used both to test given x and y and to generate y as output, given x. It can even be used to generate pairs of x and y that satisfy the Sort relation. This is the same flexibility that we have with database queries. The mechanization of theorem-proving makes it possible to query program specifications.

Note also that the specification of sortedness contains no assignment statements or other side effects. The purpose of assignment is to re-use space for the purpose of efficiency. It has no place, therefore, in program specifications where the emphasis is on clarity.

A LOGIC PROGRAM FOR THE SORTING PROBLEM

Example 3.

Sort(x x) if Ordered(x)
Sort(x y) if i < j
and $x_i > x_j$
and Interchange(x i j z)
and Sort(z y).

The declarative reading of the rules is obvious:

If x is ordered
then x is already a sorted version of itself.
If x contains some out of order pair
$x_i > x_j$ where i < j
and these are interchanged giving an
intermediate sequence z
and y is a sorted version of z
then y is also a sorted version of x.

But it is the procedural interpretation of the rules that makes them useful. Taken together the two rules behave as a procedure. Given x as input and y as output

to sort x,
repeatedly interchange out of order pairs
$x_i > x_j$ where i < j
until x is ordered.

Here we have used functional syntax with relational semantics. The single condition

$x_i > x_j$,

which employs a form of functional syntax, would need to be replaced by three conditions in relational syntax:

Contains(x i u) and
Contains(x j v) and
u > v.

We shall have more to say about the relation between functional and relational syntax later.

Notice that the procedural reading of the rules suggests a destructive assignment operation that replaces the sequence x by the result of interchanging x_i and x_j. Such an assignment that destroys the preceding value of x is an efficiency-improving operation that should not be allowed to affect the semantics of the rules as determined by their declarative reading.

VERIFICATION OF LOGIC PROGRAMS

A *complete logic specification* defines a collection of relations in the same way that a *logic program* does. Given a set of sentences S containing a predicate, say P(x y), S defines P in the sense that

(s t) is in the relation P
iff
S logically implies P(s t)
or equivalently
for first-order logic
iff
P(s t) can be proved from S.

This notion of 'definition' accords with our intuition in the case where S is a logic program. Because it is so hard to distinguish between programs and complete specifications, we use the same notion of definition for both. But notice that, according to this, any relation defined by a finite set of sentences of first-order logic is semi-computable (i.e. recursively enumerable). This conflicts with the usual notion of semantic definition in which even very simple sentences of first-order logic 'define' uncomputable relations.

We shall now prove that the sort program of example 3 meets the specification of example 2. To do this we shall show that if (s t) is in the relation defined by the program then it is also in the relation defined by the specification.

Proof by induction on the length n of a proof (computation) of Sort(s t) by means of the program. We shall show that

if the program implies Sort(s t) then
t is a permutation of s and
t is ordered.

n = 1. Sort(s t) can be proved by using the first rule of the program alone.

Therefore s = t and s is ordered.
So t is a permutation of s and t is ordered.

n = k + 1. There is a backward proof of Sort(s t), the first step of which uses the second rule of the program to reduce the goal Sort(s t) to the subgoals

$i < j$
$s_i > s_j$
Interchange(s i j r)
Sort(r t),

for some concrete i, j and r. Obviously, the proof of the subgoal Sort(r t) takes fewer than k + 1 steps. Therefore we may assume by the induction hypothesis that

t is a permutation of r and
t is ordered.

But Interchange(s i j r) implies that

r is a permutation s.

Therefore

t is a permutation of s and
t is ordered.

The relation between the specification of sortedness and various logic programs for sorting lists has been studied in detail by Clark & Darlington (1980).

The inductive method of proof, which works in this example to show that a program meets its specification, works whether the specification is partial or complete. The same proof, in particular, shows that the program meets the *partial* specification that

t is ordered whenever Sort(s t) is logically implied by the program.

Notice, however, as this example shows, there is no purely syntactic distinction between complete and partial specifications. A definition of orderedness is a complete specification for the problem of testing or generating ordered sequences, but it is only a partial specification for the sorting problem.

A PROGRAM WITHOUT A SPECIFICATION

The following example, due to Peter Hammond (Hammond & Sergot 1983), is a logical reconstruction of a small part of a medical expert system implemented in the expert system shell EMYCIN. It can be regarded either as a program without a specification or as a complete specification that runs efficiently enough not to need a separate program.

Example 4.

x should take y if x has complained of z
and y suppresses z
and Not y is unsuitable for x
y is unsuitable for x if y aggravates z
and x has condition z
aspirin suppresses inflammation
aspirin suppresses pain
etc.
aspirin aggravates peptic ulcer
lomotil aggravates impaired liver function
etc.

This example is characteristic of many others whose main objective is to represent some aspect of human knowledge or expertise. It has no specification besides the informal constraint that it reflects such knowledge and expertise as faithfully as possible. In this respect it bears more resemblance to a program specification than it does to a program. On the other hand, because it runs with tolerable efficiency, it is impossible not to regard it as a program as well. Similar examples can be found in the field of expert systems, in the formalization of legislation (Cory *et al.* 1984) and in data processing, where database queries, for example, can be regarded as specifications that behave like programs. This example also illustrates two other features of 'logic programs': negation by failure and declarative input–output.

Negation by failure

In our previous examples we have restricted ourselves to the use of rules and queries with positive atomic conditions. This is equivalent to the *Horn clause* subset of logic. In example 4 we have extended the Horn clause subset to allow negated conditions. The procedural interpretation of Horn clauses can be extended to deal with such conditions. For example, a negated condition such as

Not y is suitable for x

is judged to hold if the positive condition

y is unsuitable for x

fails to hold. Such negation by failure is consistent with classical negation (Clark 1978) provided the implicit only-if half of 'definitions' is made explicit, i.e. in this case provided the second rule is re-expressed as

y is unsuitable for x

iff there exists z such that y aggravates z

and x has condition z.

Moreover, negation by failure can only be used to test conditions and not to generate solutions. For example, given the additional assumption

John has condition peptic ulcer,

the negated goal

Find y where Not y unsuitable for John

incorrectly fails because the unnegated goal.

Find y where y unsuitable for John

succeeds. Thus, although negation by failure is correct and efficient, it is also incomplete. The limitations of negation by failure are discussed in Kowalski (1983).

Declarative input–output

In example 4 the relations

x has complained of z

and

x has condition z

are not defined. The appropriate parts of their 'definitions' can be provided dynamically by 'the user' as they are required by the system. This makes input–output *declarative* in the sense that it can be understood entirely in logical terms: the output is a logical consequence of the information initially contained in the system together with any information provided by the user (Sergot 1982). The input can be given in any order, provided it does not affect the logical implication of the output.

The extended Horn clause subset of logic

The Horn clause subset of logic, even if it is extended to allow negated conditions, lacks the expressive power of the standard form of logic. For example, the natural *specification* of the subset relation requires the use of a condition that itself has the form of a universally quantified Horn clause.

Example 5.

(*a*) $X \subseteq Y$ if For all z [$z \in Y$ if $z \in X$].

We call this generalization of the Horn clause subset of logic, in which conditions can have the form of universally quantified Horn clauses, the *extended Horn clause subset of logic*. This extension has great expressive power and many examples such as the definition of subset, the definition of greatest common divisor and the definitions of ordered sequence given later in the paper can be formalized naturally within this language. The use of conditions that are universally quantified Horn clauses often makes it possible to avoid the use of recursion. For the definition of subset, recursion is unavoidable if we restrict ourselves to the use of Horn clauses:

(*b*) Nil $\subseteq$ Y
cons(u v) $\subseteq$ Y if $u \in Y$ and $v \subseteq Y$.

The Horn clause definition of subset is the obvious recursive *program*, given that sets are represented by lists. It is very like a program in a functional programming language and it can be directly executed in PROLOG.

Given an appropriate definition of the membership relation $\in$, 5 (*a*) is a complete specification of the subset relation. It is executable in the sense that if $s \subseteq t$ is implied by 5 (*a*) together with the associated definition of $\in$ then $s \subseteq t$ can be proved by means of a mechanical theorem-proving procedure. With today's general-purpose theorem-proving procedures, however, this will be very inefficient. But by translating 'For all' into double negation and interpreting negation by failure, universally quantified Horn clause conditions can be executed both correctly and efficiently. The specification 5 (*a*) of subset, in particular, will run as an iteration:

> given s and t it will attempt to show $s \subseteq t$ by consecutively generating the elements of s and showing that they belong to t (by *failing* to show that there is any element of s that does *not* belong to t).

Such iterative execution of the specification can be more efficient than recursive execution of the program. Thus much of the work that has previously gone into the derivation of programs from specifications may have been wasted. Moreover, it has also distracted attention from the problem of designing theorem-provers that efficiently execute program specifications.

The 'specification' of subset can benefit more than the recursive program from being executed by means of different strategies. For example, an 'or-parallel' theorem-prover that can explore alternatives in parallel could, conceptually at least, attempt to show $s \subseteq t$ by exploring the elements of s in parallel, perhaps using associative look-up to show that they all belong to t. Thus we obtain different algorithms simply by changing the mode of execution (Kowalski 1979).

On the other hand, the translation of 'For all' into double negation and the interpretation of negation as failure, although correct and efficient, is incomplete. Certain logical consequences

cannot be proved by these means. In particular, although 5 (*a*) can be used to test that the subset relation holds for given s and t, by using these methods it cannot be used to generate s from t or t from s. Moreover, it will succeed only when s has finitely many members.

The derivation of logic programs from logic specifications

Given a logic program and a complete logic specification, it is often possible to derive the program from the specification. Such programs are derived by using the rules of logic and as a consequence are guaranteed to be correct.

Although in the subset example it may be better to execute the 'specification' than the 'program', in other cases no execution strategy can render a specification as efficient as a program. It is instructive, therefore, to show how 5 (*b*) can be derived from 5 (*a*), if only because it is a simple example and because such derivation is useful in other cases. For this we need to replace the if-half of the specification by its full iff form:

$$X \subseteq Y \text{ iff For all } z\ [z \in Y \text{ if } z \in X]; \tag{s1}$$

we also need auxiliary definitions to specify how sets can be represented by lists:

$$\text{Not Exists } z\ [z \in \text{Nil}] \tag{s2}$$

$$z \in \text{cons}(u\ v) \text{ iff } z = u \text{ or } z \in v. \tag{s3}$$

These three sentences, together with the axioms of equality, constitute a *complete specification*, which logically implies the program.

Proof.

(*a*) The specification implies the recursive clause of the program:

$X \subseteq Y$ if For all z $[z \in Y$ if $z \in X]$ (by (s1))
(the if-half of the specification)
$\text{cons}(u\ v) \subseteq Y$ if For all z $[z \in Y$ if $[z = u$ or $z \in v]]$ (by (s3))
$\text{cons}(u\ v) \subseteq Y$ if For all z $[z \in Y$ if $z = u]$
and For all z $[z \in Y$ if $z \in v]$
$\text{cons}(u\ v) \subseteq Y$ if $u \in Y$ and $v \subseteq Y$ (by (s1))

(*b*) The specification implies the basis of the program:

$\text{Nil} \subseteq Y$ if For all z $[z \in Y$ if $z \in \text{Nil}]$ (by (s1))
(an instance of the if-half of the specification)
$\text{Nil} \subseteq Y$ if For all z $[z \in Y$ if False] (by (s2))
$\text{Nil} \subseteq Y$ if True
$\text{Nil} \subseteq Y$.

Note that the proof of (*a*) can be regarded as generalizing the fold–unfold method of Burstall & Darlington (1977). It can be viewed both as logical proof and as symbolic execution of the specification. This method of deriving logic programs from logic specifications has been developed by Clark (Clark & Sickel 1977; Clark *et al.* 1982); Hogger (1978*a*, *b*) and others (Hanson & Tarnlund 1979; Winterstein *et al.* 1980).

We have shown that the program is partially correct by showing that it is logically implied by its specification. This may be counter-intuitive, and therefore requires explanation. Suppose

a set of sentences S defines a relation P(x y). (In a typical application, x might be input, y output and P(x y) an input–output relation.) This means that

(x y) is in relation P
iff
$S \vdash P(x\ y)$.

Where $X \vdash Y$ means that conclusion Y can be proved from assumptions X. Thus not only does S imply P(x y), it also 'computes' it, in the sense that its instances can be derived by means of a mechanical theorem-proving procedure. Suppose Spec and Prog are a complete specification and a program, respectively, defining a relation P(x y). Then to say that the program is *partially correct* relative to the specification is to say that every instance of P that is a consequence of Prog is also a consequence of Spec, i.e.

if $Prog \vdash P(x\ y)$ then $Spec \vdash P(x\ y)$.

But this holds if

$Spec \vdash Prog$

by the transitivity of the proof predicate. If the converse holds, i.e.

if $Spec \vdash P(x\ y)$ then $Prog \vdash P(x\ y)$,

then the program is *complete* in the sense that the program derives every instance of P that is defined by the specification. This holds if

$Prog \vdash Spec$.

Notice that to prove correctness or completeness we can replace either Spec or Prog by any stronger set of sentences that implies the same atomic relations. This justifies the use of the iff form of definitions and induction in such proofs. Thus it is closely related to Hoare's identification of a program (this symposium) with the *strongest* predicate that describes its behaviour.

Notice also that to prove a specification implies a program, we need to show that the specification logically implies each clause in the program. Thus the *complexity* of proving partial correctness is linear in the number of clauses in the program, and the complexity of proving completeness is linear in the number of clauses in the specification.

Verification of the Euclidean algorithm

The Euclidean algorithm can be verified by the same technique of deriving programs from specifications that was used to verify the recursive program for subset. For the Euclidean algorithm, however, the program is significantly more efficient than the specification. Here we use functional notation for the sake of clarity.

Example 6.

(*a*) gcd(x y) = z if z divides x
and z divides y
and For all u [u $\leqslant$ z if u divides x
and u divides y].

Notice that, like the definition of subset, this definition is expressed in the extended Horn clause subset of logic. However, the Euclidean algorithm for gcd can be expressed in Horn clause form:

(*a*) gcd(x y) = z if x divides y
gcd(x y) = z if x $\leqslant$ y
and x divides y with remainder r
and r $\neq$ 0
and gcd(r x) = z.

As for example 5, it can be shown that the specification (in iff form together with auxiliary properties of 'divides', '$\leqslant$' and the axioms of equality) logically implies the program. However, unlike example 5, the derivation requires genuine mathematical ingenuity. Although both specification and program are executable, the program is significantly more efficient.

Note.

(1) The functional notation gcd(x y) = z can be rewritten in relational notation as Gcd(x y z).

(2) By exploiting functional notation, the second clause of the program can be written more compactly as

gcd(x y) = gcd(r x) if x $\leqslant$ y
and x divides y with remainder r
and r $\neq$ 0.

(3) For the sake of completeness we need to add a third clause

gcd(x y) = gcd(y x) if y $\leqslant$ x.

The use of functional notation simplifies the program, but is inefficient if the equality symbol, =, is defined by means of the axioms of equality. We shall show later how functional notation can be transformed into relational notation without equality. Thus we can have the convenience of functional notation while retaining the semantics of relations. Relational semantics gives us both partial functions and non-deterministic functions as special cases.

Alternative specifications of orderedness

In our next and last example we show that the technique of deriving programs from specifications also applies when both the specification and the program are formulated in the extended Horn clause subset of logic. Moreover, it applies when both program and specification can be regarded as alternative specifications.

Example 7.

A *specification–program* of ordered sequence:

(*a*) Ordered(x) if For all i j [$x_i \leqslant x_j$ if i < j].

Another *specification–program*:

(*b*) Ordered(x) if For all i [$x_i \leqslant x_{i+1}$].

Executed by means of an extended Horn clause theorem-prover, 7 (*b*) is significantly more efficient than 7 (*a*). Given a sequence x of finite length n, it takes time/space proportional to

n to test whether x is ordered; 7 (*a*) takes time/space proportional to n^2. Both 7 (*a*) and 7 (*b*) are more flexible than a conventional algorithm in that the elements of the sequence x can be accessed either in sequence or in parallel. Either way, 7 (*b*) can be executed at least as efficiently as a conventional program.

It is instructive to compare 7 (*b*) with the corresponding Horn clause program, where sequences are represented by lists

Ordered(Nil)
Ordered(cons(u Nil))
Ordered(cons(u cons(v w))) if $u \leqslant v$
and Ordered(cons(v w)).

This is less natural than 7 (*b*) and less flexible. Not only is it restricted to sequential exploration of the sequence, but it is committed to exploring it in one particular order.

It is harder to compare 7 (*a*) and 7 (*b*) as specifications than it is to compare them as programs. What is clear, however, is that it is much easier to show that 7 (*b*) implies 7 (*a*) than it is to show the converse.

THEOREM. *Example* 7 (*b*), *with appropriate properties of* $\leqslant$, *implies* 7 (*a*).

Proof. Assume

	For all i j $[x_i \leqslant x_j$ if $i < j]$,	
then	For all i $[x_i \leqslant x_{i+1}]$	(since $i < i+1$),
then	Ordered (x)	(assuming 7 (*b*)),
therefore	7 (*a*).	

This can be interpreted as showing either that 7 (*a*) is a *correct program* relative to 7 (*b*) as specification or that 7 (*b*) is a *complete program* relative to 7 (*a*) as specification.

THEOREM. *Example* 7 (*a*), *with induction and appropriate properties of* $\leqslant$, *implies* 7 (*b*).

Proof. We shall show 7 (*a*) implies 7 (*b*) by showing that

For all i $[x_i \leqslant x_{i+1}]$

implies

For all i j $[x_i \leqslant x_j$ if $i < j]$.

The proof is by induction on j.

$j = 0$. For all i $[x_i \leqslant x_0$ if $i < 0]$ is vacuously true on the assumption that sequence indices are non-negative.

$j = k+1$. Assume $i < k+1$. We need to show $x_i \leqslant x_{k+1}$.

Case 1. $i = k$.

Then	$x_i \leqslant x_k$,	
	$x_k \leqslant x_{k+1}$	(assuming 7 (*a*)),
therefore	$x_i \leqslant x_{k+1}$.	

Case 2. $i < k$.

Then	$x_i \leqslant x_k$	(by induction hypothesis),
	$x_k \leqslant x_{k+1}$	(assuming 7 (*a*)),
therefore	$x_i \leqslant x_{k+1}$.	

The relation between functional and relational notation

Perhaps the main alternative to the use of logic for program specification is some form of functional language with equality. The semantics of such languages are usually defined by means of algebra or category theory, and rewrite or reduction rules are usually used to execute them.

If we restrict attention to first-order functional languages (in which functions are not allowed as arguments to functions) then it is possible to transform equations into Horn clauses, providing them with a model theoretic semantics and predicate logic proof rules. This is done by treating functions as a special case of relations. We use the schemas

$$f(x) = y \text{ iff } F(x\ y), \tag{E1}$$

$$P(f(x)) \text{ iff For all } y\ [P(y) \text{ if } f(x) = y], \tag{E2}$$

where with every function symbol f there is associated a predicate symbol F and P(y) is any formula with free variable y. For simplicity we consider functions of one argument. Functions of several arguments can be treated in the same way.

Consider the simple example of a functional program for computing the length of lists:

length(Nil) = 0
length(cons(u v)) = length(v) + 1

We can use (E1) and (E2) to transform this into a Horn clause program.

(*a*)	length(Nil) = 0	
	Length(Nil 0)	(by (E1))
(*b*)	length(cons(u v)) = length(v) + 1	
	Length(cons(u v) length(v) + 1)	(by (E1))
	Length(cons(u v) y) if length(v) + 1 = y	(by (E2))
	Length(cons(u v) y) if Plus(length(v) 1 y)	(by (E1))
	Length(cons(u v) y) if Plus(z 1 y) and length(v) = z	(by (E2))
	Length(cons(u v) y) if Plus(z 1 y) and Length(v z)	(by (E1))

Notice that the application of (E2) in the penultimate step involves treating the whole formula

Length(cons(u v) y) if Plus(length(v) 1 y)

as P(length(v)). By using (E2) this becomes

[Length(cons(u v) y) if Plus(z 1 y)] if length(v) = z,

which simplifies to

Length(cons(u v) y) if Plus(z 1 y) and length(v) = z.

In Kowalski (1983) this step was performed more directly by using an additional schema (E3)

P(f(x)) iff Exists y [P(y) and f(x) = y],

which unnecessarily assumes that the function f is total.

The schema (E1) and (E2) have been used in effect as a higher-order program that transforms equations into Horn clauses. This transformation, moreover, provides the basis for a simple *proof* that any recursively enumerable function can be computed by means of Horn clauses. Take any set of recursion equations defining a recursively enumerable function and apply (E1) and (E2) to produce a Horn clause program that computes the same function. It is straightforward to verify that each computation step, which uses the recursion equations as rewrite rules, can be mimicked by backwards reasoning by using the corresponding Horn clauses to get the same result. (The proof, which is given in greater detail in Kowalski (1983), uses (E3) in addition to (E1) and (E2), and therefore incorrectly assumes that recursively enumerable functions are total.)

Conclusion

The examples investigated in this paper show how hard it is to distinguish logic programs from complete logic specifications. The only criterion that can be used to discriminate between them seems to be relative efficiency, but this applies just as much to pairs of programs as it does to pairs of programs and specifications.

Given a distinction between program and specification, however, verification of the program reduces to a demonstration of logical implication. Given a complete specification, in particular, it is often possible to verify the program by showing that it can be derived from the specification by using the rules of logic.

Logic sufficiently blurs the distinction between program and specification that many logic programs can just as well be regarded as executable specifications. On one hand, this can give the impression that logic programming lacks a programming methodology; on the other, it may imply that many of the software engineering techniques that have been developed for conventional programming languages are inapplicable and unnecessary for logic programs.

References

Burstall, R. M. & Darlington, J. 1977 Transformation for developing recursive programs. *J. Ass. comput. Mach.* **24**, 44–67.

Clark, K. L. 1978 Negation as failure. In *Logic and data bases*, pp. 293–322. New York: Plenum Press.

Clark, K. L. & Darlington, J. 1980 Algorithm classification through synthesis. *Computer J.* 61–65.

Clark, K. L. & McCabe, F. 1984 *Micro-PROLOG: programming in logic*. Englewood Cliffs, N.J.: Prentice-Hall.

Clark, K. L., McKeeman, W. M. & Sickel, S. 1982 Logic program specification of numerical integration. In *Logic programming* (ed. K. L. Clark & S.-A. Tarnlund), pp. 123–139. London: Academic Press.

Clark, K. L. & Sickel, S. 1977 Predicate logic: a calculus for deriving programs. *Proc. 5th Int. Joint Conf. on Artif. Intell. Cambridge, Mass.*

Clark, K. L. & Tarnlund, S.-A. 1977 A first order theory of data and programs. In *Proc. IFIP 1977*, pp. 939–944. Amsterdam: North-Holland.

Clark, K. L. & Tarnlund, S.-A. (eds) 1982 *Logic programming*. London: Academic Press.

Colmerauer, A., Kanoui, H., Pasero, R. & Roussel, P. 1973 Un Système de Communication Homme–machine en Français. Groupe Intelligence Artificielle, Université d'Aix Marseille, Luminy.

Cory, H. T., Hammond, P., Kowalski, R. A., Kriwaczek, F., Sadri, F. & Sergot, M. 1984 *The British Nationality Act as a logic program*, Department of Computing, Imperial College, London.

Ennals, J. R. 1982 *Beginning micro-PROLOG*, Computers in education. London: Heinemann.

Hammond, P. & Sergot, M. 1983 A PROLOG shell for logic-based expert systems. In *Proc. 3rd BCS Expert Systems Conference*, pp. 95–104.

Hansson, A. & Tarnlund, S.-A. 1979 A natural programming calculus. In *Proc. 6th IJCAI, Tokyo, Japan*, pp. 348–355.

Hogger, C. J. 1978*a* Program synthesis in predicate logic. In *Proc. AISB/GI Conf. on Artif. Intell., Hamburg*, pp. 18–20.

Hogger, C. J. 1978*b* Goal oriented derivation of logic programs. In *Proc. MFCS Conf., Polish Academy of Sciences, Zakopane*, pp. 267–276.
Hogger, C. 1981 Derivation of logic programs. *J. Ass. comput. Mach.* **28**, 372–422.
Kowalski, R. A. 1974 Predicate logic as programming language. In *Proc. IFIP*, pp. 569–574. Amsterdam: North-Holland.
Kowalski, R. A. 1979 Algorithm = logic + control. *J. Ass. comput. Mach.* **22**, 425–436.
Kowalski, R. A. 1979 Logic for problem solving. New York, Amsterdam: Elsevier.
Kowalski, R. A. 1983 Logic programming. In *Proc. IFIP*, pp. 133–145. Amsterdam: North-Holland.
Robinson, J. A. 1965 A machine oriented logic based on the resolution principle. *J. Ass. comput. Mach.* **12**, 23–41.
Sergot, M. 1982 A query-the-user facility for logic programming. In *Proc. ECICS, Stresa, Italy* (ed P. Degano & E. Sandewall) pp. 27–41 Amsterdam: North-Holland.
Warren, D., Pereira, L. M. & Pereira, F. 1977 Prolog – the language and its implementation compared with Lisp. In *Proc. Symp on Artif. Intell. and Programming Languages. SIGPLAN Notices* (no. 8) **12** and *SIGART Newsletter* **64**, 109–115.
Winterstein, G., Dausmann, M. & Persch, G. 1980 Deriving different unification algorithms from a specification in logic. In *Proceedings of Logic Programming Workshop, Debrecen, Hungary* (ed. S.-A. Tarnlund), pp. 274–285.

Discussion

M. A. JACKSON (101 *Hamilton Terrace, London, U.K.*). Professor Kowalski spoke more than once of the 'efficiency' of a specification. Does he interpret efficiency with respect to some operational definition of his semantics?

R. KOWALSKI. The execution of a logic program or specification is made by means of a mechanical proof procedure. Given a fixed proof procedure, different specifications–programs behave with different efficiency. The efficiency of a specification, therefore, can only be evaluated relative to some proof procedure. Such a proof procedure can be regarded as defining an operational semantics. Model theoretic semantics defines a denotational semantics.

M. J. ROGERS (*Department of Computer Science, University of Bristol, U.K.*). Programs written in PROLOG seem in practice to be very much more difficult to check for correctness than those written in a procedural language. Part of this difficulty appears to stem from tracing the actions in taking the first matching clause and then following the program through nested loops. The problem is further complicated by the action of assert and retract clauses.

R. KOWALSKI. In the paper I have restricted my attention to logic programming with Horn clauses and various extensions. I have not considered PROLOG and its relation with logic programming.

Pure logic programs can be verified without taking their behaviour into account. This makes it possible to use increasingly sophisticated proof procedures to execute logic programs without complicating correctness proofs.

It is possible to write pure logic programs in PROLOG, but because of its depth-first search, this can give rise to infinite loops and other less extreme forms of inefficiency, especially when executing specifications rather than programs. To avoid such loops the 'programmer' may need to control the order in which clauses are executed. The verification of such programs then needs to take behaviour into account; and, because PROLOG's behaviour includes back-tracking and is more complicated than conventional program execution, verification of such programs can be more complicated than verification of conventional programs.

The dynamic assertion and retraction of clauses in PROLOG programs introduces a form of

destructive assignment, for the purpose of improving efficiency. This further complicates behaviour and, what is worse, destroys the behaviour-independent semantics of the program and significantly complicates program verification.

There are two main ways to improve this undesirable situation. (1) Insist on greater logic programming discipline within PROLOG programs. Where this leads to inefficiency, transform the logic program in a correctness-preserving way so that it runs efficiently without using the extralogical features of PROLOG. The resulting program will be less obviously correct, but at least its correctness can be demonstrated declaratively without needing to consider its behaviour. (2) In the longer term, we need to develop improved logic programming languages, which do not rely on extralogical features for the sake of efficiency. Special attention needs to be paid, in particular, to combatting the worst cases of inefficiency, which arise as a result of not being able to use programmer-controlled destructive assignment.

J. C. SHEPHERDSON (*School of Mathematics, University of Bristol, U.K.*). Can Professor Kowalski explain why one often says 'if' when one means 'if and only if'?

R. KOWALSKI. The 'if-' half of the 'if-and only if' half of definitions is the computationally useful part. On the one hand, the explicit use of 'if and only if' syntax obscures the pragmatic intention conveyed by using 'if'; on the other hand, using 'if' obscures the fact that 'only if' is also intended.

The use of 'if' syntax facilitates incremental program and specification development. It makes it easy to add more clauses to a definition, without explicitly retracting an explicitly stated 'only-if' assumption.

The use of 'if' when 'if and only if' is intended can be regarded as a special case of default reasoning, where some assumption is made 'by default'. Default reasoning is common in artificial intelligence, especially in systems based on frames, where it is generally regarded to be in conflict with classical logic. It can be argued that replacing 'if' halves of definitions by 'if and only if' gives a logical reconstruction of default reasoning.

D. SANNELLA (*Department of Computer Science, University of Edinburgh, U.K.*).

(1) I do not see why Professor Kowalski's logic program for subset is more amenable to parallel execution than an equivalent functional program would be. Functional programs offer the same opportunities for parallel execution as logic programs, and for exactly the same reasons.

(2) In his gcd example, Professor Kowalski seemed to indicate that the search for gcd(x, y) would be limited to numbers less than x and y. Putting a limit on the search requires either a careful choice of the specification of divides (which must state more or less explicitly that divides (x, y) is false if $x > y$) or else a lot of cleverness on the part of the compiler. The fact that some specifications lead to non-terminating programs suggests that the relation between specifications and programs is not as close as Professor Kowalski says.

R. KOWALSKI. (1) Logic programs provide for two main kinds of parallelism, and-parallelism and or-parallelism. And-parallelism is the analogue of parallel evaluation of subterms in functional programming languages. The opportunity for or-parallelism, however, results from the non-deterministic computation of relations instead of functions. It is the analogue of parallel

search for answers to queries in relational databases. This combination of opportunities for both and-parallelism and or-parallelism reflects the fact that logic programming can be regarded as generalizing and unifying functional programming and relational databases. The resulting unification also encompasses rule-based languages of the kind used for expert systems.

(2) There are two ways to 'execute' the specification of gcd: by means of a standard theorem-prover for first-order logic or by using negation by failure. Because negation by failure is incomplete in the general case, it may be incomplete in a particular case such as the specification of gcd, especially if, as D. Sannella points out, the definition of divides(x y) does not have some explicit or implicit constraint that $x < y$.

However, if we use any complete theorem-prover for first-order logic and the definition of gcd implies the existence of a gcd, for particular numbers x and y, then, by the completeness theorem for first-order logic, the theorem-prover is guaranteed to terminate successfully with an existence proof. Most theorem-provers, including all theorem-provers based on the resolution principle, will construct the gcd as a by-product of the proof. Thus, except for the inefficiency involved in using a general-purpose theorem-prover for computation, there is no difference in principle between a program and a complete specification, except for efficiency.

M. M. LEHMAN (*Department of Computing, Imperial College, London, U.K.*). Professor Kowalski said 'you cannot get a specification right the first time'. May I suggest that he should have said 'you cannot get a specification right', period.

A specification can be wrong in one of two ways (or both). It may, for example, be internally inconsistent, in that it includes (at least) two inconsistent statements, a situation that could be detected (for example by a verification procedure) calculable for a formal specification. But for a specification defining a program that is to be used to find a solution to a real problem currently of interest, the issue is one of satisfaction, not correctness. A specification may be *effectively* incorrect in that, as formulated, even though totally self consistent, the problem whose solution is specified is not that to which a solution is required.

In practice, this always happens. The very development of a specification changes one's viewpoint of the problem and of means for its solution. The very execution of a program changes one's view and definition of the problem one wishes to solve. Hence computer applications and the specifications that define programs that implement them are, by their very nature, evolutionary. A specification must be viewed as a dynamic object that always needs to be adapted. A specification is never absolutely right.

R. KOWALSKI. I very much agree with the spirit of Professor Lehman's remark: it is very difficult to get a specification absolutely right. However, the ability to execute a specification would help a great deal to test its assumptions and to identify where they might be changed to avoid unacceptable consequences. The conventional software life cycle that waits to test a specification until it is implemented as a program is a very inefficient way of improving specifications.

Whether it is ever possible to get a specification absolutely right is not the main issue. What matters is that we should be easily able to change a specification when it is necessary or desirable to do so. The use of logic to formalize specifications provides a proof theoretic framework which facilitates the alteration of specifications to more adequately meet user requirements as well as to meet changing requirements.

D. PARK (*Computer Science Department, Warwick University, Coventry, U.K.*). Many people are reluctant to discard *procedural* programming concepts; perhaps this is because they see the execution of a program as primarily a *simulation* of a succession of events in the world, rather than as a process of *deduction* about what holds in one particular state of the world. Does the ideal programming language perhaps need concepts of both sorts? Is there some compromise position?

R. KOWALSKI. You have identified what is probably the most important, unresolved problem in logic programming. It is an instance of a more general problem, known as the 'frame problem', in artificial intelligence. There are various proposals and outlined solutions for this problem. I have discussed one of these proposals in Kowalski (1983). Although I am not certain what the right solution is, I hope that it will not involve procedural programming with explicit destructive assignment as we know it today.