

A NATURAL PROGRAMMING CALCULUS

Åke Hansson

Department of Computer Science
University of Stockholm and
The Royal Institute of Technology
106 91 Stockholm
Sweden

Sten-Åke Tärnlund

We shall lay down a programming calculus in which we develop a methodology for reasoning about data and programs. Our language, L, is ordinary first order predicate logic. A subset of this language, Lp, is our programming language that can be run efficiently in PROLOG. The calculus consists of a natural deduction system that is used for deducing programs and a system for efficient computation of programs. The axioms of the calculus characterize the data structures. Definitions are used for program specifications and mappings between data structures. The programs are deduced from the axioms and the definitions. Examples of deduced programs are a LISP-program and a program that Burstall and Darlington could not obtain in their transformation system.

1. INTRODUCTION

Our main purpose is to develop a methodology for reasoning about data and programs, and especially to enable us to arrive at a programming calculus. In this connection we shall take up a system of natural deduction invented by Jaśkowski and Gentzen, and from this we shall deduce programs.

Natural deduction systems are natural in several ways. One important point is their characterization of informal reasoning. In particular, the structure of unformalized reasoning preserves its structure when it is formalized. In fact, the suggestion by Łukasiewicz to formalize reasoning from assumptions seems to be a first step towards natural deduction systems.

We can think of a natural deduction system as a formalization of means-ends analysis in the GPS-system (see Newell & Simon [14]). Taking this view, a derivation of a program follows a general structure. An induction hypothesis is an initial state. A program specification together with axioms of data structures or data structure mappings are operators that transform an initial state to a final state that is the program. The general structure of a deduction is of great interest not only for our understanding of program deductions but also for developing methods for automated reasoning. Unfortunately, we have to leave out the deductions for reasons of space. A detailed description of the deductions is found in Hansson & Tärnlund [7].

We shall make use of Quine's [16] natural deduction system for classical logic. Natural deduction systems are not restricted to classical logic but apply also to many other systems e.g., second or-

der logic, modal logic and set theory. A comprehensive description of natural deduction systems as well as of their history is given by Prawitz [15].

Our language is first order predicate logic (L) defined in its standard way with bound and free variables, primitive signs for quantification ($\forall$ and $\exists$), conjunction (&), disjunction ($\vee$), conditional ($\rightarrow$), negation ($\sim$) and bi-conditional ($\leftrightarrow$). In section 2 this language is used for writing down axioms e.g., to characterize data structures, and in sections 3-5 for definitions of programs and data structure mappings. We use the inference rules of the natural deduction system to derive programs from the axioms, definitions and mappings.

The language L should be distinguished from the restricted language of Horn clauses Lp, which is our programming language (see Colmerauer et al. [6], Hayes [8] and Kowalski [10]). Thus a deduction of a program starts from axioms and definitions in L and arrives at a Horn clause program in Lp. Horn clause programs do not restrict the set of computable functions (see Tärnlund [19]) and since the advent of PROLOG (see Roussel [18] and Warren [21]) they can also be run efficiently by a resolution-like theorem prover (see Robinson [17]).

The programming language evidently becomes an integrated part of a logic calculus, in the sense of Church [4], where the deductions are determined by inference rules and axioms. It should be noted that our system consists of a natural deduction system for reasoning and a resolution system for the efficient computation of programs.

*This work was supported by the Swedish Natural Science Research Council.

2. DATA STRUCTURES

We now turn to the notion of data structures, which are premises in our deductions. We shall have to write down these structures explicitly in order to make formal inferences from them. These formalizations are axioms in our calculus.

We shall take up four data structures: simple linear lists, difference lists, binary trees and ordered-binary trees. Our characterization of these data structures has already been introduced in Tärnlund [20].

A simple linear list is empty ($\emptyset$) or constructed $(x.y)$ from an element x and a list y . We formalize this idea in our first axiom.

Axiom 1.

$$\forall w \{ \text{list}(w) \leftrightarrow w = \emptyset \vee \exists x \exists y [w = x.y \ \& \ \text{element}(x) \ \& \ \text{list}(y)] \}$$

We are using a dot, $.$, for list constructions which like the extensions of the elements are supposed to be defined elsewhere.

Generally, there are useful lemmas that could be derived from the axioms which make the deductions of programs shorter. Following Clark & Tärnlund [5] these lemmas consist of an induction step and a base. For reasons of space we have to leave them out.

Our next axiom characterizes a more complicated linear list called a difference list or a d-list for short. It is useful for direct access to the rear of a linear list and for concatenating two lists efficiently. We may think of a d-list as a pair $\langle u, w \rangle$ of linear lists u and w , where each element on w occurs in the same order at the rear of u e.g., $u = a.b.c.x$ and $w = c.x$ giving the pair $\langle a.b.c.x, c.x \rangle$, which denotes the simple list $a.b.\emptyset$. Our second axiom formalizes the idea of a d-list.

Axiom 2.

$$\forall w \{ \text{d-list}(w) \leftrightarrow \forall z \exists u \exists v [w = \langle u, z \rangle \ \& \ (u = z \vee u = v.z \ \& \ \text{element}(v) \vee \text{d-list}(\langle u, v \rangle) \ \& \ \text{d-list}(\langle v, z \rangle))] \}$$

We assume that the pair constructor, $\langle \rangle$, is already defined.

Binary trees are interesting and useful data structures not only for their own sake but also because any tree can be represented by a binary tree. An object is a binary tree if and only if it is empty or constructed $t(x, y, z)$ where y is an element, moreover, the subtrees x and z are binary trees themselves. This idea is formalized in our third axiom.

Axiom 3.

$$\forall w \{ \text{binary-tree}(w) \leftrightarrow w = \emptyset \vee \exists x \exists y \exists z [w = t(x, y, z) \ \& \ \text{element}(y) \ \& \ \text{binary-tree}(x) \ \& \ \text{binary-tree}(z)] \}$$

When unambiguous a binary tree is called a tree.

The concept of ordering is fundamental in programming. We shall need a "belongs-to" relationship in order to specify the idea of ordering.

An element u belongs to a list w if and only if w is identical with a simple list $x.y$ and u is identical with x or u belongs to the list y . This is written more precisely in our fourth axiom.

Axiom 4.

$$\forall u \forall z \{ u \in z \leftrightarrow \text{list}(z) \ \& \ \exists x \exists y [z = x.y \ \& \ (u = x \vee u \in y)] \}$$

This inductive definition is naturally viewed from the front of the list towards the rear. However, it is equally natural to view a list from the rear towards the front. But in order to achieve the latter idea we have to make use of a d-list since a linear list does not work in the latter direction. The following picture might be useful for understanding our next axiom.

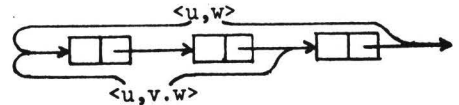


Fig.1 Two d-lists.

An element x belongs to a d-list $\langle u, w \rangle$ if and only if there exists a one-element shorter d-list $\langle u, v.w \rangle$ such that x is identical with v or belongs to a d-list $\langle u, v.w \rangle$. We make this notion concise in an axiom.

Axiom 5.

$$\forall x \forall z \{ x \in z \leftrightarrow \exists z' [\text{d-list}(z) \ \& \ \text{d-list}(z') \ \& \ \forall w \exists u \exists v [z = \langle u, w \rangle \ \& \ z' = \langle u, v.w \rangle \ \& \ (x = v \vee x \in z')]] \}$$

An element u belongs to a binary tree $t(x, y, z)$ if and only if u is identical with y or u belongs to x or z . This idea is written precisely in an axiom.

Axiom 6.

$$\forall u \forall w \{ u \in w \leftrightarrow \text{binary-tree}(w) \ \& \ \exists x \exists y \exists z [w = t(x, y, z) \ \& \ (u \in x \vee u = y \vee u \in z)] \}$$

We finish this section with an axiomatization of an ordered binary tree; ob-tree for short. An empty tree is an ob-tree and a tree $t(x, y, z)$ is ordered if and only if each element on the left subtree x is less than the element y and each element on the right subtree z is greater than y , moreover, the trees x and z are ob-trees.

Axiom 7.

$$\forall w \{ \text{ob-tree}(w) \leftrightarrow \text{binary-tree}(w) \ \& \ [w = \emptyset \vee \exists x \exists y \exists z [w = t(x, y, z) \ \& \ \forall u [(u \in x \rightarrow u < y) \ \& \ (u \in z \rightarrow y < u)] \ \& \ \text{ob-tree}(x) \ \& \ \text{ob-tree}(z)]] \}$$

We shall also need to distinguish between an empty list and a constructed list i.e., $\emptyset \neq x.y$, as

well as between an empty tree and a constructed tree i.e., $\emptyset \neq t(x,y,z)$. We can now, for example, derive that there is no element on an empty list or on an empty tree i.e., $\sim(u \in \emptyset)$. We shall also assume some standard axiomatization of arithmetic.

3. DEDUCTION FROM INDUCTIVE ASSUMPTIONS

The intention of our programming calculus is to arrive at systematic methods for deducing programs. Generally speaking, a deduction of a program follows from a program specification and from the axioms of data structures given in section 2 or mappings between data structures. These axioms, specifications and mappings are written in our first order language L. By contrast, a program is written in the language L_p which usually is Horn clauses and a subset of L.

A deduction is built up on the inference rules in Quine's natural deduction system [16]. In addition we shall also make use of rules for identity.

It is a point of methodological interest that the programs are written inductively on a data structure. This fact is used for deducing programs from their specification by an inductive argument i.e., a program is deduced from an induction hypothesis. Moreover, this method of deducing programs gives a correctness proof of the programs for free. A simple example will shed light on these ideas.

Suppose we wish to deduce a program which checks whether or not an element x is less than all elements of a list l. We owe this "lessall"-program to Manna & Waldinger [12]. First, we write down a definition in the language L which is a precise specification of the program.

Definition 1.

$$\forall x \forall l \{ \text{lessall}(x,l) \leftrightarrow \text{element}(x) \ \& \ \text{list}(l) \ \& \ \forall z [z \in l \rightarrow x < z] \}$$

This definition is explicit in contrast to an implicit one written inductively on a data structure. As we shall see explicit definitions contain more alternative programs than implicit definitions and so we consider them more abstract.

We observe that definition 1 contains a simple list as data structure. In fact, a data structure provides a general rule for generating the structure of a program deduction. First, we assume that there is a program on the data structure e.g., lessall(x,l), where x is an element and l a constructed list. From this assumption a consequent is deduced e.g., lessall(x,u.l), where u.l contains one element more than l. In this way, by the deduction theorem, we can deduce a part of a program, which is an induction step. Secondly, we can readily complete the program by an induction base e.g., lessall(x, $\emptyset$).

We can sum up our deductions in Hansson & Tärnlund [7] in the following theorem, which is a program that checks whether or not an element and a list satisfy the "lessall"-relationship.

Theorem 1.

$$\begin{aligned} &\text{lessall}(x,\emptyset) \leftrightarrow \text{element}(x) \\ &\text{lessall}(x,u.l) \leftrightarrow x < u \ \& \ \text{element}(u) \ \& \ \text{lessall}(x,l) \end{aligned}$$

We follow a convenient convention for logic programs and omit universal quantifiers whose scope is the entire sentence. We can omit the "type"-checks if we assume that the program is used with respect to its intended interpretation. In this way we arrive at the following simple program which we can run efficiently in PROLOG.

$$\begin{aligned} &\text{lessall}(x,\emptyset) \\ &\text{lessall}(x,u.l) \leftarrow x < u \ \& \ \text{lessall}(x,l) \end{aligned}$$

3.1 Program correctness

It is to be noticed that the deduction of our program ensures that the program is correct. For example, a correctness statement of the "lessall"-program is:

$$\forall x \forall l \{ \text{element}(x) \ \& \ \text{list}(l) \ \& \ \text{lessall}(x,l) \rightarrow \forall z [z \in l \rightarrow x < z] \}$$

A meta-argument tells us that this statement is already proved during the deduction of the program. A central idea about the deduction is that it is an induction argument on lists consisting of a base case and an induction step i.e.,

$$P(\emptyset) \ \& \ \forall x \forall y [P(y) \rightarrow P(x.y)] \rightarrow \forall z P(z)$$

where the antecedent consists of the base case and the induction step of the "lessall" program respectively. So, by induction we prove that the program is true of exactly all elements and lists in the "lessall"-relationship, i.e., we have a model of

$$\text{lessall}(x,l)$$

which by definition 1 is true for

$$\forall z [z \in l \rightarrow x < z]$$

It remains to be proved that the program terminates. We can, in fact, do this by induction on the data structure following methods developed in Clark & Tärnlund [5].

4. DEDUCTION FROM AN ABSTRACT SPECIFICATION

Deductions of programs from specifications usually contain several alternatives of data structures as well as a choice between operations on data structures. We shall say that a specification is less abstract when it provides fewer deductions. In this way an abstract specification gives more power and a natural idea would be to write down just abstract definitions. Up to a point such a

development has great advantages, as we shall see in section 4.1, where different programs are deduced from alternative assumptions about data structures and in section 4.3, where new programs are deduced from a choice between operations on data structures.

Unfortunately, all specifications are not abstract enough to provide alternative deductions of programs. This is certainly true of specifications that are very close to being programs themselves. In order to arrive at alternative programs in this situation we can of course try to write down a more powerful specification, but then we are faced with the problem of showing that the original specification satisfies the more abstract one. However, we can also deduce an alternative program from the original specification by introducing a mapping between data structures. We take up this idea in section 4.2.

4.1 Hypothesis About Alternative Data Structures

Evidently we can make use of our natural deduction system for hypothetical reasoning introduced in the preceding section and thus systematically deduce programs from alternative hypothesis.

Let us get a firmer idea by an example. Suppose that we have an abstract definition of programs for computing a list of factorials. Let us further assume that we first of all want to deduce a program with the factorials in descending order in the list and, secondly, to deduce a program with the factorials in ascending order. We owe the first of these programs to Burstall & Darlington [2]. The idea of putting each factorial at the front of the list is a stack operation in character and leads to a list of factorials in descending order. By contrast, the idea of putting each new factorial at the rear of the list is a queue operation in nature and gives the factorials in ascending order. These broad outlines of deducing programs from abstract specifications with alternative hypothesis about data structures are depicted in figure 2.

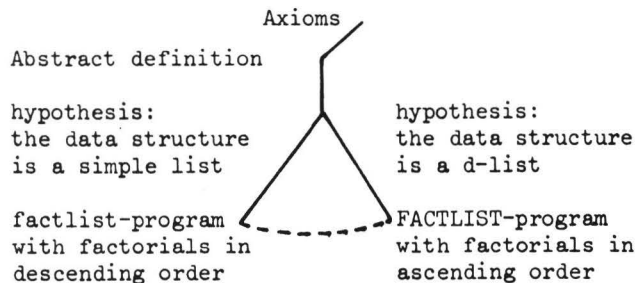


Fig.2 Deductions of alternative programs from the same specification

We shall use bounded quantifiers. We say that

$\forall F(y)$ is equivalent to $F(1) \& F(2) \dots \& F(n)$ and $1 \leq y \leq n$
 $\exists F(y)$ is equivalent to $F(1) \vee F(2) \dots \vee F(n)$. $1 \leq y \leq n$

Let us now write down an abstract specification of factlist.

Definition 2.

$\forall n \forall l \{ \text{factlist}(n)=l \leftrightarrow n=0 \& l=\emptyset \vee n \geq 1 \& \forall y \leq n \exists n' [\text{fact}(y)=n' \rightarrow n' \in l] \& \forall n' \exists y [n' \in l \rightarrow \text{fact}(y)=n'] \}$

Obviously, the elements in the list could have any property. In fact, the only information we use about the factorials is that "fact" is a function from element to element. We have

fact: element $\rightarrow$ element

We can now deduce the two programs factlist and FACTLIST respectively. The deductions are identical for the first 7 steps but branch as illustrated in figure 2, when we make assumptions about data structures. The left branch follows from the assumption that the data structure is a simple list and the right branch from the assumption that it is a d-list (see Hansson & Tärnlund [7]). We obtain the following program.

Theorem 2.

factlist(0)= $\emptyset$
 factlist(n+1)=u.l + n $\geq$ 0 & list(l) &
 fact(n+1)=u & factlist(n)=l

In Hansson & Tärnlund [7] we also have a deduction of a FACTLIST-program.

Theorem 3.

FACTLIST(0)=<u,u>
 FACTLIST(n+1)=<u,w> + n $\geq$ 0 & d-list(<u,v,w>) &
 fact(n+1)=v & FACTLIST(n)=<u,v,w>

We recall from section 3 that the deductions of theorems 2 and 3 also give us correctness of these programs.

Our second deduction of the FACTLIST-program that computes a list of factorials in ascending order is particularly interesting since Burstall & Darlington [2] write frankly "Notice, however, that to do factlist in ascending order is not easy with our technique and seems to require an extension of the rules. We are currently investigating the problem".

Moreover, we want to call attention to the dotted line in fig.2 which is supposed to illustrate another deduction of a FACTLIST-program. This deduction makes use of mappings between the data structures simple list and the d-list. We shall take up this deduction in the next section.

4.2 Data Structure Mappings

According to the dotted line in fig.2 a deduction requires a correspondence between the data structure simple list and d-list respectively. This one-to-one correspondence between the element in a stack and a queue is depicted in our next figure.

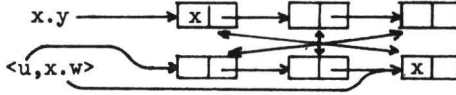


Fig.3 A one-to-one correspondence between a simple list $x.y$ and a d-list $\langle u, x.w \rangle$.

Let us give these correspondences names, domains and ranges in a usual way.

map1: simple list $\rightarrow$ d-list
map2: d-list $\rightarrow$ simple list.

We now have to define these correspondences more precisely in order to carry out a deduction from the factlist program with factorials in descending order to a program with factorials in ascending order.

Definition 3.

$$\forall l \forall q \{ \text{map1}(l)=q \leftrightarrow l=\emptyset \ \& \ \forall u (q=\langle u, u \rangle \vee \exists x \exists y \forall w \exists u [\text{list}(l) \ \& \ l=x.y \ \& \ \text{d-list}(q) \ \& \ q=\langle u, w \rangle \ \& \ \text{map1}(y)=\langle u, x.w \rangle]] }$$

A definition of map2 is straightforward when we have map1.

Definition 4.

$$\forall q \forall l \{ \text{map2}(q)=l \leftrightarrow \text{map1}(l)=q \}$$

We have two functions: factlist and FACTLIST.

factlist: element $\rightarrow$ simple list
FACTLIST: element $\rightarrow$ d-list.

This brings us to the following definitions of these functions.

Definition 5.

$$\forall n \forall q \{ \text{FACTLIST}(n)=q \leftrightarrow \exists l [\text{factlist}(n)=l \ \& \ \text{map1}(l)=q] \}$$

Definition 6.

$$\forall n \forall l \{ \text{factlist}(n)=l \leftrightarrow \exists q [\text{FACTLIST}(n)=q \ \& \ \text{map2}(q)=l] \}$$

These definitions and the following factlist program:

factlist(0)= $\emptyset$
factlist(n+1)=m.l $\leftarrow$ n $\geq$ 0 & fact(n+1)=v & list(l) & factlist(n)=1

give a FACTLIST-program again. Hence, we have obtained theorem 3 twice, but this time by a mapping between data structures (see Hansson & Tärnlund [7]).

We can, of course, deduce the factlist program in theorem 2 in an analogous way from FACTLIST. Consequently, the programs factlist and FACTLIST are equivalent.

4.3 Hypotheses About Alternative Operations on a Data Structure

We shall now treat the idea of keeping the same data structure but making alternative hypotheses about operations on the data structure. In this way, a program is first deduced which inserts a new element into an empty subtree of an ordered binary tree such that the new tree is also ordered. Secondly, a program is deduced which inserts a new element into the root of an ordered binary tree and keeps the new tree ordered.

The important point about these programs is their operation on their data structure. The first program inserts the new element in one of its subtrees until an empty tree is reached. By contrast, the second program inserts its new element into the root of the binary tree and reorganizes the subtrees so that they stay ordered. We owe this program to Keith Clark.

Let us now write down an abstract specification of these programs. We are in fact, a bit strict and make the definition true only for ordered binary trees. We think of an insert relation between individuals u, v and w if and only if u is an ob-tree, v is an element and w is an ob-tree, moreover, each element on w is on u or is identical to v .

Definition 7.

$$\forall u \forall v \forall w \{ \text{insert}(u, v, w) \leftrightarrow \text{ob-tree}(u) \ \& \ \text{element}(v) \ \& \ \text{ob-tree}(w) \ \& \ \forall z [z \in w \leftrightarrow z=v \vee z \in u] \}$$

In Hansson & Tärnlund [7] we first deduce one case of a leaf insert program and secondly one case of a root insert program. The first 14 steps in the deductions are identical until we make a choice between operations on the data structure. We have, thus, the following case of a leaf insert program.

Theorem 4.

$$\text{insert}(t(x, y, z), v, t(x', y, z')) \leftarrow v < y \ \& \ z=z' \ \& \ \text{ob-tree}(t(x, y, z)) \ \& \ \text{insert}(x, v, x')$$

Let us now write down a corresponding case for a root insert program.

Theorem 5.

$$\text{insert}(t(x, y, z), v, t(x', v, t(x'', y, z))) \leftarrow v < y \ \& \ x_2=t(x', v, x'') \ \& \ x=x_1 \ \& \ \text{ob-tree}(t(x, y, z)) \ \& \ \text{insert}(x_1, v, x_2)$$

We depict a relationship between input and output in our root insert program in the next figure. Notice that subtree x is partitioned in two sub-

trees so that x' contains all elements less than v and x'' all elements greater than v .

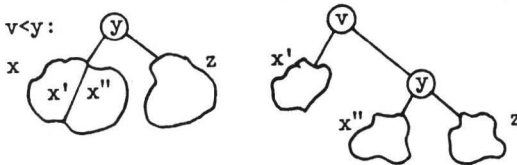


Fig.4 Root insert

In this way we can deduce the entire programs for leaf insert and root insert, respectively.

Theorem 6.

```
insert(t(x,y,z),v,t(x',y,z')) + v<y & z=z' &
    ob-tree(t(x,y,z)) & insert(x,v,x')
insert(t(x,y,z),v,t(x',y,z')) + y<v & x=x' &
    ob-tree(t(x,y,z)) & insert(z,v,z')
insert(t(x,y,z),v,t(x',y,z')) + v=y & x=x' &
    z=z' & ob-tree(t(x,y,z))
insert(∅,v,t(∅,v,∅))
```

Theorem 7.

```
insert(t(x,y,z),v,t(x',v,t(x'',y,z))) + v<y &
    x2=t(x',v,x'') & x=x1 & ob-tree(t(x,y,z)) &
    insert(x1,v,x2)
insert(t(x,y,z),v,t(t(x,y,z'),v,z'')) + y<v &
    z2=t(z',v,z'') & z=z1 & ob-tree(t(x,y,z)) &
    insert(z1,v,z2)
insert(t(x,y,z),v,t(x',v,z')) + v=y & x=x' &
    z=z' & ob-tree(t(x,y,z))
insert(∅,v,t(∅,v,∅))
```

We recall that we obtain a correctness proof of the programs.

5. DEDUCTION FROM A NON-ABSTRACT SPECIFICATION

The abstract character of the specification of factlist in the preceding section gives the advantage of two methods of deducing a program, namely, by choice of data structure or by data structure mapping. Unfortunately, there are quite natural specifications that are less abstract e.g., already bound to a data structure. This situation obviously eliminates the method that provides choices between data structures for deduction of alternative programs. In such a case we can either redefine our specification more abstractly or use the method of data structure mapping, shown in section 4.2. We shall take up the latter method again on a specification of a twist relation. We owe this program to Burstall & Darlington [2].

5.1 Data Structure Mappings for Deducing Alternative Programs

The twist program is a function.

twist: binary tree → binary tree

An example of this program is displayed below.

twist:

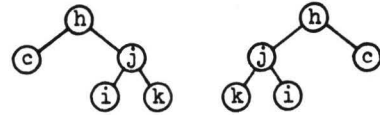


Fig.5 Twist of a binary tree

Let us now consider the problem of substituting this data structure for a "LISP"-tree structure and in this way arrive at a TWIST-program, which "TWISTS" a LISP-tree. Thus we want to deduce a TWIST-program, which is a function from LISP-trees to LISP-trees i.e.,

TWIST: LISP-tree → LISP-tree

An example of the TWIST-function is displayed in the next figure.

TWIST:

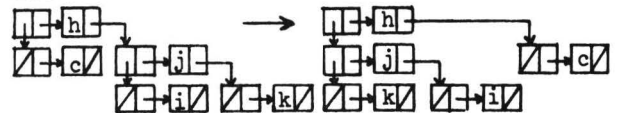


Fig.6 TWIST of a LISP-tree

This TWIST-program, although written in logic and potentially run in PROLOG, is a LISP-program in character, and thus it is natural to ask for a LISP-program. At this point three sub-problems demand solution, a definition of a mapping between a binary tree and a LISP-tree, a definition of TWIST in twist and an axiomatization of relevant LISP-functions. Let us sum up this situation in a figure.

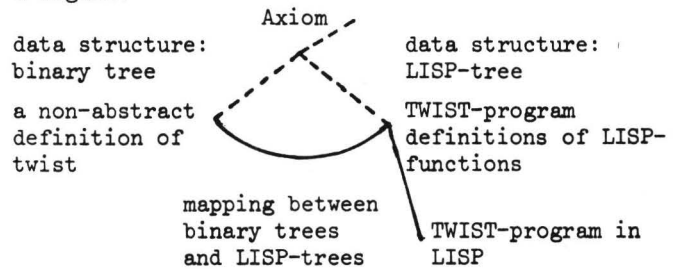


Fig.7 Deduction of a TWIST-program by mappings between data structures

Informally, we need a one-to-one correspondence between the data structures binary tree and LISP-tree.

map1: LISP-tree → binary tree
map2: binary tree → LISP-tree

map2:
map1:

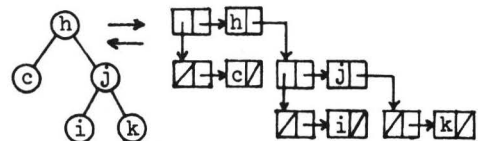


Fig.8 Functions between data structures

Before we can write down this correspondence more precisely we shall have to axiomatize a data structure LISP-tree built up on a LISP-list $\text{cons}(x,y)$, and so we have to introduce the latter data structure.

Axiom 8.

$$\forall x \forall y \exists z \{ \text{cons}(x,y)=z \leftrightarrow (\text{element}(x) \vee \exists u \exists w \text{ cons}(u,w)=y) \}$$

A LISP-tree is a subset of a LISP-list. In fact, it is either nil or constructed, $\text{cons}(x,\text{cons}(u,y))$, where x is a LISP-tree, u an element and y a LISP-tree. We have:

Axiom 9.

$$\forall l \{ \text{LISP-tree}(l) \leftrightarrow l=\text{nil} \vee \exists x \exists y \exists u \{ l=\text{cons}(x,\text{cons}(u,y)) \ \& \ \text{element}(u) \ \& \ \text{LISP-tree}(x) \ \& \ \text{LISP-tree}(y) \} \}$$

As usual we distinguish between an empty list and a constructed list.

Axiom 10.

$$\forall x \forall y \sim \{ \text{cons}(x,y)=\text{nil} \}$$

We can now write down the functions map1 and map2 .

Definition 6.

$$\forall l \forall t \{ \text{map1}(l)=t \leftrightarrow l=\text{nil} \ \& \ t=\emptyset \vee \exists u \exists l_1 \exists l_2 \exists t_1 \exists t_2 \{ l=\text{cons}(l_1,\text{cons}(u,l_2)) \ \& \ \text{LISP-tree}(l) \ \& \ \text{binary-tree}(t) \ \& \ t=\text{cons}(t_1,u,t_2) \ \& \ \text{map1}(l_1)=t_1 \ \& \ \text{map1}(l_2)=t_2 \} \}$$

We get map2 immediately from map1 .

Definition 7.

$$\forall l \forall t \{ \text{map2}(t)=l \leftrightarrow \text{map1}(l)=t \}$$

Let us now specify twist on binary trees, in fact, the definition is quite natural although it is not abstract enough to provide a deduction of TWIST , without the mappings map1 and map2 .

Definition 8.

$$\forall t_1 \forall t_2 \{ \text{twist}(t_1)=t_2 \leftrightarrow t_1=\emptyset \ \& \ t_2=\emptyset \vee \exists u \exists t_{11} \exists t_{12} \exists t_{21} \exists t_{22} \{ t_1=\text{cons}(t_{11},u,t_{12}) \ \& \ t_2=\text{cons}(t_{22},u,t_{21}) \ \& \ \text{binary-tree}(t_1) \ \& \ \text{binary-tree}(t_2) \ \& \ \text{twist}(t_{11})=t_{21} \ \& \ \text{twist}(t_{12})=t_{22} \} \}$$

We can now define TWIST by three mappings. First, from a LISP-tree l_1 to a binary tree t_1 .

$\text{map1}: l_1 \rightarrow t_1$

Secondly, from a binary tree t_1 to a (twisted) binary tree t_2 .

$\text{twist}: t_1 \rightarrow t_2$

Finally, from a binary tree t_2 to a LISP-tree l_2 .

$\text{map2}: t_2 \rightarrow l_2$

We are now ready to define TWIST precisely.

Definition 9.

$$\forall l_1 \forall l_2 \{ \text{TWIST}(l_1)=l_2 \leftrightarrow \exists t_1 \exists t_2 \{ \text{map1}(l_1)=t_1 \ \& \ \text{twist}(t_1)=t_2 \ \& \ \text{map2}(t_2)=l_2 \} \}$$

We still have to deduce a TWIST -program which is independent of twist and the functions map1 and map2 . We can deduce the following program that "TWISTS" a LISP-tree, without any mapping function (see Hansson & Tärnlund [7]).

Theorem 8.

$$\text{TWIST}(\text{nil})=\text{nil}$$

$$\text{TWIST}(\text{cons}(x,\text{cons}(y,z)))=\text{cons}(z',\text{cons}(y,x')) \leftarrow \text{TWIST}(x)=x' \ \& \ \text{TWIST}(z)=z'$$

Moreover, our inductive proof shows that the TWIST -program is total, i.e.,

Lemma 1.

$$\forall w \exists u \{ \text{LISP-tree}(w) \rightarrow \text{TWIST}(w)=u \}$$

Before proceeding to a derivation of a LISP-program let us pause a moment for a comment on the deductions. Although they are made by hand several of them have also been deduced in Weyrauch's FOL system [22]. For example see appendix 1 in Hansson & Tärnlund [7] for a deduction in FOL of the TWIST -program in theorem 8. It is of course a natural development to get these deductions automatically. However, FOL is useful for systematic developments of programs, and in addition it checks the inference and ensures that we arrive at correct programs.

5.2 Deduction of a LISP-program

It is a natural development within our calculus to go a little further and deduce a LISP-program that twists a tree. In order to carry out this program we need an axiomatization of the LISP functions car and cdr .

Axiom 11.

$$\forall x \forall y \{ \text{car}(x)=y \leftrightarrow \exists w \text{ cons}(y,w)=x \}$$

Axiom 12.

$$\forall x \forall y \{ \text{cdr}(x)=y \leftrightarrow \exists w \text{ cons}(w,y)=x \}$$

We shall make use of a LISP-syntax containing "if ... then ... else ..." as proposed by McCarthy [13]. So we have to define this form.

Definition 10.

$$\forall x \forall y \{ B(\text{if } A \text{ then } x \text{ else } y) \leftrightarrow (A \rightarrow B(x)) \ \& \ (\sim A \rightarrow B(y)) \}$$

In this way we do not extend our natural deduction system which we might have done and thus introduce an inference rule for "if ... then ... else ..." introduction and elimination respectively (see Manna [11]). We get the following LISP-program for TWIST (see Hansson & Tärnlund [7]).

Theorem 9.

```

TWIST(w)= if w=nil then nil else
           cons(TWIST(cdr(cdr(w))),cons(car(cdr(w)),
                                         TWIST(car(w))))

```

It is worthwhile noting that our method laid down in this paper also applies to arbitrary programming languages provided their programming constructs are axiomatized like axioms 11 and 12. This axiomatic method seems to be a new approach to program verification.

Unfortunately, no other programming language than pure LISP, except logic itself, of course, has been given a treatment that is rigorous and precise enough for a programming calculus (see Boyer & Moore [1], Cartwright [3] and McCarthy [13]). In particular, the specification of PASCAL (see Hoare & Wirth [9]) does not suffice. It is obviously undesirable to restrict the generality of programming languages so they do not fit into a programming calculus. Therefore, developments of new programming languages should demand solution of two problems; design of the language and axiomatization of the language constructs, for example, as outlined above.

6. CONCLUSION

A methodology for reasoning about programs and for systematic development of programs is of great interest. In this connection we have employed a deductive methodology and obtained a calculus for programming without a detour for a formalization of the programming language. Thus, it is an important point that our language for data structures and program specifications is first order predicate logic, a subset of which is our programming language. This is of great advantage when we deduce programs. Unfortunately, we have not, for reasons of space, displayed any deductions. In fact, they are all examples of hypothetical reasoning formalized in a natural deduction system. By contrast, a program is executed by a PROLOG-like system. Hence, the calculus consists of two distinct logical systems. We have focused on a methodology for systematic development of programs, but it should be noted that the deductive methodology ensures that a derived program is correct. Consequently, we also obtain a methodology for proving that programs are correct.

REFERENCES

- [1] Boyer R. & Moore J.S., Proving Theorems about LISP Functions, JACM vol 22 no 1, 1975
- [2] Burstall R.M & Darlington J., A Transformation System for Developing Recursive Programs, JACM vol 24 no 1, 1977
- [3] Cartwright R.S., A Practical Formal Semantic Definition for Typed LISP, Computer Science Dept. Stanford University, 1977

- [4] Church A., Introduction to Mathematical Logic, Princeton, 1956
- [5] Clark K. & Tärnlund S-Å., A First Order Theory of Data and Programs, IFIP-77, Toronto
- [6] Colmerauer A. et al., Un Système de Communication Homme-Machine en Français, Groupe d'intelligence Artificielle U.E.R de Luminy Université d'Aix-Marseille, Luminy, 1972
- [7] Hansson Å. & Tärnlund S-Å., A Natural Programming calculus, Dept. of Computer Science University of Stockholm and The Royal Institute of Technology, 1979
- [8] Hayes P., Computation and Deduction, Proc. MFCS Conference Czechoslovakian Academy of Sciences, 1973
- [9] Hoare C.A.R. & Wirth N., An Axiomatic Definition of the Programming Language PASCAL, Acta Informatica 2, 1973
- [10] Kowalski R., Predicate Logic as Programming Language, Proc. IFIP-74, North-Holland, 1974
- [11] Manna Z., Mathematical Theory of Computation, McGraw-Hill Inc., 1974
- [12] Manna Z. & Waldinger R., Synthesis: Dreams= Programs, Memo AIM-302, AI-lab., Computer Science Dept. Stanford University, 1977
- [13] McCarthy J., Representation of Recursive Programs in First Order Logic, Draft, AI-lab., Computer Science Dept. Stanford University, 1977
- [14] Newell A. & Simon H.A., Human Problem Solving, Englewood Cliffs, N.J. Prentice-Hall, 1972
- [15] Prawitz D., Natural Deduction, A Proof-Theoretical Study, Almqvist & Wiksell, Stockholm, 1965
- [16] Quine W.V., Methods of Logic, Second Edition Routledge & Kegan Paul, London, 1959
- [17] Robinson J.A., A Machine-Oriented Logic Based on the Resolution Principle, JACM vol 12 no 1, 1965
- [18] Roussel P., PROLOG: Manuel de Référence et d'utilisation, Groupe d'intelligence Artificielle, U.E.R de Luminy, Marseille, 1975
- [19] Tärnlund S-Å., Horn Clause Computability, BIT 17, 2, 1977
- [20] Tärnlund S-Å., An Axiomatic Data Base Theory, in Logic and Data Bases (eds.) Gallaire H. and Minker J., Plenum Press N.Y. 1979
- [21] Warren D., Implementing PROLOG - Compiling Predicate Logic Programs, Dept of Artificial Intelligence, Edinburgh University, No 39, 1977
- [22] Weyhrauch R.W. & Filman R.E., An FOL Primer, Stanford AI-lab, Memo AIM-288, Computer Science Dept. Stanford University, 1976