THEOREM PROVING TECHNIQUES AND P-FUNCTIONS FOR LOGIC DESIGN AND LOGIC PROGRAMMING

by D. SNYERS and A. THAYSE

Philips Research Laboratory, Av. Van Becelaere 2 - Box 8, B-1170 Brussels, Belgium

Abstract

It is shown that theorem proving methods can lead to program synthesis and algorithm implementation by using pairs of logic laws: a deductive law for proving the theorem and a constructive law for synthesizing the program or algorithm. The construction of program schemata in an algorithmic programming environment and logic programming in a declarative programming environment appear as two particular and extreme cases of the model of pairs of laws acting on pairs of logic expressions. A systematic examination of deductive laws and of constructive laws is presented. The set of all possible pairs of laws provides us with a tool for classifying the different approaches for materializing algorithms (e.g. hardware, microprogrammed, algorithmic programming, declarative programming, deductive approach for recursive routines).

Computing Reviews Classification: 1.1.2.3./B.1.2.

1. Introduction

Given a problem, how do we find an algorithm for its solution? Once we have found an algorithm how can we derive a computer implementation of this algorithm? Questions of this nature are of interest to logic designers, to programmers and to theoretically oriented computer scientists. We shall examine various lines of research that attempt to answer questions such as these.

By algorithm synthesis we mean the transformation of an informal description of the problem into an algorithm (which solves the problem) expressed in an implementation language. By implementation language we mean a description language which corresponds to or which immediately leads to a computer implementation. The computer implementation can be a hardware implementation made-up of integrated circuits and digital devices; it can also be a programmed implementation in a high-level programming language. This high-level language can in turn be an algorithmic language of the Algol-Pascal family or a language aimed at symbolic computations such e.g. the Lisp functional language or the Prolog logic programming language. In any case the synthesis of an algorithm can always be viewed as the transformation of a specification given in an informal language into an equivalent specification given in an implementation language.
Theorem proving techniques and P-functions

In a series of papers\(^1\)\(^-\)\(^4\) the P-function calculus was introduced as a theoretical framework for performing this transformation from an informal language into an implementation language. This calculus is connected to the Glushkov model of computation (also called algorithmic state machine) which was proposed to describe and implement algorithms according to the finite state-machine (or finite automaton) model\(^5\)\(^-\)\(^8\).

The purpose of the P-function calculus is to develop transformation techniques which allow us to obtain binary programs (i.e. programs made up of decision instructions of the type \textit{if then else} and of execution instructions of the type \textit{do}) from a problem or an algorithm specification given in an informal language. The binary program leads in turn to an implementation of the algorithm in a chosen technology, e.g. hardware (by means of logical components), microprogrammed (by means of address decoders, Read only memories, sequencers) or programmed; when a programmed realization is chosen, the design corresponds to an implementation of the algorithm in an algorithmic language\(^2\)\(^-\)\(^6\),\(^9\).

At the same time Manna and Waldinger\(^10\) proposed a deductive approach to program synthesis. This approach regards program synthesis as a theorem proving task and relies on a theorem proving method that combines the features of transformation rules, unification and mathematical induction within a single framework. The result of the theorem proving approach is the description of an applicative program i.e. a program which yields an output but produces no side effects.

Besides the deductive approach to program synthesis, theorem proving also constitutes the basic program execution mechanism in logic programming. The main purpose of the present paper is to establish a close relationship between the theorem proving approach to algorithm synthesis and the P-function calculus. This relationship allows us to show that algorithm implementation models as different as algorithmic state machine and logic programming can be approached via the same logic transformation laws.

The paper is organized as follows:

\textit{Section 2} contains an introduction to algorithm synthesis and connects this synthesis to logic transformations occurring in the course of the theorem proving process.

\textit{Section 3} shows how constructive theorem proving together with algorithm implementation naturally leads to the concepts of pair of functions (P-functions) and of pair of laws acting on these P-functions.

\textit{Section 4} is devoted to an exhaustive review of the logic deductive laws that can be used for algorithm synthesis. In particular, we show how pairs of logic laws (deductive and constructive laws) can be used as a tool for classifying
various types of computer implementations and of computer languages. The notations used in this paper are those of mathematical logic:

- \( \lor \) : sum or disjunction
- \( \land \) : product or conjunction
- \( \rightarrow \) : implication
- \( \leftrightarrow \) : equivalence
- \( \neg \) : negation
- \( \oplus \) : ring sum
- \( \Rightarrow \) : substitution.

2. Types of computer implementations for algorithms

As mentioned in the introduction, the synthesis of an algorithm can be viewed as the transformation of a specification given in an informal language into an equivalent specification given in an implementation language (fig. 1a). The logic languages such as propositional logic language and the first-order logic language play an important role in the transformation process schematically represented in fig. 1. Logic languages are often used as intermediates between informal description and implementation languages (fig. 1b).

![Diagram](image)

Fig. 1. Computer algorithm synthesis scheme.
First of all, remember that mathematical logic has its origin in the dream of Leibniz of a universal symbolic calculus which could encompass all mental activity of a logically rigorous nature, in particular, the whole of mathematics. Consequently almost all problems and algorithms can easily be expressed in terms of a logic language. Moreover, most of the computer structures and primitives are easily expressed in terms of logic. Switching theory can be considered as the basis for logic design and hence for hardware implementation; the basic decision primitive of algorithmic languages is the logic operation if then else. More fundamentally logic programming requires us to describe the logical structure of problems in a first-order logic language. This means that in logic programming, first-order logic expressions become executable instructions of a language. In summary, since logic languages are both universal languages for describing mathematics (and thus problems and algorithms) and computer implementation-like languages, they will play a key role in the transformation process between informal descriptions of problems and their computer implementations.

In a first step we shall transform the informal description of a problem into an equivalent description in a logic language (e.g. propositional logic or first-order logic). Then we shall transform the logic language description into a structured logic language description, i.e. a description from which an algorithm and its implementation can be deduced. We shall see that the appropriate type of structured logic language strongly depends on the chosen implementation. The transformation process of fig. 1a has to be modified: the synthesis will be performed in three steps as shown in fig. 1b. We illustrate the transformation process of fig. 1b by means of four examples gathered at the end of this section.

Example 1 is taken from computer arithmetic. It deals with the realization of an algorithm which computes either the sum or the difference of two numbers given by their binary representation. It is a typical logic design problem which uses propositional logic (or Boolean algebra) as logic language. The chosen implementation language is made-up of two types of instructions: decision instructions of the type if then else and execution instructions of the type do^11). We know that this type of language leads to wired, microprogrammed and programmed (in an algorithmic language) implementations of algorithms^2,5,6,8). The transformation between logic language and structured logic language is based on the algorithmic state-machine model also called Glushkov model of computation^7,8,12-14). The logic transformation is obtained by using pairs of logic laws acting on pairs of logic functions, also called P-functions^1-4,9,15). Example 1 illustrates the right-hand branch of the synthesis diagram represented in fig. 2.
Fig. 2. The theorem proving approach.

Informal language

Grammar

Montague grammars

Logic language

First-order logic

Propositional logic

[Theorem proving, deduction]

Structured first-order logic

Structured propositional logic

[State machine, P-functions]

Recursive logic functions for applicative program synthesis

Algorithmic implementation, program schemata

Recursive routines for applicative programs:

Functional programming (Lisp)

Logic programming (Prolog)

Logic design (hardware)

Microprogramming

Algorithmic programming

Logic languages based on resolution (Prolog)

Logic languages based on natural deduction

Recursive routines for applicative programs:

Functional programming (Lisp)

Logic programming (Prolog)

(Example 3)

(Example 4)

(Example 2)

(Example 1)
Example 2 deals with the synthesis of a recursive routine which computes the quotient and the remainder of two integers. It is known that no functional language (such as Lisp) and no logic language (such as Prolog) could efficaciously work without the possibility of recursive calls. Since Prolog is a logic language which executes a subset of first-order logic expressions we have chosen first-order logic for both the logic language and the structured logic language. Example 2 illustrates another branch of the synthesis diagram of fig. 2. This branch represents the deductive approach to program synthesis as it was defined by Manna and Waldinger\(^\text{1)}\). This approach regards algorithm synthesis as a theorem proving task and relies on a theorem proving method that combines the features of transformation rules, mathematical induction and unification in first-order logic within a single framework. In this model, the logic transformation between logic language and structured logic language is based on Robinson’s *resolution rule*\(^\text{16)}\) or on some non-clausal deduction rules\(^\text{10,17,18)}\).

Example 3 deals with a syntactic analysis problem; given a grammar and a vocabulary we want to verify whether a given list of words from the vocabulary is a legal (or syntactically correct) sentence of the grammar. When dealing with problems evolving from natural language processing we shall choose context-free grammars, or their first-order logic extension: define clause grammars, as structured logic language. Observe that grammars constitute already a structured formalism for the description of a natural language. Hence the writing of a grammar in logic constitutes a structured logic language. The syntactic rules (or productions) of logic grammars are a particular type of first-order logic expressions\(^\text{19-21)}\). The syntactic rules of these grammars are executable instructions of the logic programming language Prolog\(^\text{22)}\). Example 3 illustrates the left-hand side branch of the synthesis diagram of fig. 2, i.e. the synthesis of an algorithm by means of a logic programming language based on resolution. Logic programming differs fundamentally from algorithmic programming in requiring us to describe the logical structures of problems (in example 3, the grammar structure) rather than making us prescribe how the computer has to go about in solving them. Hence, in order to synthesize an algorithm by means of a logic programming implementation, we have first to transform the description of the problem at hand in terms of a collection of grammar rules. This transformation constitutes generally a straightforward way for solving problems related to natural languages processing. In general however, the transformation of a problem to be solved in terms of a set of grammar rules constitutes a round-about way to synthesize an algorithm. In order to overcome this difficulty the logic programming approach followed in the present paper investigates the feasibility of defining efficient languages based
on natural deduction. The features of these languages will be introduced after
the logic transformation, which translates a logic language into a structured
logic language, has been defined.

As mentioned above when the logic language is propositional logic, the algo-
rithm synthesis process is formalized by the Glushkov model of computation
while the logic transformation is formalized by pairs of logic laws (a deductive
law and a constructive law) acting on pairs of logic functions (P-functions). The
result of the transformation is a program schema (made up of two types of
instructions) which in turns leads to wired, microprogrammed and (algo-
rithmic) programmed implementations. We show in sec. 2 that the technique of
pairs of logic laws exactly corresponds to some program synthesis methods
based on mechanical theorem proving techniques. More precisely, the first
logic law proves the theorem while the second logic law constructs the desired
algorithm. The early work in program synthesis relied strongly on mechanical
theorem-proving technique\textsuperscript{23,24); however, the difficulty of representing the
principle of mathematical induction in a resolution framework hampered these
systems in the formation of programs with iterative or recursive loops.

More recently, Manna and Waldinger\textsuperscript{10,26,28}) described a theorem-proving
approach for program synthesis that combines mathematical induction and
logic transformation rules within a single deductive system which is used for
the construction of recursive routines. In summary (as indicated in fig. 2) the
transformation between a logic and a structured logic language can be per-
formed either by using transformation laws acting on P-functions\textsuperscript{1) or by
using the deductive approach by Manna and Waldinger\textsuperscript{10). Both approaches
make use of theorem proving techniques which express the algorithm design in
terms of proving that a sum of logic functions is valid (i.e. \textit{identically true}) or
that a product of logic functions is inconsistent (i.e. \textit{identically false}). Observe
now that Prolog, and more generally any logic programming language, are
automatic theorem provers; a Prolog program can be interpreted as a set of
first-order logic Horn clauses (or sums of literals) and executing a program is
reduced to proving that the product of these clauses is inconsistent. It appears
thus that theorem proving can be considered as a unified mathematical tool
for algorithmic state machine design, program synthesis by a deductive ap-
proach and logic programming. In algorithmic state machine design and de-
ductive program synthesis, theorem proving is used for synthesizing programs
while in logic programming, theorem proving is the basis of the execution
mechanism of the language itself. Let us now consider logic programming in a
more detailed way.

There are several quite distinct approaches to logic programming corres-
ponding to different ways of thinking about proofs. Within each of the ap-
Theorem proving techniques and P-functions

proaches there are several variations corresponding to the specific deductive laws that are executable by the language. Further down we shall ignore the minor variations. The different basic approaches are important, though, for different philosophies of proof lend themselves to different types of languages. It is however important to remember that, while there are many notions of proof, there is only one notion of provability for first-order logic, as the completeness theorem by Gödel shows\textsuperscript{27,28}. The first type of logic programming language that has been developed is the Prolog type language. Prolog is based on a very simple but efficient proof procedure which uses the resolution as unique inference (or execution rule). This type of language is based on a Hilbert-style of theorem proving, where the emphasis is on logical axioms keeping the rules of inference at a minimum. The inference rule in a Prolog language is the resolution which is a logic operation which eliminates literals (or variables) at each execution. The Prolog-like languages are easy to define and admit simple proof strategies but are difficult to use. We have already pointed out that, in order to synthesize an algorithm by means of a Prolog implementation, we have first to transform the description of the problem at hand in terms of a collection of grammar rules, which are particular types of Skolemized first-order logic expressions. General logic expressions that are natural to write when synthesizing algorithms cannot directly be expressed in terms of Prolog instructions. Therefore a logic programming class of languages based on a natural deduction system of Gentzen's type\textsuperscript{27,28} has recently been investigated\textsuperscript{29,30}. Gentzen's system emphasizes the importance of inference rules, reducing the role of logical axioms to an absolute minimum. Instead of acting on literals (as resolution did) the Gentzen's rules of inference act on logical operations (\lor, \land, \neg, \rightarrow, \forall, \exists), which can be introduced or suppressed during the deduction steps. These inference rules lead to more direct proof procedures than resolution did. From a logic programming point of view, the use of Gentzen's inference rules allows us to interpret quantified logic expressions as program instructions.

The use of universal quantification and of various logical operations, such as equivalence and implication, in the definition of instructions enables us to write interesting types of programs in application areas such as queries to data bases (see example 4). Such programs are awkward (or impossible) to express in logic languages based on resolution without using metalogical, or extra logical features. This does not mean that logic programming languages based on resolution cannot execute instructions which are functions or which contain various logical operations such as equivalence or implication: e.g. the Tablog language defined by Malachi, Manna and Waldinger\textsuperscript{26} executes first-order logic formulas including equality, negation and equivalence. Logic pro-
gramming based on Gentzen's deduction system incorporate, however, these features in a more coherent way; it must also be noted that logic programming based on natural deduction remains at the present time at an experimental level.

Let us finally point out that natural language processing has been connected to first-order logic through the so-called Montague grammars\(^3\)). The syntactic rules of these grammars are expressed in terms of logic expressions made-up of quantifiers, equivalence and implication among others. A logic programming language based on natural deduction rules could thus be used in a certain direct way to perform natural language processing.

The syntactic scheme of fig. 2 summarizes the contents of this section.

2.1. Examples
Example 1: Synthesis of a controlled Add/Subtract cell

**Informal language\(^3\)**

Synthesize an algorithm which computes either the sum or the difference of two numbers given by their binary representation. The algorithm uses two binary digits \(X_1\) and \(X_2\), which are the bits of the first and of the second number respectively, and a carry digit \(X_3\); these inputs produce a 4-dimension vector \([S_0, C_0, S_1, C_1]\) as output:

- \(S_0\): sum function equal to 1 iff an odd number of variables \(X_1, X_2, X_3\) is 1;
- \(C_0\): carry function equal to 1 iff at least two of the variables \(X_1, X_2, X_3\) are 1;
- \(S_1\): difference function equal to 1 iff an even number of variables \(X_1, X_2, X_3\) is 1;
- \(C_1\): difference carry equal to 1 iff at least two of the variables \(X_1, \neg X_2, X_3\) is 1.

**Logic language** (propositional logic or Boolean logic)

| Sum function   | \(S_0 = X_1 \oplus X_2 \oplus X_3\); |
| Sum carry      | \(C_0 = X_1 \land X_3 \lor X_2 \land X_3\); |
| Difference function | \(S_1 = X_1 \land \neg X_2 \land X_3\); |
| Difference carry | \(C_1 = X_1 \lor X_2 \lor \neg X_2 \land X_3\). |

**Output vector**

\[\sigma_1 = [0010] \text{ if } \neg S_0 \land C_0 \land \neg S_1 \land C_1 = f_1 = \neg x_1 \land \neg x_2 \land x_3 \text{ is 1} \]
\[\sigma_2 = [1001] \text{ if } S_0 \land \neg C_0 \land S_1 \land C_1 = f_2 = \neg x_2 (x_1 \oplus x_3) \text{ is 1} \]
\[\sigma_3 = [0111] \text{ if } \neg S_0 \land C_0 \land S_1 \land C_1 = f_3 = x_1 \land \neg x_2 \land x_3 \text{ is 1} \]
\[\sigma_4 = [0110] \text{ if } \neg S_0 \land C_0 \land \neg S_1 \land C_1 = f_4 = x_2 (x_1 \oplus x_3) \text{ is 1} \]
\[\sigma_5 = [1000] \text{ if } S_0 \land \neg C_0 \land \neg S_1 \land C_1 = f_5 = \neg x_1 \land x_2 \land \neg x_3 \text{ is 1} \]
\[\sigma_6 = [1101] \text{ if } S_0 \land C_0 \land \neg S_1 \land C_1 = f_6 = x_1 x_2 x_3 \text{ is 1}. \]

**Structured logic language** (structure formulas of propositional logic)

Algorithm \(N = x_3 [x_1 (\neg x_2 \sigma_3 \lor x_2 \sigma_6) \lor \neg x_1 (\neg x_2 \sigma_2 \lor x_2 \sigma_4)] \lor \)
\[\neg x_3 [x_1 (\neg x_2 \sigma_2 \lor x_2 \sigma_4) \lor \neg x_1 (\neg x_1 \sigma_1 \lor x_2 \sigma_6)].\]
Theorem proving techniques and P-functions

Implementation language (for wired or explicit programmed implementations)

N if $x_3$
then if $x_1$
then if $x_2$ then do $\sigma_6$
else do $\sigma_3$
else if $x_2$ then do $\sigma_4$
else do $\sigma_2$
else if $x_1$
then if $x_2$ then do $\sigma_4$
else do $\sigma_2$
else if $x_2$ then do $\sigma_5$
else do $\sigma_1$

Example 2: Synthesis of a recursive routine\(^{10}\)

Informal language

Synthesize an algorithm which computes the integer quotient $q$ and the remainder $r$ of a nonnegative integer $a$ by a positive integer $b$.

Logic language (first-order logic or predicate logic)

$$(\forall a)(\forall b) [(a \geq 0) (b > 0) \rightarrow (\exists q)(\exists r) (a = bq + r) (r \geq 0) (b > r)].$$

Structured logic language (recursive forms of first-order logic)

quotient $(a, b) = \text{if } (a < b) \text{ then } 0 \text{ else quotient } (a - b, b) + 1$
remainder $(a, b) = \text{if } (a < b) \text{ then } a \text{ else remainder } (a - b, b)$.

Implementation language

LISP: (def quotient(a b)
   (cond((a < b)0)
         (t(plus quotient(a - b)b)1)))

PROLOG: quotient(A,B,0) :- A < B
   quotient(A,B,QQ) :- AA is A - B
                  quotient(AA,B,Q),
                  QQ is Q + 1

Example 3: Syntactic analysis of sentences

Informal language

Synthesize an algorithm which can be used either to generate all the valid sentences of a context-free grammar, or to recognize whether a string of words
is a valid sentence of this context-free grammar. Construct moreover the parse tree (which associates to each word or group of words a syntactic category) of the generated or of the recognized sentence. The words of the vocabulary are: erase, print, the, a, next, first, last, line, word, character, of.

Structured logic language (context-free grammar)

- sentence -> verb, noun-phrase
- noun-phrase -> article, adjective, noun
- noun-phrase -> article, adjective, noun, preposition, noun-phrase
- verb -> erase
- verb -> print
- article -> the
- article -> a
- adjective -> next
- adjective -> first
- adjective -> last
- noun -> line
- noun -> word
- noun -> character
- preposition -> of

Implementation language (Prolog implementation of a context-free grammar)

1. sentence (sn(V,N,Np),X,Y) :- verb(V,X,Z), noun-phrase (Np,Z,Y).
4. verb (v(erase), [erase|S],S).
5. verb (v(print), [print|S],S).
6. article (art(the), [the|W],W).
7. article (art(a), [a|W],W).
8. adjective (adj(next), [next|S],S).
9. adjective (adj(first), [first|S],S).
10. adjective (adj(last), [last|S],S).
Theorem proving techniques and P-functions

(11): noun (n(line), [line|Y],Y).
(12): noun (n(word), [word|Y],Y).
(13): noun (n(character), [character|Y],Y).
(14): preposition (prep(of), [of|U],U).

Example 429)

Informal language

Synthesize an algorithm which, from a list of students and of a list of courses, can select the students taking all the courses of a given type (e.g. the mathematics courses).

Logic language and implementation language

Instruction (1) states that x is a 'math-major' if and only if he takes all the 'math-courses';
Instruction (2) defines the relation 'person x' takes 'course y' by means of the predicate 'takes(x,y)';
Instruction (3) states all the 'math-courses'.

(1): math-major(x) \iff \forall y (math-course(y) \rightarrow takes(x,y)).
(2): takes(x,y) \iff ((x = D) \land (y = C3)) \lor ((x = J) \land (y = C1)) \lor ((x = J) \land (y = C3)).
(3): math-course(z) \iff (z = C1) \lor (z = C3).

3. From theorem proving to P-functions

Automatic theorem proving is an important subject in artificial intelligence: it has been applied to many areas such as program analysis, program synthesis, deductive question-answering systems and problem solving systems33-35). As mentioned in sec. 2, we want to use logic to represent problems and to obtain their solutions, i.e. implementations of algorithms in our case. Moreover, we want to obtain these implementations by means of some theorem proving methods.

In a theorem proving method we essentially have to prove that a formula logically follows from other formulas. We shall call a statement that a formula logically follows from formulas a theorem. A demonstration that a theorem is true, in the sense that a formula logically follows from other formulas, will be called a proof of the theorem. The problem of automatic theorem proving is to consider automatic methods for finding proofs of theorems.

There are many problems that can be conveniently transformed into theorem-proving problems; as long as algorithm implementation is considered, theorem proving can be used as follows:
D. Snyers and A. Thayse

— find a constructive proof of the desired algorithm existence (theorem proving part);
— derive the algorithm from the constructive proof (algorithm synthesis part).

Stated otherwise, if a constructive proof can be found that an algorithm does exist, then derive this algorithm from the constructive proof.

If we want to use logic transformation rules in the theorem proving part, we have first to express the algorithm design problem in terms of proving that a conclusion is a logical consequence of a finite set of axioms.

**Definition**

Given logic formulas \( A_1(x), \ldots, A_q(x) \) and \( C(x) \) of the logic variables \( x = (x_1, \ldots, x_n) \), \( C \) is said to be a logical consequence of \( A_1, \ldots, A_q \) if and only if for any value \( e \) of \( x \) in which \( A_1 \land \ldots \land A_q \) is true (is equal to 1), \( C \) is also true. \( A_1, \ldots, A_q \) are called axioms of \( C \).

**Theorem 1** (forward deduction theorem)

Given formulas \( A_1, \ldots, A_q \) and a formula \( C \), \( C \) is a logical consequence of \( A_1, \ldots, A_q \) if and only if the formula \( (A_1 \land \ldots \land A_q \land \neg C) \) is inconsistent (i.e. identically false or equal to 0).

**Theorem 2** (backward deduction theorem)

Given formulas \( A_1, \ldots, A_q \) and a formula \( C \), \( C \) is a logical consequence of \( A_1, \ldots, A_q \) if and only if the formula \( (\neg A_1 \lor \ldots \lor \neg A_q \lor C) \) is valid (i.e. identically true or equal to 1).

Theorems 1 and 2 show that proving that a particular formula is a logical consequence of a finite set of formulas is equivalent to proving that a related formula is valid or inconsistent.

If \( C \) is a logical consequence of \( A_1, \ldots, A_q \), the formula

\[
(A_1 \land A_2 \land \ldots \land A_q) \to C
\]

is called a theorem, and \( C \) is called the conclusion of the theorem.

The theorem proving part of the algorithm design problem can thus be stated as follows:

From a finite set of logic expressions derive (if possible) the logic constant false (0) or true (1) by means of logic transformation laws, called deductive laws (see sec. 4).

Derivation of an algorithm using a theorem-proving technique requires first that the proof of the theorem must be constructive; moreover it is clear that this proof construction should also produce the desired solution (or algorithm)
Theorem proving techniques and \(P\)-functions

at the time of completion of the proof. This means that the proof of the theorem must know how to find an algorithm described in the required structured logic language. In this respect we shall associate with each axiom and conclusion of the theorem an output entry. This output entry will have no bearing on the proof itself but will record the algorithm part that has been constructed at each stage of the derivation. It will thus be required that, at the time of completion of the proof (having obtained the logic constant false or true), the output entry will produce the expected algorithm described in the adequate structured logic language.

Associating an output entry to assertions and goals is a way of introducing the concept of 'a pair of functions. Remember that the concept of a pair of (logic) functions has been introduced\(^1\) in the context of algorithm synthesis in an algorithmic programming environment. With each step of the algorithm synthesis we associate a pair of logic functions, the left part (or domain) of which indicates what part of the algorithm remains described in a logic language while the right part (or codomain) indicates what part of the algorithm has already been transformed into a structured logic language. The interpretation, in terms of pairs of functions, of the logic transformation which derives a structured logic description from a logic description is indicated in fig. 3. Observe that in the \(P\)-function approach, the logic transformation laws become pairs of logic laws: deductive laws acting on the domain and constructive laws acting on the codomain. These laws will be defined in sec. 4.

The syntactic scheme of fig. 4 summarizes the theorem proving approach to algorithm synthesis and its connection to \(P\)-function theory.

Let us go a step further in the \(P\)-function formalism. The proof that a conclusion logically follows from a set of axioms can be obtained by using either the forward deduction theorem 1 or the backward deduction theorem 2. In a forward deduction process the proof is completed after obtaining the constant 0 from an iterative application of the forward deduction law (denoted \(\tau_i\)) to the axioms and to the negated conclusion. Let us denote the axioms and negated conclusion by \(f_1(x), f_2(x), \ldots, f_P(x)\); the following binary laws acting on formulas of the propositional logic are examples of forward deduction laws:

\[
\perp_{f_1}(f_k, f_1) = (x_i \lor f_k(x)) \land (\neg x_i \lor f_1(x)), \quad x_i \in x, \quad (1)
\]

\[
\perp_{f_1}(f_k, f_1) = f_k(x_i) \Rightarrow 0 \lor f_1(x_i) \Rightarrow 1), \quad x_i \in x, \quad (2)
\]

where a notation such as \(f_k(x_i) \Rightarrow e_i\) means that in \(f_k(x)\), the occurrence of the literals \(x_i\) and \(\neg x_i\) have to be replaced by the constants \(e_i\) and \(\neg e_i\), respectively.
P-function formalism

Fig. 3. The P-function formalism.

\[
\langle \text{algorithm description in a logic language}; \phi \rangle \quad \text{: initial description in a logic language}
\]

\[
\langle \text{axioms, conclusion: } f_i; \quad \text{logic constants and variables: } \phi \rangle
\]

\[
\text{deductive} \quad \text{laws } t_i \text{ or } \tau_i; \quad \text{constructive} \quad \text{laws } t^*_i
\]

\[
\langle \text{true or false} ; \quad \text{algorithm description in structured-logic} \rangle
\]

\[
\langle \phi ; \quad \text{algorithm description in a structured logic language} \rangle \quad \text{: final description in structured logic}
\]
Theorem proving techniques and P-functions

**Theorem proving**

1. Find a constructive proof of the desired algorithm existence
2. Express the algorithm design problem in terms of proving that a conclusion $G$ is the logical consequence of the finite set of axioms: $A_1, \ldots, A_q$
3. Prove either that:
   - $(\neg A_1 \lor \cdots \lor \neg A_q \lor G)$ is valid
   - $(A_1 \land \cdots \land A_q \land \neg G)$ is inconsistent
4. From a finite set of logic expressions derive (if possible) the logic constants 0 or 1 by means of deductive laws

**Algorithm synthesis**

1. If a constructive proof can be found that an algorithm does exist, then derive this algorithm from the constructive proof
2. From logic constant and/or from existentially quantified variables of the conclusion $G$, produce the desired algorithm (expressed in a structured logic language) by means of constructive laws

**Diagram**

- Logic language
  - Set of logic expressions
    - Deductive laws
      - 0 or 1
  - Logic
    - Constructive laws
      - Algorithm
        - Structured logic language

Fig. 4. Theorem proving and P-functions.
We prove in sec. 4 that an iterative application of the laws \( \land_i (1 \leq i \leq n) \) (or more generally of any forward deduction laws \( \tau_i (1 \leq i < n) \)) produces the constant 0 (or false) if and only if \( f_1 \land \ldots \land f_p \) is inconsistent.

Dually, in a backward deduction process the proof is completed after obtaining the constant 1 from an iterative application of the backward deduction laws (denoted \( \Gamma_i \)) to the conclusion and to the negated axioms. Let us denote the negated axioms and conclusion by \( f_1(x), f_2(x), \ldots, f_p(x) \); the following binary laws acting on formulas of the propositional logic are examples of backward deduction laws:

\[
\Gamma_i(f_k, f_i) = (\neg x_i \land f_k(x)) \lor (x_i \land f_i(x)), \quad x_i \in x, \\
\Gamma'_i(f_k, f_i) = f_k(x_i \Rightarrow 0) \land f_i(x_i \Rightarrow 1), \quad x_i \in x.
\]

We prove in sec. 4 that an iterative application of the laws \( \Gamma'_i (1 \leq i \leq n) \) (or more generally of any backward deduction laws \( \Gamma_i (1 \leq i \leq n) \)) produces the constant 1 (or true) if and only if \( f_1 \lor \ldots \lor f_p \) is valid.

Consider now the algorithm synthesis part (or codomain) of the P-function. We shall show in sec. 4 that the synthesis of the algorithm will be obtained from an iterative application (the same iteration as in the theorem proving part) of constructive laws (denoted \( \Gamma_i^* \)) to some functions \( \phi_1, \ldots, \phi_p \) appearing in the codomain of the P-functions \( \langle f_1; \phi_1 \rangle, \ldots, \langle f_p; \phi_p \rangle \). This notation \( \langle f_j; \phi_j \rangle \) will be used through this text for denoting a pair of logic functions having \( f_j \) as domain and \( \phi_j \) as codomain. The law \( \Gamma_i \) is a typical construction law for problems evolving from implementation in an algorithmic language (see example 1 and sec. 4).

In summary, the \( \langle \text{theorem proving}; \text{algorithm synthesis} \rangle \) processes can be represented in terms of a transformation from an initial system of P-functions to a final P-function, using pairs of \( \langle \text{deductive; constructive} \rangle \) laws, i.e.

\[
\begin{align*}
\text{initial system of P-functions} & \quad \rightarrow \quad \text{final P-function} \\
\langle f_1; \phi_1 \rangle & \quad \rightarrow \quad \langle \Gamma_i \text{ or } \tau_i; \Gamma_i^* \rangle \\
\vdots & \\
\langle f_p; \phi_p \rangle & \quad \rightarrow \quad \langle 1 \text{ or } 0; \text{ algorithm description} \rangle \\
(1 \leq i \leq n) &
\end{align*}
\]

Let us now introduce the first-order logic extension of the deduction laws (see also sec. 4). Let \( f_1, \ldots, f_p \) be first-order logic formulas in the predicates \( x = (x_1, \ldots, x_n) \); if the substitution \( \theta \) is a most general unifier for the instances of the predicate \( x_i \in x \) appearing in \( f_k \) and \( f_i \), the first-order extensions of the deductive laws are defined as follows:
Theorem proving techniques and P-functions

\[ \perp_{ib}(f_k, f_l) = (x; \theta \lor f_k \theta) \land (\neg x; \theta \lor f_l \theta), \] (6)

\[ \perp_{ib}(f_k, f_l) = f_k \theta(x; \theta \Rightarrow 0) \lor f_l \theta(x; \theta \Rightarrow 1), \] (7)

\[ \tau_{ib}(f_k, f_l) = (\neg x; \theta \land f_k \theta) \lor (x; \theta \land f_l \theta), \] (8)

\[ \tau_{ib}(f_k, f_l) = f_k \theta(x; \theta \Rightarrow 0) \land f_l \theta(x; \theta \Rightarrow 1), \] (9)

As mentioned in sec. 2, first-order logic is used in logic programming. The constructive law acting on the codomain of the P-function is in this case the unification operation \( \theta \) (see sec. 4.2); the pairs of laws acting on P-functions are thus of the form

\[ \langle \text{backward law } \perp_{ib} \text{ or forward law } \tau_{ib}; \theta \rangle. \]

In logic programming, we are in general considering the synthesis of programs whose specifications have the form of a proof that a conclusion \( C(x, z) \) logically follows from axioms \( A_1, \ldots, A_{p-1}(x) \), i.e. that

\[ (\forall x) ((A_1(x) \land \ldots \land A_{p-1}(x)) \rightarrow (\exists z) C(x, z)). \]

The conclusion \( C \) is a formula representing the relationship between the program input (represented by the vector \( x \)) and the program output (represented by the vector \( z \)). We can thus state the program synthesis problem as the problem of finding a constructive proof that the formula \( C(x, z) \) logically follows from the axioms \( A_1, \ldots, A_{p-1} \), and if it does we want an instance \( z \) as a composition of functions of \( x \); if \( \theta \) represents the composition of the successive substitutions \( \theta_1, \ldots, \theta_q \) that occur during the theorem proving process, the program output in logic programming is the instance \( z\theta \).

The \( \langle \text{theorem proving; program computation} \rangle \) process in logic programming can thus be represented in terms of a transformation from an initial system of P-functions to a final P-function, i.e.:

\[ \langle f_1(x) ; 0 \rangle \quad \rightarrow \quad \langle \tau_{ib}, \text{or } \tau_{ib}; \theta \rangle \]

\[ \langle f_{p-1}(x) ; 0 \rangle \quad \rightarrow \quad \langle 1 \text{ or } 0; \text{program output} \rangle \]

Example 3 illustrates this transformation.

Let us finally consider the synthesis of recursive routines for applicative programs. It is well known\(^{10,84} \) that structural induction is of special importance for theorem proving methods intended for recursive program synthesis because it is the application of induction which introduces recursive calls into
Fig. 5. The deduction steps of example 1.

\[ A_1 = \langle f_1; \sigma_1 \rangle \quad A_5 = \langle f_5; \sigma_5 \rangle \quad A_2 = \langle f_2; \sigma_2 \rangle \quad A_4 = \langle f_4; \sigma_4 \rangle \quad A_3 = \langle f_3; \sigma_3 \rangle \quad A_6 = \langle f_6; \sigma_6 \rangle \]

\[ \langle \neg x_1 \neg x_2 \neg x_3; \sigma_1 \rangle \langle \neg x_1 x_2 \neg x_3; \sigma_6 \rangle \langle \neg x_2 (\neg x_1 \Theta x_3); \sigma_2 \rangle \langle x_2 (\neg x_1 \Theta x_3); \sigma_4 \rangle \langle x_1 \neg x_2 x_3; \sigma_3 \rangle \langle x_1 x_2 x_3; \sigma_6 \rangle \]

\[ \langle T_2'; T_2 \rangle \quad \langle T_2'; T_2 \rangle \quad \langle T_2'; T_2 \rangle \]

\[ B_3 = \langle \neg x_1 \neg x_3; x_2 \sigma_6 \vee \neg x_2 \sigma_1 \rangle \quad B_1 = \langle \neg x_1 \Theta x_3; x_2 \sigma_4 \vee \neg x_2 \sigma_3 \rangle \quad B_2 = \langle x_1 x_3; x_2 \sigma_6 \vee \neg x_2 \sigma_3 \rangle \]

\[ \langle T_1'; T_1 \rangle \quad \langle T_1'; T_1 \rangle \quad \langle T_1'; T_1 \rangle \]

\[ C_1 = \langle \neg x_3; x_1 (x_2 \sigma_4 \vee \neg x_2 \sigma_2) \vee \neg x_1 (x_2 \sigma_6 \vee \neg x_2 \sigma_1) \rangle \quad C_2 = \langle x_3; x_1 (x_2 \sigma_6 \vee \neg x_2 \sigma_3) \vee \neg x_1 (x_2 \sigma_4 \vee \neg x_2 \sigma_3) \rangle \]

\[ \langle T_3'; T_3 \rangle \]

\[ D = \langle 1; x_3 [x_1 (x_2 \sigma_6 \vee \neg x_2 \sigma_3) \vee \neg x_1 (x_2 \sigma_4 \vee \neg x_2 \sigma_2)] \neg x_3 [(x_2 \sigma_4 \vee \neg x_2 \sigma_2) \vee \neg x_1 (x_2 \sigma_5 \vee \neg x_2 \sigma_1)] \rangle \]
Theorem proving techniques and P-functions

the program being synthesized. We can describe structural induction as follows. In attempting to prove that a conclusion $C$ logically follows from axioms $A_1, \ldots, A_q$, we can assume that

$$\forall x \left( \bigwedge_j A_j(x) \rightarrow (\exists z) \ C(x, z) \right)$$

can be inductively verified for all $y = (y_1, \ldots, y_n)$ that are strictly less than $x$ with respect to a given ordering relation (i.e. for all $y$ for which a formula $R(x, y)$ is true). The following recursive axiom holds thus as induction hypothesis

$$R(x, y) \rightarrow (\bigwedge_i A_i(y) \rightarrow C(y, g(y))).$$

The recursive program synthesis is carried out by adding a recursion axiom $f_R(x, y)$ and by performing the following transformation:

\[
\begin{array}{l}
\text{initial system of P-functions} \\ \langle f_1(x) ; 0 \rangle \\ \vdots \\ \langle f_{p-1}(x) ; 0 \rangle \\ \langle f_R(x, y) ; 0 \rangle \\ \langle f_p(x, z) ; z \rangle
\end{array} \rightarrow
\begin{array}{l}
\langle 1 \ or \ 0; \ algorithm \ description \rangle \\ \langle \tau_{i_0}; \ or \ \tau_{i_0}; \theta \rangle
\end{array}
\]

Example 2 illustrates this transformation.

3.1. Examples

Example 1 (continuation)

The logic transformation based on theorem proving is written in the P-function formalism as follows:

\[
\begin{array}{l}
\langle f_1; \sigma_1 \rangle \\ \langle f_2; \sigma_2 \rangle \\ \langle f_3; \sigma_3 \rangle \\ \langle f_4; \sigma_4 \rangle \\ \langle f_5; \sigma_5 \rangle \\ \langle f_6; \sigma_6 \rangle
\end{array} \rightarrow
\begin{array}{l}
\langle 1; x_3[x_1(\neg x_2\sigma_2 \lor x_3\sigma_3) \lor \neg x_1(\neg x_2\sigma_2 \lor x_2\sigma_4)] \lor \\
\neg x_3[x_1(\neg x_2\sigma_2 \lor x_3\sigma_3) \lor \neg x_1(\neg x_2\sigma_1 \lor x_2\sigma_4)] \rangle.
\end{array}
\]

The interconnection of the elementary steps of this transformation is indicated in fig. 5.

Example 2 (continuation)

The program synthesis problem reduces to finding a constructive proof for the following implication:
D. Snyers and A. Thayse

\[(\forall a)(\forall b) [(a \geq 0) (b > 0) \rightarrow (\exists r) (\exists q) (a = bq + r) (r \geq 0) (b > r)]\]

and to deriving formulas \(q = q(a, b), r = r(a, b)\) which satisfy this proof. For a backward deduction procedure, the initial system of P-functions is:

\[A_1: \langle \neg \text{axiom}; 0, 0 \rangle = \langle (a < 0) \lor (b \leq 0); 0, 0 \rangle, \quad (11)\]

\[A_2: \langle \neg \text{recursion}; 0, 0 \rangle = \langle R(a, b, a', b') (a' \geq 0) (b' > 0) \land \neg [(a' = b'q'(a', b') + r'(a', b')) (0 \leq r'(a', b')) (r'(a', b') < b')]; 0, 0 \rangle, \quad (12)\]

\[A_3: \langle \text{conclusion}; q, r \rangle = \langle (a = bq + r) (r \geq 0) (b > r); q, r \rangle. \quad (13)\]

Before making use of the recursion (12) let us recall that the structure of a recursive definition is practically always the same:

— it is a conditional expression,
— simple cases are tested first,
— the last pair of the conditional expression, which often starts with the predicate 1 (or true) since all other cases have been exhausted, applies the defined function recursively to a simpler argument.

We have first to obtain the simple case; to this end we define a substitution: \(\theta_1 = [q = 0, r \Rightarrow a]\) and a binary law \(\tau\) with the variable \((0 \leq a)\) and the substitution \(\theta_1\) as arguments:

\[B_1 = \tau_{[0 \leq a]} \theta_1(A_1, A_2) = \langle (a < b); 0, a \rangle. \quad (14)\]

The domain covered by the simple case is characterized by the relation \((a < b) = \text{true}\); we want the recursive part to cover the remaining domain, i.e.: \((a \geq b) = \text{true}\). This is obtained by means of the substitution \([a' \Rightarrow a - b]\): the condition \((a' \geq 0)\) of (12) becomes in this way \((a - b \geq 0)\), or equivalently \((a \geq b)\).

The relation \((a' = b'q' + r')\) of (12) becomes \((a - b = b'q' + r')\) and the most general unifier between this expression and the relation \((a = bq + r)\) of (13) is the substitution \([q = q' + 1, b' \Rightarrow b, r \Rightarrow r']\). Moreover since \((a' \geq 0)(b' > 0) = (a \geq b)(b > 0)\), the only additional condition to be verified by the input vector is \((a \geq 0)\) and thus the ordering relation \(R(a, b, a', b')\) of (12) reduces to \((a \geq 0)\). We define the substitution \(\theta_2\) and the auxiliary variable \(y\) as follows:

\[\theta_2 = [a' \Rightarrow a - b, b' \Rightarrow b, q \Rightarrow q'(a', b') + 1, r \Rightarrow r'(a', b')], \quad y = (a = bq + r) (r \geq 0) (b > r), \quad B_2 = \tau_{\theta_2}(A_2, A_3) = \langle (a \geq 0) (b > 0) (a \geq b); q'(a - b, b) + 1, r'(a - b, b)\rangle. \]

The following P-functions are successively computed
\[
\begin{align*}
\langle \neg \text{axiom} ; 0,0 \rangle &= \langle (a < 0) \lor (b \leq 0) \rangle ; 0,0 \\
\langle \neg \text{induction axiom} ; 0,0 \rangle &= \langle R(a' > 0) (b' > 0) \neg [(a' = b'q'(a', b') + r'(a', b')) (0 < r'(a', b')) (r'(a', b') < b')] \rangle ; 0,0 \\
\langle \text{conclusion} ; q,r \rangle &= \langle (a = bq + r) (r > 0) (b > r) \rangle ; q,r
\end{align*}
\]

\(\langle T_{\Theta} ; \theta \rangle\)

\[\langle 1 ; q(a,b) \rangle = \text{if } (a < b) \text{ then } 0 \text{ else } q(a-b, b) + 1, r(a, b) \rangle = \text{if } (a < b) \text{ then } a \text{ else } r(a-b, b)\]

\[
\begin{align*}
\langle \text{axiom} ; 0,0 \rangle \langle \text{conclusion} ; q,r \rangle & \quad \langle \neg \text{induction axiom} ; 0,0 \rangle \langle \text{conclusion} ; q,r \rangle \\
\langle \text{T}(a \geq 0) ; \theta_1 \rangle & \quad \langle \text{T}(\text{r}(a = bq + r) (r > 0) (b > r)) ; \theta_2 \rangle \\
\langle \neg \text{axiom} ; 0,0 \rangle \langle (a \geq 0) (b > 0) (a \geq b) ; q'(a-b,b) + 1, r'(a-b,b) \rangle & \quad \langle \text{T}(a \geq 0) ; - \rangle \\
\langle \neg \text{axiom} ; 0,0 \rangle \langle (b > 0) (a \geq b) ; q'(a-b,b) + 1, r'(a-b,b) \rangle & \quad \langle \text{T}(b > 0) ; - \rangle \\
\langle (a < b) ; 0,a \rangle & \quad \langle a \geq b ; q'(a-b,b) + 1, r'(a-b,b) \rangle
\end{align*}
\]

\[
\begin{align*}
\langle \text{T}(a \geq b) ; - \rangle & \quad \theta_1 = \{q = 0, r = 0\} \\
\theta_2 = \{a' \Rightarrow a-b, b' \Rightarrow b, q = q'(a',b') + 1, r \Rightarrow r'(a',b')\}
\end{align*}
\]
D. Snyers and A. Thayse

\[ B_3 = T_{(a \geq 0)}(A_1, B_2) = \langle (b > 0) (a \geq b) ; q'(a - b, b) + 1, r'(a - b, b) \rangle, \]
\[ B_4 = T_{(b > 0)}(A_1, B_3) = \langle (a \geq b) ; q'(a - b, b) + 1, r'(a - b, b) \rangle, \]
\[ B_5 = T_{(a \geq b)}(B_1, B_4) = \langle 1 ; (a < b) 0 \lor (a \geq b) (q'(a - b, b) + 1), \]
\[ \hspace{1cm} (a < b) a \lor (a \geq b) r'(a - b, b), \]
\[ \hspace{1cm} = \langle 1 ; q'(a, b), r'(a, b) \rangle. \]

The interconnection of the elementary steps of this transformation is indicated in fig. 6.

Example 3 (continuation)

The Prolog program is formed by the 14 instructions (1)–(14) (see sec. 2); it can be used in three different ways.

In order to perform a syntactic analysis of a string of words, we type the following instruction:

(15_1):  
\[ ?-\text{sentence}(P, \text{[erase, the, last, word, of, the, next, line]}, \text{[]}). \]

This instruction means: find a sentence with a parse tree \( P \) and having \([\text{erase, the, last, word, of, the, next, line}]\) as initial list of words (before parsing) and the empty list: \([],\) as final list of words (after parsing). The instruction (15_1) constitutes the Prolog program input; the program output is the result of the

![Fig. 7. The parse tree of example 3.](image-url)
Fig. 8. The deduction steps of example 3.

Table: Deduction Steps

<table>
<thead>
<tr>
<th>Step</th>
<th>Unifications</th>
<th>Integrated Unifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>sn(V,NP) = P, X = Q, (1) = Y</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>article(Art,S,W) v v noun-phrase(NP,S,D) (2)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>adjective(Adj,W,R) v v noun(N,R,D) (6)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>noun-phrase(NP,S,D) (4)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>noun-phrase(NP,S,D) (15)</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>False</td>
<td></td>
</tr>
</tbody>
</table>
unification operations acting on the variable $P$, i.e.:

$$P = \text{sn(erase, art(the), adj(last), n(word), prep(of), np(art (the), adj(next), n(line)))}.$$  

The parse tree $P$ is also represented by fig. 7. If $x$ are the variables in the instructions (1)–(14), the Prolog program proves the following theorem:

$$(\forall x) (((1) \land \ldots \land (14)) \rightarrow (\exists P) \text{sentence}(P,[\text{erase, \ldots , line}], [])).$$

We can also generate a sentence from the parse tree; in this case the input instruction is:

$$(15): \text{?-sentence (sn(erase, np(art(the), adj(last), n(word))), Q, [])}.$$  
The program output is the sentence

$$Q = [\text{erase, the, last, word}].$$

Finally the Prolog program can also be used to generate valid sentences together with their parse tree; the input instruction is then

$$(15_a): \text{?-sentence}(P,Q, []).$$

This instruction means: find a sentence with a parse tree $P$ and having $Q$ as initial string of words and [] as final list of words; the program tries to prove the following theorem:

$$(\forall x) (((1) \land \ldots \land (14)) \rightarrow (\exists P) (\exists Q) \text{sentence}(P, Q, [])).$$

The program outputs are the instances of $P, Q$ for which the theorem is verified, i.e.

$$P = \text{sn(verb(erase), np(art (the), adj(next), n(character)))},$$

$$Q = [\text{erase, the, next, character}].$$

The execution of the program recording to the Prolog strategy is depicted in fig. 8. Fig. 8a constitutes the theorem proving part of the P-functions while fig. 8b constitutes the algorithm synthesis part of the P-functions. The results are obtained by means of 6 deduction steps; the deduction laws are omitted in fig. 8.

4. Laws for theorem proving and program synthesis

In the previous section we have discussed how to find algorithms by theorem proving. In this section, we shall consider proof laws and proof procedures.

An important approach to automatic theorem proving was given by Herbrand³⁶). By definition, a valid formula is a formula that is true under all
interpretations, i.e. for all values of its variables x. Herbrand developed an algorithm to find an interpretation that can falsify a given formula. However, if the given formula is indeed valid, no such interpretation can exist and his algorithm will halt after a finite number of steps. The method of Herbrand is the basis for most present automatic proof procedures. A major breakthrough was made by Robinson\textsuperscript{16}, who introduced the resolution principle. Resolution proof procedure is much more efficient than any earlier procedure. It works on a set of axioms and conclusion written as a product of clauses (or sums of literals). Since the introduction of the resolution principle, several refinements have been suggested in attempts to further increase its efficiency. Some of these refinements are non-clausal resolution procedures, i.e. resolution procedures working on axioms and conclusion written in terms of well-formed logic expressions\textsuperscript{1,10,18}.

4.1. Propositional logic

Let \( f_1(x), f_2(x), \ldots, f_p(x) \) be Boolean functions of the variables \( x = (x_1, \ldots, x_n) \). We define binary operations \( T_i \) acting on Boolean functions as follows (see also eq. (3)):

\[
T_i(f_k, f_l) = \neg x_i f_k(x) \lor x_i f_l(x). \tag{15}
\]

Let \( \langle g; h \rangle \) be a pair of Boolean functions or P-functions; let us take the pair of laws \( \langle T_i; T_i \rangle \) as deductive and constructive laws (see sec. 3).

**Theorem 3**

a. There is an iterative application of operations \( \langle T_i; T_l \rangle \), \( 1 \leq i \leq n \), which, when applied to the initial system of P-functions \( \langle f_j(x); \phi_j \rangle \), \( 1 \leq j \leq p \), will produce the P-function \( \langle 1; \lor f_j \phi_i \rangle \) (with \( f_j \leq f_j \lor \phi_j \)) if and only if \( \lor f_j = 1 \) (is valid); if moreover \( f_k f_l = 0 \) (is inconsistent) \( \forall k \neq l \), then \( f_j' = f_j \lor \phi_j \) and the resulting P-function is: \( \langle 1; \lor f_j \phi_j \rangle \).

b. There is an iterative application of operations \( \langle T_i; T_l \rangle \), \( 1 \leq i \leq n \), which, when applied to the initial system of P-functions \( \langle f_j(x); \phi_j \rangle \), \( 1 \leq j \leq p \), will produce the P-function \( \langle 0; \land (f_j'' \lor \phi_i) \rangle \), \( f_j'' \geq f_j \lor \phi_j \) if and only if \( \land f_j = 0 \) (is inconsistent); if moreover \( \neg f_k \lor \neg f_l = 1 \) (valid) \( \forall k \neq l \), then \( f_j'' = f_j \lor \phi_j \) and the resulting P-function is: \( \langle 0; \land (f_j \lor \phi_j) \rangle \).

**Proof** (see appendix A)

The constructive proof of theorem 3 provides us with the sequences of operations which produce the transformations.
The synthesis of algorithms by means of theorem proving methods moreover requires that the value 1 should be obtained in the domain of the final P-function (16) if \( \bigvee f_j \equiv 1 \) and that the value 0 should be obtained in the domain of the final P-function (17) if \( \bigwedge f_j \equiv 0 \). This means that rewriting (or simplification) rules should be introduced within the transformation \( T_i \), which could transform the expressions \( \bigvee f_j \) and \( \bigwedge f_j \) into 1 and 0, respectively.

\[
T_i(f_k, f_i) = \neg x_i f_k \vee x_i f_i \\
= \neg x_i f_k (x_i \Rightarrow 0) \vee x_i f_i (x_i \Rightarrow 1) \\
T_i(f_k, f_i) = (x_i \vee f_k) (\neg x_i \vee f_i) \\
= (x_i \vee f_k (x_i \Rightarrow 0)) (\neg x_i \vee f_i (x_i \Rightarrow 1))
\]

We prove in appendix B that the forms (19) and (18) of the law \( T_i \) provide us with the desired transformations (20) and (21) as long as the rewriting rules (1 \( \vee x = 1 \vee \neg x = 1 \)) and (0 \( \wedge x = 0 \wedge \neg x = 0 \)) are performed, respectively.

\[
\{\langle f_j; \varphi_j \rangle, 1 \leq j \leq p \} \xrightarrow{\langle T_i; T_i \rangle}{\bigl(1 \leq i \leq n\bigr)} \langle 1; \bigvee f'_j \varphi_j \rangle, \quad (16)
\]

\[
\{\langle f_j; \varphi_j \rangle, 1 \leq j \leq p \} \xrightarrow{\langle T_i; T_i \rangle}{\bigl(1 \leq i \leq n\bigr)} \langle 0; \bigwedge (f''_j \vee \varphi_j) \rangle. \quad (17)
\]

The same law \( T_i \) can thus be used for proving that a sum of functions is valid and that a product of functions is inconsistent. Let us denote \( \perp_i \) the dual (in the lattice sense) of \( T_i \), i.e.

\[
\perp_i(f_k, f_i) = (x_i \vee f_k) (\neg x_i \vee f_i).
\]

Since \( T_i(f_k, f_i) = \perp_i(f_k, f_i) \), the law \( T_i \) is self-dual; these laws can be rewritten as follows

\[
T_i(f_k, f_i) = \neg x_i f_k \vee x_i f_i \wedge f_k (x_i \Rightarrow 0) f_i (x_i \Rightarrow 1).
\]

\[
\perp_i(f_k, f_i) = (x_i \vee f_k) (\neg x_i \vee f_i) (f_i (x_i \Rightarrow 0) \vee f (x_i \Rightarrow 1)).
\]

Let us define reduced (or simplified) laws as follows

\[
T'_i(f_k, f_i) = f_k (x_i \Rightarrow 0) \wedge f_i (x_i \Rightarrow 1), \quad (23)
\]

\[
\perp'_i(f_k, f_i) = f_k (x_i \Rightarrow 0) \vee f_i (x_i \Rightarrow 1). \quad (24)
\]

We prove in appendix C that the reduced laws (23) and (24) can be used instead of (19) and (18) in constructive proofs for proving the validity of sums.
Theorem proving techniques and P-functions

of functions and the inconsistency of products of functions. This provides us
with a constructive proof of the following transformations

\[
\langle f_j; \varphi \rangle, 1 \leq j \leq p \leftarrow (1; \lor f_j \varphi), \quad (25)
\]

\[
\langle f_j; \varphi \rangle, 1 \leq j \leq p \rightarrow (1; \land (f_j'' \lor \varphi)). \quad (26)
\]

The laws \( T_i' \) and \( \bot_i' \) are dual laws. These laws are easier to use than the law \( T_i \),
in the sense that their automatic implementation (by means e.g. of computer programs) is generally straightforward. In non-automatic theorem proving the law \( T_i \) leads to proofs containing a smaller number of deductions than the laws \( T_i' \) and \( \bot_i' \) provided rewriting rules of the form: \( x \lor \neg x = 1 \) and \( x \land \neg x = 0 \)
are introduced in the course of the proof. This derives from the following inequalities

\[
T_i' < T_i = \bot_i < \bot_i'.
\]

Let us now show that the law \( \bot_i' \) is nothing but Robinson's resolution principle when the functions \( f_k, f_l \) are clauses, i.e. sums of literals.

The resolution principle for the propositional logic can be stated as follows \( 16,33 \)

For any two clauses (or sums of literals) \( C_1 \) and \( C_2 \), if there is a literal \( L_1 \) in \( C_1 \)
that is complementary to a literal \( L_2 \) in \( C_2 \), then delete \( L_1 \) and \( L_2 \) from \( C_1 \) and
\( C_2 \) respectively, and construct the disjunction of the remaining clauses.

**Proof of** \( f_1 \land f_2 = 0 \)

**Proof of** \( f_1 \land f_2 = 1 \)

\[
\tau_1(f_1, f_2) \quad \tau_2(f_1, f_2) \quad \tau_3(f_2, f_1) \quad \tau_4(f_2, f_1)
\]

\[
\tau_1 = \bot_1 \quad \text{or} \quad \bot_1'
\]

\[
t_i = T_i \quad \text{or} \quad T_i'
\]

Fig. 9. Deduction schemes for the example 5.
\[
\begin{array}{|c|c|c|}
\hline
f_1 &= x_1x_3 \lor x_2 \neg x_3 \\
f_2 &= \neg x_1x_3 \lor \neg x_2 \neg x_3 \\
\hline
f_1 &= (x_1 \lor \neg x_3) (x_2 \lor x_3) \\
f_2 &= (\neg x_1 \lor \neg x_3) (\neg x_2 \lor x_3) \\
\hline
f_1 &= (x_1 \lor x_2 \neg x_3) (x_2 \lor x_1x_3) \\
f_2 &= (\neg x_1 \lor x_2 \neg x_3) (\neg x_2 \lor \neg x_1x_3) \\
\hline
\end{array}
\]

Fig. 10. The use of the deduction laws in the example 5.
Theorem proving techniques and P-functions

The resolution operation (denoted \( R \)) coincides with the reduced law \( \perp' \) when its operands are clauses; let \( C_1 \) and \( C_2 \) be denoted as follows

\[
C_1 = L \lor C'_1, \quad C_2 = \neg L \lor C'_2.
\]

We have

\[
\perp'_R(C_1, C_2) = C'_1 \lor C'_2 = R(C_1, C_2).
\]

We illustrate the use of the various laws by means of example 5.

Example 5

\[
f_1 = x_1x_3 \lor x_2 \neg x_3
\]

\[
f_2 = \neg x_1x_3 \lor \neg x_2 \neg x_3.
\]

Figs 9 and 10 illustrate the use of the laws \( \perp'_i \) or \( \perp'_f \) for proving that \( f_1 \land f_2 = 0 \) and of the laws \( \top_i \) or \( \top'_f \) for proving that \( f_1 \lor f_2 = 1 \). Fig. 9 shows the two derivation schemes while fig. 10 illustrates the computations for three different types of expression of \( f_1, f_2 \): a sum of products, a product of sums and a general, well formed expression.

4.2. First-order logic

In the propositional logic, the basic elements are atoms; these atoms were denoted by lower-case letters \( x_1, \ldots, x_n \) in the preceding sections. Through atoms we build up well formed expressions denoted \( f_1, \ldots, f_p \). The first-order logic has three more logical notions called terms, predicates and quantifiers. Much of the everyday and mathematical language can be symbolized by the first-order logic (see e.g. refs 27, 28 and 33). In propositional logic theorem proving we want to verify that a formula \( \phi \) is (or is not) a logical consequence of the axioms \( \phi_1, \ldots, \phi_{p-1} \). In the first-order logic, any predicate symbol \( x_1, \ldots, x_n \) takes a specified number of arguments; in theorem proving we want to verify that a formula \( \phi \) is a logical consequence of the axioms \( \phi_1, \ldots, \phi_{p-1} \) and we ask for which value of the arguments \( ((\phi_1 \land \ldots \land \phi_{p-1}) \rightarrow \phi_p) \) holds; the value of the arguments are obtained through substitution and unification.

A substitution is a finite set of the form \( \{ v_1 \mapsto t_1, \ldots, v_n \mapsto t_n \} \) where every \( v_i \) is a variable, every \( t_i \) is a term different from \( v_i \) and no two elements in the set have the same variable before the substitution symbol.

Let \( \theta = \{ v_1 \mapsto t_1, \ldots, v_n \mapsto t_n \} \) be a substitution, and \( f \) be a formula. Then \( f\theta \) is an expression obtained from \( f \) by replacing simultaneously each occurrence of the variable \( v_i \), \( 1 \leq i \leq n \), in \( f \) by the term \( t_i \); \( f\theta \) is called an instance of \( f \).

A substitution is called a unifier for a set \( \{ \phi_1, \ldots, \phi_k \} \) if and only if \( \phi_1\theta = \phi_2\theta = \ldots = \phi_k\theta \); the set \( \{ \phi_1, \ldots, \phi_k \} \) is said to be unifiable if there is a unifier for it.
A unifier \( \theta \) for a set \( \{f_1, \ldots, f_k\} \) is a most general unifier if and only if for each unifier \( \gamma \) there is a substitution \( \lambda \) such that \( \gamma = \theta \lambda \). It can then be verified that the theorem of sec. 4.1 as well as the transformations (16, 17, 20, 21, 25 and 26) hold in first-order logic provided that the first-order extensions (6–9) of the laws (1–4) are used, respectively. A detailed proof of the completeness of the law \( L_9 \) for proving the inconsistency of a set of logic expressions can be found in Murray. Similar arguments as those developed by Murray can be used for proving the completeness of the other laws (a simple proof of the completeness can also be obtained from the inequalities (27) and Murray's statement).

Example 6 (ref. 33, p. 9)

The axioms and the conclusion are represented by the following formulas

\[
\begin{align*}
A_1: & \quad (\forall x)(E(x) \neg V(x) \rightarrow (\exists y)(S(x, y) C(y))). \\
A_2: & \quad (\exists x)(P(x) E(x) (\forall y)(S(x, y) \rightarrow P(y))), \\
A_3: & \quad (\forall x)(P(x) \rightarrow \neg V(x)), \\
C: & \quad (\exists x)(P(x) C(x)).
\end{align*}
\]

Transforming axioms and the negation of the conclusion into Skolem standard forms, we obtain

\[
\begin{align*}
A_1: & \quad \neg E(x) \lor V(x) \lor S(x, f(x)) C(f(x)), \\
A_2: & \quad P(a) E(a) (\neg S(a, y) \lor P(y)), \\
A_3: & \quad \neg P(x) \lor \neg V(x), \\
\neg C: & \quad \neg P(x) \lor \neg C(x).
\end{align*}
\]

A proof that \( C \) logically follows from the axioms is given in fig. 11.

When the axioms and the conclusion are written as a sum of products (or implicants) or as a product of sums (or clauses or implicates), deductive laws easier to use than the reduced laws \( T^i \) and \( L^i \) can be defined. Consider two formulas \( f_k(x) \) and \( f_i(x) \); let us write each of these formulas as a sum of products and as a product of sums and let us explicitly consider the variable \( x_i \) of \( x \):

\[
f_j = \neg x_i f_j^0 \lor x_i f_j^1 \lor f_j^2 = (x_i \lor F_j^0) (\neg x_i \lor F_j^1) F_j^2, \quad j = k, l,
\]

where \( f_j^2 \) is the sum of the products independent of \( x_i \), \( \neg x_i f_j^0 \) is the sum of the products dependent on \( \neg x_i \), and \( x_i f_j^1 \) is the sum of the products dependent on \( x_i \) (in \( f_j \)); \( F_j^0 \), \( (x_i \lor F_j^0) \) and \( (\neg x_i \lor F_j^2) \) are the dual concepts.

If the substitution \( \theta \) is a most general unifier for the instances of the predicate \( x_i \in x \) appearing in \( f_k \) and \( f_i \), we define (semantic) deduction laws as
Fig. 11. The deduction steps of example 6.

\[ A_1 = \neg E(x) \lor V(x) \lor S(x, f(x)) \land C(f(x)) \]

\[ A_2 = P(a) \land E(a) \land (\neg S(a, y) \lor P(y)) \]

\[ A_3 = \neg P(x) \lor \neg V(x) \]

\[ \neg C = \neg P(x) \lor \neg C(x) \]

\[ B_1 = \bot_{E(x=a)}(A_2, A_1) = V(a) \lor S(a, f(a)) \land C(f(a)) \]

\[ B_2 = \bot_{P(x=a)}(A_2, A_3) = \neg V(a) \]

\[ B_3 = \bot_{V(B_1, B_2)} = S(a, f(a)) \land C(f(a)) \]

\[ B_4 = \bot_{S(y=f(a))}(B_3, A_2) = P(a) \land E(a) \land P(f(a)) \]

\[ B_5 = \bot_{P(x=f(a))}(B_4, \neg C) = \neg C(f(a)) \]

\[ B = \bot_{C(B_3, B_5)} = 0 \]
follows
\[ \top_{\theta}(f_k, f_i) = f_k^0 \lor f_i^1, \quad (27) \]
\[ \bot_{\theta}(f_k, f_i) = f_k^0 \lor f_i^1. \quad (28) \]

Semantic laws have been defined by Sanchez and Thayse\(^{(37)}\). It can be verified that \( \bot_{\theta} \) (as well as \( \top_{\theta} \)) reduces to Robinson's resolution principle when \( f_j \) and \( f_k \) are clauses; dually, \( \top_{\theta} \) (as well as \( \top_{\theta} \)) reduces to Tison's\(^{(38)}\) consensus operation when \( f_j \) and \( f_k \) are products of literals.

The completeness of the laws \( \top_{\theta} \) and \( \bot_{\theta} \) can be derived by using similar arguments as those developed by Chang and Lee\(^{(33)}\) for proving the completeness of the resolution principle.

Observe that the following inequalities hold between the deductive laws
\[ \top_{\theta} \text{ (or consensus)} < \top_{\theta} < \top_{\theta} = \bot_{\theta} < \bot_{\theta} < \bot_{\theta} \text{ (or resolution)} \]
\[ \text{backward deductive laws} \quad \text{forward deductive laws} \quad (29) \]

Among these laws, \( \top_{\theta} \) and \( \bot_{\theta} \) are the most general laws; this means that in non-automatic theorem proving the use of \( \top_{\theta} \) provides us with proofs with a smaller number of deduction steps than the use of \( \top_{\theta} \) or \( \top_{\theta} \) (in a backward deduction scheme) or of \( \bot_{\theta} \) or \( \bot_{\theta} \) (in a forward deduction scheme). The elementary deduction step is, however, more complicated when using the law \( \top_{\theta} \) instead of the other laws\(^{1-4}\).

4.3. Transformations acting on P-functions and algorithm implementation

We indicate in this section how transformations acting on P-functions provide us with algebraic models for algorithm implementation schemes. First consider the algorithmic implementation model (example 1). We have shown (refs 1-4, example 1) that implementations of algorithms in an algorithmic language (program schemata model) are grounded on the following transformation

\[ \text{initial system of P-functions} \quad \longrightarrow \quad \text{final P-function} \]
\[ \{\langle f_j; \sigma_j \rangle, 1 \leq j \leq p \} \quad \longrightarrow \quad \langle 1; \lor f_j \sigma_j \rangle \quad (30) \]

In this model, potential parallelism in algorithms is reflected by the non-disjoint character of the functions \( f_j(3k, l: f_k \land f_i \neq 0) \); the transformation (30) then corresponds to the synthesis of a generalized fork instruction (made-up of \textit{if then else decision} instructions and of \textit{fork} instructions). This instruction must be associated to a generalized join instruction (made-up of \textit{then if} instructions and of \textit{join} instructions) whose implementation is obtained from
Theorem proving techniques and P-functions

the following transformation

\[
\text{initial system of P-functions} \quad \longrightarrow \quad \text{final P-function}
\]

\[
\{ \langle \neg f_j; \sigma_i \rangle, 1 \leq j \leq p \} \quad \longrightarrow \quad \langle 0; (\neg f_j \vee \sigma_i) \rangle
\]

\[(1 \leq i \leq n) \quad (31)\]

Consider now logic programming based on resolution (example 3). The constructive law in logic programming is the substitution \( \theta \); the instructions of a program are Skolemized first-order logic functions \( \{ f_j \} \). We define the set of P-functions associated with a program as \( \{ \langle f_j; 0 \rangle \} \). The program is activated by a call instruction represented by a first-order logic function \( f_{p+1} \) containing existential quantified variables \( z \); we associate with this instruction the P-function \( \langle f_{p+1}; z \rangle \).

In the logic programming language Prolog the formulas \( f_1, \ldots, f_{p-1}, f_p \) are Horn clauses (i.e. sums with at most one positive literal); the computations performed during the execution of a Prolog program can be modelled by the following transformation

\[
\text{initial system of P-functions} \quad \longrightarrow \quad \text{final P-function}
\]

\[
\{ \langle f_j; 0 \rangle, \langle f_{p+1}; z \rangle \} \quad \longrightarrow \quad \langle 0; \text{result of computation} \rangle
\]

\[(1 \leq i \leq n) \quad (32)\]

Malachi, Manna and Waldinger\(^\text{25}\) have proposed a logic programming language (called Tablog) whose instructions are Skolemized first-order logic formulas; its computation can be modelized as follows

\[
\text{initial system of P-functions} \quad \longrightarrow \quad \text{final P-function}
\]

\[
\{ \langle f_j; 0 \rangle, \langle f_{p+1}; z \rangle \} \quad \longrightarrow \quad \langle 0; \text{result of computation} \rangle
\]

\[(1 \leq i \leq n) \quad (33)\]

Consider finally the deductive approach\(^\text{10}\) for recursive program synthesis. Let us denote by \( \{ f_j \}, f_R \) the axioms and the recursive axiom (or their negation) and by \( f_{p+1} \) the conclusion (or its negation) of the theorem to be proved; the \( f_j, f_R, f_{p+1} \) are first-order Skolemized formulas and \( f_{p+1} \) contains existential quantified variables denoted \( z \). Both transformations (34) and (35) constitute models for the synthesis of recursive programs.

\[
\text{initial system of P-functions} \quad \longrightarrow \quad \text{final P-function}
\]

\[
\{ \langle f_j; 0 \rangle, \langle f_R; 0 \rangle, \langle f_{p+1}; z \rangle \} \quad \longrightarrow \quad \langle 1; \text{program} \rangle
\]

\[(1 \leq i \leq n) \quad (34)\]

\[
\{ \langle f_j; 0 \rangle, \langle f_R; 0 \rangle, \langle f_{p+1}; z \rangle \} \quad \longrightarrow \quad \langle 0; \text{program} \rangle
\]

\[(1 \leq i \leq n) \quad (35)\]
initial system of P-functions: \( \langle f_j(x); \phi_j \rangle, 1 \leq j < p \)  

\[ \langle \text{deductive laws } t_{i\theta} \text{ or } t_{i\theta} ; \text{constructive laws } t_{i\theta}^{*}, 1 \leq i \leq n \rangle \]

Fig. 12. Deductive and constructive laws for P-functions.

final P-function: \( \langle 1 \text{ or } 0; \text{algorithm description or program output} \rangle \)
Theorem proving techniques and P-functions

(Note that if the codomain of one of the two P-functions is 0, i.e.: \( \langle f_k; 0 \rangle \), \( \langle f_i; z \rangle \) the use of a pair of laws \( \langle t_i \text{ or } \tau_i; \theta \rangle \) in the transformations (34) and (35) instead of \( \langle t_i \text{ or } \tau_i; t_{00} \rangle \) will provide us with a simpler form of the final P-function, i.e. of the resulting program.)

The conclusions of this section are gathered in fig. 12. The laws for proving that a product of formulas is 0 or that a sum of formulas is 1 have been arranged in a lattice form; the construction of the lattice of deductive laws will be continued in the next section.

4.4. Theorem proving and prime implicant extraction

First consider propositional logic. A sum of Boolean functions \( f_1 \lor \ldots \lor f_p \) has 1 as prime implicant if and only if \( \lor f_j = 1 \); a product of Boolean functions \( f_1 \land \ldots \land f_p \) has 0 as prime implicate if and only if \( \land f_j = 0 \) (see e.g. ref. 39 or any book on switching theory or Boolean algebra). Prime implicant extraction algorithms can thus also be viewed as provers that a sum of functions is 1 while prime implicate extraction algorithms can also be viewed as provers that a product of functions is 0. A well-known procedure for finding prime implicants of a Boolean function is based on what Quine \(^{40}\) first called the consensus of implicants. The original method developed by Quine is generally referred to as the iterative consensus; it is fully detailed in most of the books dealing with switching theory. This method has been improved by Tison \(^{38}\) who suggested a more efficient algorithm which is called the generalized consensus. Thayse \(^{42}\) showed that the concepts of meet and join differences are a convenient algebraic support for both the iterative and the generalized consensus.

The meet difference \( p_i f \) of \( f \) with respect to \( x_i \) is the function

\[
p_i f = f(x_i = 0) \land f(x_i = 1).
\]

It is equal to the sum of the implicants of \( f \) which are independent of \( x_i \). The join difference \( q_i f \) of \( f \) with respect to \( x_i \) is the function

\[
q_i f = f(x_i = 0) \lor f(x_i = 1).
\]

It is equal to the product of the implicates of \( f \) which are independent of \( x_i \). Similar arguments as those developed in sec. 4.1 allow us to state that the meet and join differences

\[
p_i (f_k \lor f_i) = (f_k(x_i = 0) \lor f_i(x_i = 0)) (f_k(x_i = 1) \lor f_i(x_i = 1)),
\]

\[
q_i (f_k \land f_i) = f_k(x_i = 0) f_i(x_i = 0) \lor f_k(x_i = 1) f_i(x_i = 1),
\]

can be used as deductive laws for proving \( \lor f_j = 1 \) and that \( \land f_j = 0 \), respectively.
Example 5 (continuation)

\[ f_1 = x_1 x_3 \lor x_2 \neg x_3 \lor x_2 \neg x_3, f_2 = \neg x_1 x_3 \lor \neg x_2 \neg x_3 \]
\[ p_1(f_1 \lor f_2) = (x_2 \lor x_1 x_3 \lor \neg x_2 \neg x_3) (x_2 \lor x_1 x_3 \lor \neg x_2 \neg x_3) = g_1 \]
\[ p_2(g_1) = (x_3 \lor \neg x_3) = g_2 \]
\[ p_3(g_2) = 1. \]

This proves that \( f_1 \lor f_2 = 1 \).

\[ q_1(f_1 \land f_2) = (x_3 \lor x_2 \neg x_3) (\neg x_2 \lor x_3) (x_2 \lor x_3) (x_3 \lor \neg x_2 \lor x_3) = g'_1 \]
\[ q_2(g'_1) = x_3 \land \neg x_3 = g'_2 \]
\[ q_3(g'_2) = 0. \]

This proves that \( f_1 \land f_2 = 0 \).

The laws \( p_i(f_k \lor f_i) \) and \( q_i(f_k \land f_i) \) can thus be used in theorem proving procedures; however they cannot be used in algorithm synthesis since the constructive law \( \top \) requires that the respective parts of \( f_k \) and \( f_i \) should be kept separate in the deduction process. This is not the case when using the laws \( p_i \) and \( q_i \) (see example 5 above).

The laws \( p_i \) and \( q_i \) can be used in first-order logic theorem proving procedures. They can also be used in algorithm synthesis when the constructive law is the substitution \( \theta \); they cannot however be used when the constructive laws are \( \top \) (for the same reasons as those quoted for propositional logic).

The various laws for proving that a sum of functions is 1 or that a product of functions is 0 are gathered in the lattice of fig. 13. As for fig. 12 (or as for the inequalities (29)) the complexity of each of these laws and the number of elementary steps of the proof they require, is reflected by the distance in the lattice of these laws from \( f_k \lor f_i \) or from \( f_k \land f_i \).

The following relations holds between the meet and join differences and the binary deductive laws

\[ p_{\theta}(\top_{\theta}(f_k, f_i)) = \top_{\theta}(f_k, f_i). \]  \hspace{1cm} (40)
\[ q_{\theta}(\bot_{\theta}(f_k, f_i)) = \bot_{\theta}(f_k, f_i). \]  \hspace{1cm} (41)
Theorem proving techniques and P-functions

Fig. 13. The lattice of deductive laws.
These relations lead us to state the following theorem:

**Theorem 4**

a. If there exists an iterative application of operations \( t_{10}(f_k, f_i) \) which when applied to the set of functions \( f_1(x), \ldots, f_p(x) \), produces the sum \( \bigvee f_j(x) \), then there exists an iterative application of operations \( p_{10}(t_{10}(f_k, f_i)) \) which when applied to the same set of functions produces the constant 1 if and only if \( \bigvee f_j(x) = 1 \).

b. If there exists an iterative application of operations \( \tau_{10}(f_k, f_i) \) which when applied to the set of functions \( f_1(x), \ldots, f_p(x) \), produces the product \( \bigwedge f_j(x) \), then there exists an iterative application of operations \( q_{10}(\tau_{10}(f_k, f_i)) \) which when applied to the same set of functions produces the constant 0 if and only if \( \bigwedge f_j(x) = 0 \).

**Proof**

a. If an iterative application of \( t_{10}(f_k, f_i) \) produces \( \bigvee f_j \), then an iterative application of \( p_{10}(\bigvee f_j) \) produces 1 if and only if \( \bigvee f_j \) has a prime implicant equal to 1. Hence an iterative application of \( p_{10}(t_{10}(f_k, f_i)) \) produces 1 if and only if \( \bigvee f_j = 1 \).

b. If an iterative application of \( \tau_{10}(f_k, f_i) \) produces \( \bigwedge f_j \), then an iterative application of \( q_{10}(\bigwedge f_j) \) produces 0 if and only if \( \bigwedge f_j \) has a prime implicate equal to 0. Hence an iterative application of \( q_{10}(\tau_{10}(f_k, f_i)) \) produces 0 if and only if \( \bigwedge f_j = 0 \).

Consider the lattice of operations represented by fig. 13. It is formed by a central plane containing the formulas: \( f_k \lor f_i, f_k \land f_i, \bigvee_{10}(f_k, f_i) = \bigvee_{10}(f_k, f_i) \) and \( \bigwedge_{10}(f_i, f_k) = \bigwedge_{10}(f_i, f_k) \). The upper plane and the lower plane are obtained by performing meet and join derivations on the vertices of the central plane.

For the sake of simplicity let us denote by \( x \) one of the variables of \( x \). The operation \( T_x(f_k, f_i) \) (for the sake of conciseness the substitution \( \theta \) will be dropped in the following) will be renamed as follows:

\[
T_x(f_k, f_i) = \bigvee \neg_{10}(f_k, f_i) = \neg x f_k \lor x f_i. 
\]

Hence

\[
T_{11}(f_k, f_i) = 1 f_k \lor 1 f_i = f_k \lor f_i.
\]

The vertices \( T_{\neg x}(f_k, f_i) \) and \( T_{11}(f_k, f_i) \) of fig. 13 can now be considered as two extreme operations of a set defined as follows:

\[
T_{x_1 x_2}(f_k, f_i) = x_1 f_k \lor x_2 f_i; \quad \neg x \leq x_1 \leq 1, \ x \leq x_2 \leq 1. \tag{42}
\]

This set of operations is represented by the arc \( (T_{11}, T_{\neg x}) \) of fig. 14.
Theorem proving techniques and P-functions

Fig. 14. Extension of deductive laws.

Philips Journal of Research Vol. 41 No. 5 1986 499
Similar arguments as those used in appendix A allow us to state that an iterative application of these laws will produce the function $\bigvee f_j$ if and only if $\neg x \leq x_1$, $x \leq x_2$. We have also

$$T_{\neg xx}(f_k, f_i) = (x \lor f_k(x \Rightarrow 0)) (\neg x \lor f_i(x \Rightarrow 1)) = T_{\neg xx}(f_k(0), f_i(1)).$$

Hence

$$T_{00}(f_k(0), f_i(1)) = (0 \lor f_k(x \Rightarrow 0)) (0 \lor f_i(x \Rightarrow 1)) = T'_{xx}(f_k, f_i).$$

The vertices $T_{\neg xx}(f_k, f_i) = T_{\neg xx}(f_k(0), f_i(1))$ and $T'_{xx}(f_k, f_i) = T_{00}(f_k(0), f_i(1))$ of fig. 14 can it turn be considered as two extreme operations of a set defined as follows:

$$T_{x_1 x_2}(f_k(0), f_i(1)) = (x_1 \lor f_k(x \Rightarrow 0)) (x_2 \lor f_i(x \Rightarrow 1))$$

$$0 \leq x_1 \leq \neg x, \ 0 \leq x_2 \leq x.$$

This set of operations is represented by the arc $(T_{\neg xx}, T_{00})$ of fig. 14. These laws can be used for proving that a sum of functions is 1.

In summary the plane $(T_{\neg xx}, T_{11}, T_{00})$ of fig. 14 characterizes the domain of the laws (42) and (43) which can be used for obtaining the sum $\bigvee f_j$ from \{f_1, \ldots , f_\rho\} and the constant 1 if $\bigvee f_j \equiv 1$.

Dual laws can now be defined as follows

$$\bot_x(f_k, f_i) = \bot_{\neg xx}(f_k, f_i) = (x \lor f_k) (\neg x \lor f_i).$$

Hence

$$\bot_{00}(f_k, f_i) = (0 \lor f_k) (0 \lor f_i) = f_k \land f_i,$$

and

$$\bot_{x_1 x_2}(f_k, f_i) = (x_1 \lor f_k) (x_2 \lor f_i), \ 0 \leq x_1 \leq x, \ 0 \leq x_2 \leq \neg x$$

$$\bot_{\neg xx}(f_k, f_i) = \neg x f_k(x \Rightarrow 0) \lor x f_i(x \Rightarrow 1) = \bot_{\neg xx}(f_k(0), f_i(1)). \quad (44)$$

Hence

$$\bot_{11}(f_k(0), f_i(1)) = 1 f_k(x \Rightarrow 0) \lor 1 f_i(x \Rightarrow 1) = \bot'_{xx}(f_k, f_i),$$

and

$$\bot_{x_1 x_2}(f_k(0), f_i(1)) = x_1 f_k(x \Rightarrow 0) \lor x_2 f_i(x \Rightarrow 1)$$

$$\neg x \leq x_1 \leq 1, \ x \leq x_2 \leq 1. \quad (45)$$

The plane $(\bot_{11}, \bot_{\neg xx}, \bot_{00}, q)$ of fig. 13 characterizes the domain of the laws (44) and (45) which can be used for obtaining the product $\bigwedge f_j$ from \{f_1, \ldots , f_\rho\} and the constant 0 if $\bigwedge f_j \equiv 0$. Note finally that the central plane of figs 13 and 14 can be considered as the domain of the constructive laws (second law of the P-function).
Some of the above laws have recently been considered by Manna and Waldinger with the purpose to accelerate derivations.

Appendix A
Proof of theorem 3
a. First consider the case \( p = 2 \)
Any iterative application of operations \( \langle T_i; T_i \rangle \) applied to \( \langle f_1; \varphi_1 \rangle, \langle f_2; \varphi_2 \rangle \) produces P-functions of the form

\[
\langle f_1u_1 \vee f_2u_2; \varphi_1u_1 \vee \varphi_2u_2 \rangle \text{ with } u_1u_2 = 0, u_1 \vee u_2 = 1. \tag{A1}
\]

It is always possible to choose arbitrarily the function \( U_1 \): this reduces to apply iteratively the decomposition

\[
U_1(x) = \neg xU_1(x) \Rightarrow 0 \vee xU_1(x) \Rightarrow 1
= T_i(u_1(x) \Rightarrow 0), u_1(x) = 1).
\]

This decomposition produces the sequence of operations leading to the desired function \( U_1 \). Let us choose \( U_1 = f_i \); this choice determines also the iterative application of \( \langle T_i; T_i \rangle \). The resulting P-function (A1) becomes

\[
\langle f_1 \vee f_2u_2; f_i \varphi_1 \vee u_2 \varphi_2 \rangle.
\]

Moreover:

\[
f_1 \vee f_2u_2 = (f_1 \vee f_2)(u_1 \vee u_2) = f_1 \vee f_2 = 1 \text{ if and only if } f_1 \vee f_2 = 1.
\]

\[
(f_1 \vee f_2 = 1) \iff (\neg f_1 \wedge \neg f_2 = 0) \iff \neg f_1 \leq f_2
\]

\[
(u_1u_2 = 0) \iff (f_1u_2 = 0) \iff u_2 \leq \neg f_1.
\]

Hence

\[
u_2 \leq f_2.
\]

The resulting P-function is thus

\[
\langle f_1u_1 \vee f_2u_2; \varphi_1u_1 \vee \varphi_2u_2 \rangle = \langle f_1 \vee f_2; f_2^i \varphi_1 \vee f_2^i \varphi_2 \rangle
= \langle 1; f_1^i \varphi_1 \vee f_2^i \varphi_2 \rangle, f_1^i \leq f_1, f_2^i \leq f_2.
\]

Assume moreover that \( f_1 \wedge f_2 = 0 \);

\[
((f_1 \vee f_2 = 1) \wedge (f_1 \wedge f_2 = 0)) \iff (f_1 = \neg f_2)
\]

\[
((u_1 \vee u_2 = 1) \wedge (u_1 \wedge u_2 \equiv 0)) \iff (u_1 = \neg u_2).
\]

Hence

\[
u_2 = \neg u_1 = \neg f_1 = f_2,
\]

and the resulting P-function is

\[
\langle f_1u_1 \vee f_2u_2; u_1 \varphi_1 \vee u_2 \varphi_2 \rangle = \langle 1; f_1^i \varphi_1 \vee f_2^i \varphi_2 \rangle.
\]
The proof can be completed by performing an induction on \( p \). The case \( p = 3 \) illustrates the induction mechanism. Reasoning as above we first take \( u_1 = f_1 \) and we obtain the P-function

\[
\langle f_1 u_1 \vee f_2 u_2; u_1 \varphi_1 \vee u_2 \varphi_2 \rangle = \langle f_1 \vee f_2; f_1 \varphi_1 \vee u_2 \varphi_2 \rangle. \tag{A2}
\]

Any iterative application of operations \( \langle T_i; T_i \rangle \) applied to \( \langle f_j; \varphi_j \rangle \) and (A2) produces P-functions of the form

\[
\langle (f_1 \vee f_2) u'_1 \vee f_3 u'_2; (f_1 \varphi_1 \vee u_2 \varphi_2) u'_1 \vee u_2 \varphi_3 \rangle, \quad \text{with } u'_1 u_2 = 0, \ u'_1 \vee u'_2 = 1. \tag{A3}
\]

Again it is possible to choose \( u'_1 = f_1 \vee f_2 \) and this choice determines the iterative application of \( \langle T_i; T_i \rangle \). The resulting P-function (A3) becomes

\[
\langle f_1 \vee f_2 \vee f_3 u'_2; f_1 \varphi_1 \vee u_2 u_1 \varphi_2 \vee u_2 \varphi_3 \rangle.
\]

Moreover

\[
f_1 \vee f_2 \vee f_3 u'_2 = (f_1 \vee f_2 \vee f_3) (u'_1 \vee u'_2) = f_1 \vee f_2 \vee f_3 = 1 \text{ iff } f_1 \vee f_2 \vee f_3 = 1.
\]

Let us prove that if \( \forall f_j < 1 \), it is impossible to obtain a P-function with a 1 in its domain (only if part of the theorem).

Any iterative application of \( T_i \) produces domain-functions of the form \( f_1 u_1 \vee f_2 u_2 \); we have

\[
f_1 u_1 \vee f_2 u_2 = (f_1 u_1 \vee f_2 u_2) (f_1 \vee f_2),
\]

and this domain is \( < 1 \) if \( f_1 \vee f_2 < 1 \).

b. A dual statement holds for proving part b of the theorem.

We have to choose for the domain of the P-function (A1) a function \( u_1 = \neg f_1 \); this choice determines the iterative application of \( \langle T_i; T_i \rangle \) which will produce a 0 in the domain of the resulting P-function. We have successively

\[
\langle f_1 u_1 \vee f_2 u_2; \varphi_1 u_1 \vee \varphi_2 u_2 \rangle = \langle f_2 u_2; \varphi_1 \neg f_1 \vee \varphi_2 u_2 \rangle,
\]

\[
= \langle f_2 u_2; (\varphi_1 \vee u_2) (\varphi_2 \vee \neg f_1) \rangle;
\]

\[
(u_1 \vee u_2 = 1) \iff (\neg f_1 \vee u_2 = 1) \iff (f_1 \neg u_2 = 0) \iff u_2 \geq f_1;
\]

\[
(f_1 f_2 = 0) \iff \neg f_1 \geq f_2;
\]

\[
\langle f_1 u_1 \vee f_2 u_2; \varphi_1 u_1 \vee f_2 u_2 \rangle = \langle 0; (f_1' \vee \varphi_1) (f_2'' \vee \varphi_2) \rangle, f_1' \geq f_1, f_2'' \geq f_2.
\]

Appendix B

Proof of transformations (20) and (21)

A direct proof of the transformations (20) and (21) can be obtained by means of arguments isomorphic to those used in the course of the proof of theorem 3. The proof however requires some different algebraic tools that we briefly describe.
Let \( u \) be a set (formal sum or disjunction) of subcubes (or products of literals) of the \( n \)-dimension Boolean algebra. Given a Boolean function \( f \), we denote by \( f* u \) (image of \( f \) by \( u \)) the Boolean function defined as follows:

\[
(f* u)(x \Rightarrow e) = 1, \text{ if } f(x \Rightarrow e) = 1 \text{ and } e \text{ belongs to a cube of } u, \\
(f* u)(x \Rightarrow e) = 0, \text{ if } f(x \Rightarrow e) = 0 \text{ or if } e \text{ does not belong to any cube of } u.
\]

The function \( f* u \) is thus the disjunction of the evaluations of \( f \) in each of the cubes of \( u \), i.e.

\[
f* u = \bigvee f(\text{cubes of } u).
\]

In particular, we have the elementary relations

\[
f* x_i = f(x_i \Rightarrow 1) \text{ and } f* \neg x_i = f(x_i \Rightarrow 0),
\]

and thus

\[
\top_i(f_k, f_i) = \neg x_i f_k(x_i \Rightarrow 0) \lor x_i f_i(x_i \Rightarrow 1) = \neg x_i (f_k* \neg x_i) \lor x_i (f_i* x_i).
\]  

(A4)

Let us show how this formalism allows us to give a constructive proof of the transformation (21).

Any iterative application of the operation (A4) to \( f_1, f_2 \) produces domain functions of the form

\[
u_1 \land (f_1* u_1) \lor u_2 \land (f_2* u_2).
\]

Let us choose \( u_1 = \neg f_1 \); we have

\[
(\neg u_1 \land u_2 = 0) \iff (f_1 \land u_2 = 0) \iff u_2 \geq f_1 \\
(f_1 f_2 = 0) \iff f_1 \geq \neg f_2,
\]

and thus: \( u_2 \geq f_2 \); from \((f_1* \neg f_1) = 0 \) and \((f_2* \neg f_2) = 0 \), we finally deduce

\[
u_1 \land (f_1* u_1) \lor u_2 \land (f_2* u_2) = (u_1 \land 0) \lor (u_2 \land 0) = 0 \lor 0 = 0.
\]

Dual types of arguments and formalisms allow us to state the transformation (20).

Appendix C

Proof of transformation (22) and (23)

Again a direct proof of the transformations (22) and (23) can be obtained by means of arguments isomorphic to those used in the course of the proof of theorem 3. Using the algebraic tools proposed in appendix B, we have

\[
\top_i(f_k, f_i) = f_k(x_i \Rightarrow 0) \lor x_i f_i(x_i \Rightarrow 1) = f_k* \neg x_i \lor f_i* x_i.
\]  

(A5)
Any iterative application of the operation (A5) to \( f_1, f_2 \) produces domain functions of the form
\[
f_1 \ast u_1 \vee f_2 \ast u_2.
\]
Let us choose \( u_1 = \neg f_1 \) and thus \( u_2 \geq \neg f_2 \); from \((f_1 \ast \neg f_1) = 0 \) and \((f_2 \ast \neg f_2) = 0 \) we deduce
\[
f_1 \ast u_1 \vee f_2 \ast u_2 = 0 \vee 0.
\]

Acknowledgement

The constructive proof of theorem 3 has been suggested by our colleague Ph. Delsart. We are much indebted to him.

REFERENCES

1) A. Thayse, Philips J. of Res. 35, 190 (1980).
7) V. Glushkov, Kibernetika 1, 1 (1965).
10) Z. Manna and R. Waldinger, ACM Transactions on programming languages and systems 2, 90 (1980).
15) B. Moret, ACM computing surveys 14, 593 (1982).
18) N. Murray, Artificial intelligence 18, 67 (1982).
36) J. Herbrand, Travaux de la société des sciences et des lettres de Varsovie 33, 128 (1930).