Theoretical Computer Science 9 (1979) 329–345


# MECHANIZING STRUCTURAL INDUCTION PART I: FORMAL SYSTEM

Raymond AUBIN

*Department of Computer Science, Concordia University, Montréal, Québec, H3G 1M8, Canada**

Communicated by M. Nivat
Received March 1977
Revised March 1978

**Abstract.** Inductive methods are basic to program proving and this paper presents the formal part of a theorem-proving system for automating mildly complex proofs by structural induction. Its features are motivated by the pragmatics of proof finding. Its syntax includes a typed language and an induction rule using a lexicographic ordering based on a substructure ordering. The domain of interpretation is a many-sorted word algebra generated by the empty set. The carriers of the algebra are ordered and functions are defined by $k$-recursion over them. Finally, the soundness and the weak completeness of the system are proved. The main quality of the language is its system of types and its correspondingly general induction rule.

## 1. Introduction

In general, proving properties of programs requires an inductive argument of one sort or other. In this context, the study of methods for mechanizing proofs by induction is of considerable interest. Any theorem-proving system breaks at some point into (1) a formal system and (2) proof finding methods, though both aspects largely influence each other. This paper expounds the formal part of a theorem-prover which automates mildly complex proofs by structural induction. It can be thought of as a generalization of number theory; but its features are primarily motivated by the design of tolerably efficient automatic proof-finding methods described in a second part [2]. A thorough exposition of the whole system can be found in Aubin [1].

Induction on the structure of data is used in this system for proving facts about recursive functions. In general terms, suppose that set $S$ is ordered and every non-empty subset of $S$ has a minimal element; then, to prove the property $P(f(c))$, for all $c$ in $S$, it suffices to show that for all $a$ in $S$, $P(f(b))$, for all $b$ less than $a$ in $S$, implies $P(f(a))$. Structural induction is often understood in practice as using a substructure ordering (see [6, 11]). Theorem-proving programs using such an inductive method were written by Brotz [5] for number theory and by Boyer and Moore [4] for a theory of lists (see also [14]).

* Present address: Recherches Bell-Northern, Ottawa, Ontario, K1Y 4H7, Canada.

The present formal system is an improvement over previous work by

(1) its typed language and

(2) its general induction rule using a lexicographic ordering based on the substructure ordering induced by the type definitions.

Besides, the formalization is pushed to a greater detail.

After an exposition of the syntax and semantics of the system, I prove its soundness and weak completeness. The general presentation is inspired from Robbin [17] and Milner, Morris and Newey [13]. A final discussion justifies the choice of some of the features of this system.

## 2. Syntax

Logicians are very concrete about the formal syntax of a language. They give a number of primitive symbols and then define the set of admissible strings of concrete symbols. I will in general, be more abstract without confusion. On the other hand, I will make use of abbreviations whereby metalinguistic names are used in place of syntactic constructs.

### *2.1. Syntactic constructs*

#### *2.1.1. Type constants*

We have an infinite list of primitive constructs $\iota_1, \iota_2, \ldots$ called *type constants.*

Type constants will be given names. We use the metavariables $\sigma, \tau$, maybe with indices, to vary over type constants. An expression like $\sigma^*$ stands for the list $\sigma_1, \ldots, \sigma_n$ $(0 \leqslant n)$ of type constants. The length of the list is denoted by length $(\sigma^*)$ but can usually be inferred from the context.

#### *2.1.2. Variables*

We have an infinite list of primitive constructs $v_1, v_2, \ldots$ called *variable tokens.* A *variable of type* $\tau$ is defined thus: if $v_i$ is a variable token and $\tau$, a type constant, then $v_i : \tau$ is a variable of type $\tau$.

Variables will normally be abbreviated by names. We use $x, y, z$ to vary over variables and $x^*, y^*, z^*$ will denote lists of distinct variables.

#### *2.1.3. Constructor constants*

We have an infinite list of primitive constructs $c_1, c_2, \ldots$ called *constructor tokens.* If $c_i$ is a constructor token and $\sigma^*, \tau$ are type constants, then $c_i : \sigma^* \to \tau$ is a *constructor constant of type* $\sigma^* \to \tau$.

The arity of a constructor constant $c_i : \sigma^* \to \tau$ is equal to length $(\sigma^*)$. We use names to abbreviate constructor constants and we use the metavariable $c$ to range over them.

### *2.1.4. Defined function constants*

This section is analogous to the previous one. We have an infinite list of primitive constructs $f_1, f_2, \ldots$ called *defined function tokens*; if $f_i$ is a defined function token and $\sigma^*, \tau$ are type constants, then $f_i : \sigma^* \to \tau$ is a *defined function constant of type* $\sigma^* \to \tau$.

Constructor and defined function constants form the class of *function constants.* The metavariables *f, g, h* are used to vary over defined function constants and over function constants; the context will make clear which is meant.

### *2.1.5. Terms*

The class of *terms of type* $\tau$ is inductively defined thus:

(1) If $x$ is a variable of type $\tau$, then $x$ is a term of type $\tau$.

(2) If $c$ is a constructor constant of type $\sigma^* \to \tau$ and $t^*$ are terms of types $\sigma^*$ respectively, then $c(t^*)$ is a term of type $\tau$.

(3) If $f$ is a defined function constant of type $\sigma^* \to \tau$ and $t^*$ are terms of types $\sigma^*$ respectively, then $f(t^*)$ is a term of type $\tau$.

(4) A construct is a term only as required by (1), (2) and (3).

A term which is not a variable will be called a *function application* (or sometimes more simply, an application). In the following text, I will freely infix function constants; the computer program makes use of a different concrete syntax. As already seen above, the metavariables *s, t, u* and *w* are used to vary over terms. An expression like $f(t^*)$ denotes the function application $f(t_1, \ldots, t_n)$ ($n = \text{arity}(f)$). An expression like $t[s/x]$ denotes the term resulting from replacing all occurrences of $x$ by $s$ in $t$; $t[s^*/x^*]$ denotes $t[s_1/x_1] \cdots [s_n/x_n]$.

## *2.2. Introduction of syntactic constructs*

Potentially, we have a countable number of type constants, variables, constructor constants, defined function constants. However, each of the constructs used in the system has to be previously introduced (or defined, or distinguished, or declared, these are all synonymous as far as I am concerned.) This introduction is done in a hierarchical manner and serves also the purpose of giving names to the constructs in question. From now on, I will not distinguish between a syntactic construct and its name. This section is really part of the metalanguage.

A *type definition* is generated by the following (abstract) BNF grammar:

⟨type definition⟩ :: = ⟨head⟩⟨body⟩

⟨head⟩ :: = ⟨type constant⟩

⟨body⟩ :: = {⟨component⟩}

⟨component⟩ :: = ⟨constructor constant⟩ {⟨type constant⟩}

where the defined type constant is in the head of the definition.

Such a definition is admissible if and only if its syntactic constructs other than the type and constructor constants being defined have been previously introduced. In the text, I will use the concrete representation, e.g.

[true:|false:]→bool

[zero:|succ:nat]→nat

[nil:|cons:nat, list]→list

[atom:nat|consx:sexpr, sexpr]→sexpr

[nulltree:|tip:nat|node:tree, nat, tree]→tree.

A type constant is said to be *reflexive* if it occurs in the body of its definition. (This term borrowed from Milner, Morris and Newey [13] is preferred to inductive or recursive.) I will come back to why mutually reflexive type constants are not permitted under the present syntax. We consider that type constants are named by the words, e.g., bool, nat, etc., as well as by the whole type definitions.

The type of a constructor constant is immediate from the type definition, e.g. → bool for true; nat, list → list for cons; etc. In practice, I will often omit the parentheses in terms like true ( ) when it can be done without confusion. By analogy with type constants, we say that a constructor constant $c_i: \sigma^* \to \tau$ is *reflective* if $\tau$ occurs in $\sigma^*$. An argument of $c_i: \sigma^* \to \tau$ occurring in the position of $\tau$ in $\sigma^*$ is called a *reflexion argument*. An *immediate predecessor* of a list $c_1(x_1^*), \ldots, c_n(x_n^*)$ is a list

$$c_1(x_1^*), \ldots, c_{i-1}(x_{i-1}^*), x_{i,j}, s_{i+1}, \ldots, s_n,$$

such that $x_{i,j}$ is a reflexion argument of $c_i(x_i)^*$ and $s_k (i+1 \leq k \leq n)$ is any term.

Variables are not recursive and need not be hierarchically introduced. We say that we declared them: for example, [$a$|$b$]:bool, [$m$|$n$]:nat, [$j$|$k$|1]:list, etc.

Finally, defined function constants are introduced by stages with the help of *definitions by cases* (see [6, 11]). Here are some concrete examples:

$a \Rightarrow b$: bool ⇐
cases $a$ [true ⇐ $b$|
false ⇐ true]

$a \,\&\, b$: bool ⇐
$(a \Rightarrow (b \Rightarrow \text{false})) \Rightarrow \text{false}$

$m = n$: bool ⇐
cases $m$[zero ⇐ cases $n$[zero ⇐ true|
succ($n_1$) ⇐ false]|
succ($m_1$) ⇐ cases $n$[zero ⇐ false|
succ($n_1$) ⇐ $m_1 = n_1$]]

They introduce the function constants $\Rightarrow$, & and = for terms of type nat. As usual, the arguments of $\Rightarrow$ are called *antecedent* and *consequent* respectively; the arguments of & are called *conjuncts*. The computer program makes use of a different concrete syntax.

The abstract syntax of a definition by cases is characterized by the following BNF grammar:

⟨definition by cases⟩ :: = ⟨head⟩⟨body⟩

⟨head⟩ :: = ⟨heading⟩⟨type constant⟩

⟨heading⟩ :: = ⟨defined function constant⟩ {⟨variable⟩}

⟨body⟩ :: = ⟨empty⟩ | ⟨case expression⟩

⟨case expression⟩ :: = ⟨term⟩ | ⟨case variable⟩ {⟨case clause⟩}

⟨case variable⟩ :: = ⟨variable⟩

⟨case clause⟩ :: = ⟨pattern⟩⟨case expression⟩

⟨pattern⟩ :: = ⟨constructor constant⟩ {⟨variable⟩}

When the body is empty, then we say that the function constant is *vacuously defined*. The *recursion arguments* of a function constant are the arguments in the position of the case variables in the head of its definition.

Admissible definitions are as follows: All type constants, variables, and function constants apart from the one being defined, must have been previously introduced. Patterns must be well-typed and their variables, distinct.

Now consider the body of a definition as a rooted tree whose nonterminal and terminal vertices are labelled with case variables and terms respectively, and whose arcs are labelled with patterns. Let a *bundle tied by a case variable* $x$ be the set of patterns labelling the arcs directed away from the vertex labelled by $x$. Then the following additional admissibility conditions must be fulfilled:

(1) On case variables: they must occur in the heading and be distinct on any one path to a terminal vertex.

(2) On patterns: the constructor constants of the patterns in the bundle tied by a variable $x$ must be precisely the constructor constants for the type of $x$.

(3) On terms: the variables of any terms $t$ must occur in the heading or in the patterns on the path to $t$, but not as case variables on this path; furthermore, if the function constant being defined occurs in $t$, its recursion arguments, properly ordered, must be an immediate predecessor of the patterns labelling the path to $t$.

### *2.3. Inference rules*

This section expounds the legitimate inferences which can be made in one atomic step. Each one is given as a list of *hypotheses* separated from a *conclusion* by a line; hypotheses and conclusions are terms. If we can also infer the conjunction of the

hypotheses from the conclusion, we write a double line. We then say that the rule is *invertible. Axioms* are inference rules with an empty list of hypotheses.

### *2.3.1. Truth*

$$\frac{}{\text{true}(\,)}$$

### *2.3.2. Specialization*

$$\frac{u}{u[t/x]}$$

### *2.3.3. Definition by k-recursion*

Definition by $k$-recursion is defined by means of a mapping from definitions by cases to sets of inference rules. For each term $t$ labelling a terminal vertex in the body of a definition by cases, we have the inference rule:

$$\frac{w[f(s^*)[u^*/z^*]/x]}{w[t[u^*/z^*]/x]}$$

where $f(s^*)$ is the heading of the definition by cases, with each case variable occurring in it and labelling a vertex on the path to $t$ replaced by the pattern labelling the arc directed away from it on this path, and where $z^*$ is the list of distinct variables in $f(s^*)$.

In particular, we have:

(1) For the function constant $\Rightarrow$:

$$\frac{w[\text{true} \Rightarrow s/x]}{w[s/x]} \qquad \frac{w[\text{false} \Rightarrow s/x]}{w[\text{true}/x]}$$

(2) For the function constant &:

$$\frac{w[s \,\&\, t/x]}{w[(s \Rightarrow (t \Rightarrow \text{false})) \Rightarrow \text{false}/x]}$$

(3) For a polymorphic equality function constant of type $\tau$, $\tau \to$ bool, for every pair of constructor constants $c_1, c_2$ for type $\tau$:

$$\frac{w[c_1(t^*) = c_2(s^*)/x]}{w[\text{false}/x]}$$

if $c_1$ is different from $c_2$, and

$$\frac{w[c_1(t^*) = c_2(s^*)/x]}{w[t_1 = s_1 \,\&\, \cdots \,\&\, t_n = s_n/x]}$$

if $c_1$ is identical to $c_2$.

### *2.3.4. Modus ponens*

$$\frac{s \quad s \Rightarrow t}{t}$$

### *2.3.5. Substitutivity of equality*

$$\frac{}{u[x/z] \,\&\, y = x \Rightarrow u[y/z]}$$

### *2.3.6. Induction*

Let $p$ be the number of ways the constructor constants for the types of the variables $z^*$ can occur in this order.

Let $u_i$ $(1 \leqslant i \leqslant p)$ be implications of the form:

$$u[s^*_{i,1}/z^*] \,\&\, \cdots \,\&\, u[s^*_{i,m_i}/z^*] \Rightarrow u[c_{i,1}(x^*_{i,1}), \ldots, c_{i,n}(x^*_{i,n})/z^*]$$

where $s^*_{i,j}(1 \leqslant j \leqslant m_i)$ are precisely the immediate predecessors of $c_{i,1}(x^*_{i,1}), \ldots, c_{i,n}(x^*_{i,n})$. (The variables distinct from $z^*$ in all occurrences of $u$ in the antecedent are also implicitly replaced by distinct metavariables over terms.)

Then the induction rule can be simply stated as

$$\frac{u_1 \cdots u_p}{u}.$$

The variables $z^*$ are called *induction variables.*

For example, double induction on the variables $m$ and $n$ of type nat is:

$$\frac{\begin{array}{l} u[\text{zero}/m][\text{zero}/n] \\ u[\text{zero}/m][n_1/n] \Rightarrow u[\text{zero}/m][\text{succ}(n_1)/n] \\ u[m_1/m][s_1/n] \Rightarrow u[\text{succ}(m_1)/m][\text{zero}/n] \\ u[m_1/m][s_2/n] \,\&\, u[\text{succ}(m_1)/m][n_1/n] \\ \quad \Rightarrow u[\text{succ}(m_1)/m][\text{succ}(n_1)/n] \end{array}}{u}$$

where $s_1$ and $s_2$ are any terms.

## *2.4. Deductions and proofs*

A *deduction of a term t from a finite set of terms S* is a finite acyclic directed graph, with a set of terms $T$, including $t$ and the elements of $S$, as set of vertices, with a set of arcs $A$, and such that:

(1) If the terms $u_1, \ldots, u_n$, all in $T$, are the initial vertices of $n$ arcs in $A$ directed toward a term $u$ in $T$, then $u$ is an immediate consequence of $u_1, \ldots, u_n$ by virtue of an inference rule.

(2) The terms in $S$ have no arcs directed toward them; $t$ has zero or more arcs directed toward it; other terms in $T$ have at least one arc directed toward them.

(3) The terms in $S$ have zero or more arcs directed away from them; the term $t$ has no arcs directed away from it; and the other terms in $T$ have at least one arc directed away from them.

The term $t$ is called the *conclusion* of the deduction; the terms in $S$, the *hypotheses*. The *degenerate deduction of* $t$ is the deduction of $t$ from the singleton of $t$. A deduction which is a subgraph of another deduction $D$ is called a *subdeduction of* $D$.

A *proof of a term* $t$ is a deduction of $t$ from the empty set of hypotheses; the conclusion of a proof is called a *theorem*. In effect, the conclusion of a deduction need not be a theorem nor need the hypotheses. For example,

$$\dfrac{\dfrac{\dfrac{\text{false}}{\text{false} = \text{true}}}{(\text{true} \Rightarrow \text{false}) = \text{true}} \qquad \dfrac{\dfrac{\text{true}}{\text{true} = \text{true}}}{(\text{false} \Rightarrow \text{false}) = \text{true}}}{(a \Rightarrow \text{false}) = \text{true}}$$

is the deduction of $(a \Rightarrow \text{false}) = \text{true}$ form from the singleton of false.

## 3. Semantics

Our formal syntactic constructs are intended to denote some objects. I will first study the domain of interpretation of the constructs, and then describe this interpretation precisely. The technical background can be found in [16, 9].

### *3.1. Induction*

We actually want our domain of interpretation to have more structure than being just a collection of sets and we impose an algebraic structure on it. More specifically, our domain is a *many-sorted word algebra generated by the empty set.* Such an algebra $M = [(S); (c)]$ consists of a family $(S)$ of $k$ sets $(1 \leqslant k)$ called *carriers* and a collection of $n$-ary $(0 \leqslant n)$ functions from $n$ sets in $(S)$ to a set in $(S)$ called *constructors.* A constructor which maps into a carrier $S$ is said to be a *constructor* of $S$.

The elements of $M$ are precisely those obtained by applying the constructors in $(c)$; they are given the name of *structures.* A constructor $c$ from $S^*$ to $S$ such that $S$ is not a member of $S^*$ is called a *constant constructor with respect to* $S$.

Finally, such algebras have the property of being *totally free* from any special identity relation, that is, no nontrivial relations of the form $s_1 = s_2$ hold in it, where $s_1$ and $s_2$ are distinct elements of the algebra. This is the *unique factorization property.* It

says in our case that two structures of $M$ are identical if and only if they have been constructed by the same constructor from identical structures.

For $s$ and $t$ in any carrier $S$, we say that $s$ is an *immediate substructure* of $t$ in $S$ if and only if $t$ is the result of applying a constructor to $s$ and possibly some other structures. The reflexive and transitive closure of $S$ with the immediate substructure relation is the ordered set $[S; \leqslant]$, where $\leqslant$ denotes the *substructure relation*. More specifically, for every $s$ and $t$ in $S$, $s \leqslant t$ if and only if $s = t$ or $s \leqslant t'$, where $t'$ is an immediate substructure of $t$. As usual, I will write $s < t$ (read: '$s$ is a proper structure of $t$') in place of $s \leqslant t$ and $s \neq t$.

A structure $s$ is *minimal* in $[S; \leqslant]$ if it has no proper substructures in $S$. The minimal elements of $[S; \leqslant]$ are precisely the structures constructed by applying the constant constructors with respect to $S$.

**Fact 1.** *Every nonempty subset of* $[S; \leqslant]$ *has a minimal element* (*the minimum condition*).

**Proof.** This holds since any element $s$ of $S$ has a finite number of substructures by construction.

Note however, that not all carriers $[S; \leqslant]$ are partly well-ordered since we have e.g., for tree structures infinitely many minimal elements: nulltree, tip(zero), tip(succ(zero)), ....

Now we define the strict *lexicographic ordering* $<_L$ on $S^*$, where $S^*$ is the product of not necessarily distinct carriers of $M$, the usual way; for all $s^*$ and $t^*$ in $S^*$, $s^* <_L t^*$ if and only if $s_1 = t_1$ and $\cdots$ and $s_{i-1} = t_{i-1}$ and $s_i < t_i$, for some $i$ $(1 \leqslant i \leqslant n)$. The ordered set $[S^*; \leqslant_L]$ is the reflexive closure of $[S^*; <_L]$, i.e., $s^* \leqslant_L t^*$ if and only if $s^* <_L t^*$ or $s^* = t^*$.

**Fact 2.** *The ordered set* $[S^*; \leqslant_L]$ *satisfies the minimum condition.*

**Proof.** This holds since $[S; \leqslant]$ satisfies the minimum condition for each $S$ in $S^*$.

We can now assert that the *principle of structural induction* holds for the ordered set $[S^*; \leqslant_L]$, that is:

If, for all $t^*$ in $S^*$, $P(s^*)$ implies $P(t^*)$, for all $s^*$ in $S^*$ such that $s^* <_L t^*$, then $P(t^*)$, for all $t^*$ in $S^*$.

We say that $s^*$ is an *immediate predecessor* of $t^*$ in $[S^*; \leqslant_L]$ if $s^* <_L t^*$ and there is no other element $r^*$ in $S^*$ for which $s^* \leqslant_L r^* \leqslant_L t^*$. The principle of structural induction can be reformulated in an equivalent, though apparently stronger, way by replacing '$s^* <_L t^*$' by '$s^*$ is an immediate predecessor of $t^*$'.

### 3.2. *k-recursive functions*

This section studies a class of functions over the carriers of the algebra H.

A function $f$ from $S^* \times T^*$ to $S$, such that $S^*$, $T^*$ and $S$ are carriers of $M$, is said to be *defined by k-recursion* if and only if for all lists of $k$ constructors $c^*$ of $S^*$ respectively, $f(c_1(x_1^*), \ldots, c_k(x_k^*), y^*)$ is explicitly defined using only:

(1) The variables $x_1^*, \ldots, x_k^*, y^*$,

(2) the functions $\lambda y^* \cdot f(z^*, y^*)$, where $z^*$ is an immediate predecessor of $c_1(x_1^*), \ldots, c_k(x_k^*)$ in $[S^*; \leqslant_L]$,

(3) previously defined functions.

Note that an explicit definition is finite. In a sense, this definition scheme says too much and too little. It says too much because, insofar as Peter's results for number theory [15] are applicable to this system, definitions by $k$-recursion are reducible to a normal form which does not look like the above definition scheme. But this reduction is a bit artificial and in this system, I wish to deal with definitions by $k$-recursion as people would naturally write them. But then, the above scheme says too little since it does not cater for course-of-values and mutual recursion. The reasons behind it are essentially pragmatic and I will come back to them.

We inductively define the class of *k-recursive functions* thus:

(1) constructors are $k$-recursive functions,

(2) if $f$ is a function defined by $k$-recursion from $k$-recursive functions, then $f$ is a $k$-recursive function,

(3) a function is $k$-recursive only as required by (1) and (2).

**Fact 3.** *There exists a unique function f from $S^* \times T^*$ to S which satisfies a given definition by k-recursion.*

**Proof.** The proof by induction on the class of $k$-recursive functions and on $[S^*; \leqslant_L]$ divides into two parts. We consider all lists $c^*$ of constructors of $S^*$ respectively.

*Existence.* For all $x^*$ in appropriate carriers, all $y^*$ in $T^*$, by induction hypothesis, we have that

(1) there exists a $z$ in $S$ such that $f(z^*, y^*) = z$ for all $z^*$ immediately preceding $c_1(x_1^*), \ldots, c_k(x_k^*)$ in $[S^*; \leqslant_L]$ and for all $y^*$ in $T^*$ and

(2) for any previously defined function $g$, there exists a $z$ in an appropriate carrier such that $g(y^*) = z$ for all $y^*$ in appropriate carriers.

But $f(c_1(x_1^*), \ldots, c_k(x_k^*), y^*)$ is explicitly defined in terms of these functions only and of constructors. Hence, there exists a $z$ in $S$ such that for all $x^*$ in appropriate carriers, for all $y^*$ in $T^*$, $f(c_1(x_1^*), \ldots, c_k(x_k^*), y^*)$.

*Uniqueness.* Suppose some function $f'$ also satisfies the definition by $k$-recursion of $f$ for all $x^*$ in appropriate carriers and for all $y^*$ in $T^*$, then by induction hypothesis, we have that $f'(z^*, y^*) = f(z^*, y^*)$ for all $z^*$ immediately preceding

$c_1(x_1^*), \ldots, c_k(x_k^*)$ in $[S^*; \leqslant_L]$ and for all $y^*$ in $T^*$. Hence,

$$f'(c_1(x_1^*), \ldots, c_k(x_k^*), y^*) = f(c_1(x_1^*), \ldots, c_k(x_k^*), y^*)$$

for all $x^*$ in appropriate carriers and for all $y^*$ in $T^*$.

In conclusion, there exists a unique $z$ in $S$ such that $f(x^*) = z$ for all $x^*$ in $S^* \times T^*$; so, there exists a unique function which satisfies a given definition by $k$-recursion.

### *3.3. Interpretation*

Now that we have a reasonably clear picture of what our domain of interpretation looks like, we can give the intended meaning of our syntactic constructs. Since this section, and the following ones, will mention both syntactic constructs and objects in our domain, the latter will be underlined. Identity between elements of the domain will be written as $=$.

An interpretation is a triple $(C, M, V)$ of semantic functions, respectively called *classification*, *model*, and *valuation*. These functions map syntactic constructs into semantic objects.

We define the semantic function $C$ (classification) for type constants thus: $C$ assigns a carrier $S$ to each type constant $\sigma$.

In particular, we have that $C$(bool) is BOOL, the set of truthvalues.

The semantic functions $M$ (model) and $V$ (valuation) for other syntactic constructs in the language are mutually defined:

(1) $M$ assigns a constructor $c$ from $S^*$ to $S$ to each constructor constant $c$ of type $\sigma^* \to \tau$ where $C(\sigma_i)$ is $S_i$ and $C(\tau)$ is $S$.

(2) To each function constant $f$ of type $\sigma^* \to \tau$ defined by cases, $M$ assigns a function $f$ from $S^*$ to $S$ defined by $k$-recursion, where $C(\sigma_i)$ is $S_i$ and $C(\tau)$ is $S$.

If $f$ is vacuously defined, then $f$ is any $k$-recursive function from $S^*$ to $S$. Otherwise the definition of $f$ by $k$-recursion is formed of the following of clauses: for all terms $t$ labelling terminal vertices in the definition body of $f$,

$$V(f(x^*)[z^*/s^*]) = V(t) \quad \text{with } M(f) = f$$

where $f(x^*)$ is the heading of the definition head of $f$; $z^*$ is the list of case variables labelling the vertices on the path to $t$; and $s^*$ are the patterns tied to these variables on this path.

(3) $V$ assigns an element of $S$ to each variable of type $\sigma$ such that $C(\sigma)$ is $S$.

(4) $V(f(t^*)) = M(f)(V(t_1), \ldots, V(t_n))$.

(5) $M$ is a model and $V$, a valuation only as required by (1), (2), (3) and (4).

In particular, we have that $M$(true) is **true** and $M$(false) is **false**; the meanings of the function constants $\Rightarrow$, &, and $=$ are the functions respectively defined by:

$$\textbf{true}(\,) \Rightarrow b = b$$
$$\textbf{false}(\,) \Rightarrow b = \textbf{true}(\,)$$

$$a \,\&\, b = (a \Rightarrow (b \Rightarrow \mathbf{false}(\,)) \Rightarrow \mathbf{false}(\,)$$

$$\mathbf{c}_1(x^*) = \mathbf{c}_2(v^*) = \mathbf{false}(\,), \qquad \text{if } \mathbf{c}_1 \text{ is different from } \mathbf{c}_2.$$
$$\mathbf{c}_1(x^*) = c_2(y^*) = x_1 = y_1 \,\&\, \cdots \,\&\, x_n = y_n, \qquad \text{otherwise.}$$

With the help of the semantic functions $C$, $M$, and $V$, we can obtain a value (i.e. a structure) for any term in our language. For the moment, we will focus our interest on boolean terms. Let $B = [(\mathrm{BOOL}, S); (\mathbf{true}, \mathbf{false}, c)]$ be a many-sorted algebra. We say that a term $t$ of type bool is **valid** in $B$ if and only if $V(t) = \mathbf{true}(\,)$ for all values of its variables and vacuously defined function constants.

Before closing this section, there remains a point to be clarified. We have seen, for example, that the meaning of $\Rightarrow$ of type bool, bool $\to$ bool is the function $\Rightarrow$ from BOOL $\times$ BOOL to BOOL. However, one can legitimately ask whether the function $\Rightarrow$ carries the same information as implication. The question arises because we have a logic of terms only. We first need some definitions. For all functions $f$ from $S^*$ to BOOL with $S_i$ different from BOOL for some $S_i$ of $S^*$, we define the relation $P[f]$ such that $P[f](x^*)$ if and only if $f(x^*) = \mathbf{true}(\,)$; such functions $f$ are called *predicates*. Similarly, for all functions $f$ from BOOL* to BOOL, we define the composition of sentential connectives ('implies', 'and', etc.) $Q[f]$ such that $Q[f]$ $(P[\mathbf{g}_1](x_1^*), \ldots, P[\mathbf{g}_n](x_n^*))$ if and only if $f(\mathbf{g}_1(x_1^*), \ldots, \mathbf{g}_n(x_n^*)) = \mathbf{true}(\,)$; such functions $f$ are called *connectives*.

**Fact 4.** *The interpretation respects truth.*

**Proof.** We have that $P[\mathbf{true}]$ is the true relation since $\mathbf{true}(\,) = \mathbf{true}(\,)$.

**Fact 5.** *The interpretation respects falsity.*

**Proof.** We have that $P[\mathbf{false}]$ is the false relation since $\mathbf{false}(\,) \neq \mathbf{true}(\,)$.

**Fact 6.** *The interpretation respects implication.*

**Proof.** We want to show that $Q[\Rightarrow]$ is the sentential connective 'implies'. This is immediate from the fact that truth and falsity are respected by the interpretation. For example, $(\mathbf{true}(\,) \Rightarrow \mathbf{false}(\,)) = \mathbf{false}(\,)$ if and only if the true relation does not imply the false relation.

**Fact 7.** *The interpretation respects conjunction.*

**Proof.** This is immediate from the previous results.

**Fact 8.** *The interpretation respects equality.*

**Proof.** This is the most interesting case. We want to show that $(x = y) = \mathbf{true}(\,)$ if and only if $x = y$ for all $x$ and $y$ in a carrier $S$; in other words, $P[=]$ is the identity relation. The proof is by induction on the family $(S)$ of carriers as hierarchically introduced by the type definitions, and on the ordered sets $[S; \leqslant_L]$. Suppose $\mathbf{c}^*$ are precisely the constructors of $S$ and consider all pairs of constructors $\mathbf{c}_1$ and $\mathbf{c}_2$. We have that

$$(\mathbf{c}_1(x^*) = \mathbf{c}_2(y^*)) = \mathbf{true}(\,) \quad \text{iff} \quad (x_1 = y_2 \,\&\, \cdots \,\&\, x_n = y_n) = \mathbf{true}(\,).$$

But by induction hypothesis, equality is respected for previously introduced carriers and for elements of $S$ preceding $\mathbf{c}_1(x^*)$ and $\mathbf{c}_2(y^*)$. So, we have $x_1 = y_1$ and $\cdots$ and $x_n = y_n$, since conjunction is respected. We finally get $\mathbf{c}_1(x^*) = \mathbf{c}_2(y^*)$ because of the unique factorization property of identity. In conclusion, our interpretation respects equality.

## 4. Soundness

The least property which a formal system must have if we want to give any substance to our claim of proving theorems is soundness, that is, we want to make sure that the terms provable in the system are indeed valid.

**Fact 9.** *If a boolean term t is a theorem, then t is valid.*

**Proof.** The demonstration is by induction on the structure of proofs. We must show that for each rule of inference, if the hypotheses are valid, then the conclusion is also valid.

*Truth.* We have that $V(\text{true}(\,)) = M(\text{true})(\,) = \mathbf{true}(\,)$.

*Specialization.* Since $u$ is valid, it is **true**( ) for all values assigned to its vacuously defined function constants and variables. In particular, it is **true**( ) for $V(t)$ assigned to $x$ for all values of the vacuously defined function constants and of the variables in $t$. Hence, $u[t/x]$ is valid.

*Definition by k-recursion.* By the definition of $M$ and $V$, the inference rules constituting the definition by $k$-recursion of a defined function constant $f$ and the clauses of the definition by $k$-recursion of $M(f)$ correspond precisely. So, for any constituent of the definition of $f$,

$$\frac{w[f(s^*)/x]}{w[t/x]}$$

if and only if $V(f(s^*)) = V(t)$. But, the latter identity holds since we have shown that functions defined by $k$-recursion are well-defined. Finally, since the identity relation is substitutive, we have that $w[f(s^*)/x]$ is valid if and only if $w[t/x]$ is valid.

*Modus ponens.* Assume that $s$ and $s \Rightarrow t$ are valid. Then we have that $V(s) = \textbf{true}()$, and $V(s) = \textbf{true}()$ implies $V(t) = \textbf{true}()$, since the interpretation respects implication. So, we can deduce that $V(t) = \textbf{true}()$ and hence, that $t$ is valid.

*Substitutivity of equality.* Since implication, conjunction, and equality are respected by the interpretation, this axiom is valid if and only if $V(u[x/z]) = \textbf{true}()$ whenever $V(u[y/z]) = \textbf{true}()$ and $y = x$. But, this is precisely equivalent to the substitution principle for the identity relation which holds in our domain.

*Induction.* Two facts should clear from the start:

(1) a list of terms of types $\tau^*$ is an immediate predecessor of another list of terms (in the sense of Section 2.2) if and only if the list of values of the first terms immediately precedes the list of values of the second terms in $[C(\tau_1), \ldots, C(\tau_n); \leqslant_L]$ (in the sense of Section 3.1);

(2) $c_1^*, \ldots, c_m^*$ are precisely the lists of constructor constants for the types $\tau^*$ if and only if the values of the terms $c_{1,1}(x_{1,1}^*), \ldots, c_{1,n_1}(x_{1,n_1}^*), \ldots, c_{m,1}(x_{m,1}^*), \ldots, c_{m,n_m}(x_{m,n_m}^*)$, for all values of the variables, are precisely the elements of $[C(\tau_1), \ldots, C(\tau_n); \leqslant_L]$.

Now assume that each $u_i$ $(1 \leqslant i \leqslant p)$ in the induction rule is valid. Then, for each of them, we have that:

$$V(u[s_i^*/z^*]) = \textbf{true}() \text{ and } \cdots \text{ and } V(u[s_m^*/z^*]) = \textbf{true}()$$

implies

$$V(u[c_1(x_1^*), \ldots, c_n(x_n^*)/z^*]) = \textbf{true}(),$$

under the provisos on $c_i$ and $s_j^*$ given in Section 2.3. But because of the two facts above, and by means of the principle of structural induction, we can deduce that $V(u) = \textbf{true}()$ and hence, that $u$ is valid.

This completes the proof.

## 5. Weak completeness

The incompleteness result of number theory extends to this formal system despite its limited form of quantification (i.e. an implicit outermost universal quantifier for all variables.) It is, however, weakly complete in the sense that every valid term without variables and vacuously defined function constants is a theorem.

**Fact 10.** *If terms $t$ and $s$ do not contain any variables and vacuously defined function constants, then $s = t$ whenever $V(s) = V(t)$.*

**Proof.** The proof is by induction on the class of terms. If $t$ or $s$ are variables, then the theorem holds vacuously. Let $t$ be $f_1(t^*)$ and $s$ be $f_2(s^*)$; by induction hypothesis, we have that $t_i = s_i$ whenever $V(t_i) = V(s_i)$. If both $f_1$ and $f_2$ are constructor constants, then by the unique factorization property of equality, we can deduce that $f_1(t^*) = f_2(s^*)$ whenever $V(f_1(t^*)) = V(f_2(s^*))$. If at least one of $f_1$ or $f_2$ is a (nonvacuously)

defined function constant, then by the uniqueness of functions defined by $k$-recursion, we also have that $f_1(t^*) = f_2(s^*)$ whenever $V(f_1(t^*)) = V(f_2(s^*))$. This completes the proof.

As a matter of fact, the converse also holds. This justifies what will be called *evaluation*, that is, the repeated application of the $k$-recursive definition rule to a term $t$ without variables and vacuously defined function constants, until it cannot be applied any more. When $t$ contains variables or vacuously defined function constants, we talk of *symbolic evaluation.*

The weak completeness theorem is a corollary of the above proposition.

**Fact 11.** *Every valid term without variables and vacuously defined function constants is a theorem.*

**Proof.** In other words, we want to show that if $V(t) = \mathbf{true}()$, then $t$ is a theorem. Assume $V(t) = \mathbf{true}()$; then by Fact 10, $t = \text{true}()$. But this is equivalent to $t \Rightarrow \text{true}()$ and $\text{true}() \Rightarrow t$; hence, by modus ponens, true( ) and $t$ are interdeducible. In conclusion, $t$ is a theorem.

## 6. Discussion

One objective of this formal system was to start from a small base in order to achieve a great degree of uniformity as regards, e.g., induction. However, as it stands, the result is not amenable to automatic proof-finding. The level of the system has to be raised by introducing more connectives (or, not, cond) and by deriving some inference rules. The latter include various forms of substitutivity for equality and a weakening rule, i.e., from $t$, infer $s \Rightarrow t$. Other rules are actually theorems; they are equalities used to put a term in normal form, i.e., a conjunction of implications whose antecedents and consequents are conjunctions and disjunctions respectively. These normalization rules have been inspired from Ketonen's dialect [12] of Gentzen's sequent calculus [10]. Proofs can then be carried out more easily.

The main feature of this system, that is, its typed language, is of great pragmatic importance. By considering abstract structures independently of their concrete representations, it is easier to prevent and detect meaningless constructions (by static type checking) and possible to obtain simpler expressions and proofs. The lack of separation between abstract data types and their representations was a serious source of difficulty for Boyer and Moore [4] when going from lists to more complex types.

The natural counterpart to type definitions is the definition of functions by cases. Case expressions are less prone to error than the conditional expressions exploited

by Boyer and Moore [4]. More importantly, because they are used by matching, they allow recursion arguments to be of a type containing infinitely many minimal elements (e.g. type tree in Section 2.2); conditionals do not so easily.

Besides the positive features, one must have noticed some of the restrictions of this system. All of them aim at simplifying the search for proofs. As a result, the power of expression of the language may suffer, but hopefully, the effects will not be so important for the class of problems considered. The expression of mutually reflexive types has been barred because no rules or strategies have been studied for them. Course-of-values and mutual $k$-recursion are excluded for similar reasons; nevertheless $k$-recursion from several bases is included though not explicitly in the scheme of Section 3.2. Dealing with general recursive functions would require an even more radical change to the strategy since the analogy between recursion and structural induction would be lost; partialness would also be introduced.

Finally, this language has no quantifiers. This cuts down an important source of complexity, but in this case, it also brings some limitations (while it does not in resolution, for example). In effect, one can think of the boolean terms as first-order formulas with outermost universal quantifiers only, except for the non-induction variables in the induction hypotheses which are existentially quantified. Thus the specialization of variables is restricted; this reduces the search space considerably, and does not appear to be too limiting to serve our purposes.

Two recent works have also had the goal of improving over Boyer and Moore [4]. Cartwright's system [7, 8] includes axiomatically defined structures constrained to belong to the same sets as denoted by our type constants. However, axiomatic definitions allow membership to a set to be discussed within the formal language and consequently, yield a more powerful induction rule. Moreover, Cartwright deals with general recursive functions by the addition of an undefined value. Boyer and Moore [3] also improved on their earlier work. Their new system is even more powerful than Cartwright's since structures can be defined axiomatically without any constraints. On the other hand, the system can be extended with any total function.

This increased power of Cartwright's, and Boyer and Moore's systems has a price: since well-typedness is not a syntactic feature, it has to be dynamically proved as opposed to statically checked as in my system. Moreover, in the case of Boyer and Moore, one has to show that a set admits induction before using this rule on it and also that a function is total before introducing it in the system. In my system, admissibility to induction and totality are syntactic features and consequently, can be statically checked.

## 7. Conclusions

The formal system presented in this paper sets up the basis of a theorem-proving system whose search strategy is described in a subsequent paper (see [2]). Thus the

claim of proving theorems can be substantiated formally. The pragmatics of proof-finding was the prime motivation for many aspects of the system, in particular, its typed language. Some extensions are worth studying (richer domain than word algebra, general recursive functions), but not independently of the problem of finding proofs in such an improved system.

## Acknowledgment

This work was carried out at the School of Computer Science and Artificial Intelligence, University of Edinburgh. I am especially grateful to my directors of studies, Robin Milner and Rod Burstall. I was supported by the Commonwealth Scholarship Commission and the Conseil national de recherches du Canada.

## References

[1] R. Aubin, Mechanizing structural induction, Ph.D. thesis, University of Edinburgh, Edinburgh (1976).

[2] R. Aubin, Mechanizing structural induction, part II: Strategies, *Theoret. Comput. Sci.* **9**(3) (1979) 347–362, this volume.

[3] R.S. Boyer and J. Moore, A lemma driven automatic theorem prover for recursive function theory, *Proc. 5th International Joint Conference on Artificial Intelligence* (1977) 511–519.

[4] R.S. Boyer and J. Moore, Proving theorems about LISP functions, *J.ACM* **22** (1975) 129–144.

[5] D.K. Brotz, Embedding heuristic problem solving methods in a mechanical theorem prover, STAN-CS-74-443, Computer Science Department, Stanford University (1974).

[6] R.M. Burstall, Proving properties of programs by structural induction, *Comput. J.* **12** (1969) 41–48.

[7] R. Cartwright, A practical formal semantic definition and verification system for TYPED LISP, Ph.D. thesis, Stanford University (1976).

[8] R. Cartwright, User-defined data types as an aid to verifying LISP programs, in: S. Michaelson and R. Milner, Eds., *Proc. Third Int. Coll. Automata, Languages and Programming* (Edinburgh University Press, Edinburgh, 1976) 228–256.

[9] P.M. Cohn, *Universal Algebra* (Harper and Row, New York, 1965; Weatherhill, Tokyo, 1965).

[10] G. Gentzen, Recherche sur la déduction logique, trad. et comm. R. Feys et J. Ladrière (Presses universitaires de France, Paris, 1955).

[11] C.A.R. Hoare, Recursive data structures, *Internat. J. Comput. Information Sci.* **4** (1975) 105–132.

[12] O. Ketonen, Untersuchungen zum Pradikatenkalkul, rev. P. Bernays, *J. Symbolic Logic* **10** (1945) 127–130.

[13] R. Milner, L. Morris and M. Newey, A logic for computable functions with reflexive and polymorphic types, in: G. Huet et G. Kahn, Eds., *Actes coll. Construction, Amélioration et Vérification de Programmes* (Institut de recherche d'informatique et d'automatique, Rocquencourt, France, 1975) 371–394.

[14] J. Strother Moore, Computational logic: structure sharing and proof of program properties, Ph.D. thesis, University of Edinburgh, Edinburgh (1973).

[15] R. Peter, *Recursive Functions* (Academic Press, New York, 1967).

[16] F.P. Preparata and R.T. Yeh, *Introduction to Discrete Structures* (Addison-Wesley, Reading, MA, 1973).

[17] J.W. Robbin, *Mathematical Logic* (Benjamin, New York, 1969).