Stanford Artificial Intelligence Laboratory
Memo AIM-267

Computer Science Department
Report No. STAN-CS-7 5-520

September 1975

On the Representation of Data Structures in LCF with Applications to Program Generation

by

Frederich W. von Henke

Research sponsored by

Advanced Research Projects Agency
ARPA Order No. 2494
and
Deutsche Akademische Austauschdienst

COMPUTER SCIENCE DEPARTMENT
Stanford University

Stanford Artificial Intelligence Laboratory
Memo AIM-267

September 1975

Computer Science Department
Report No. STAN-CS-75-520

On the Representation of Data Structures in LCF with Applications to Program Generation

by

Frederich W. von Henke

ABSTRACT

In this paper we discuss techniques of exploiting the obvious relationship between program structure and data structure for program generation. We develop methods of program specification that are derived from a representation of recursive data structures in the Logic for Computable Functions (LCF). As a step towards a formal problem specification language we define definitional extensions of LCF. These include a calculus for (computable) homogeneous sets and restricted quantification. Concepts that are obtained by interpreting data types as algebras are used to derive function definition schemes from an LCF term representing a data structure; they also lead to techniques for the simplification of expressions in the extended language. The specification methods are illustrated with a detailed example.

Authors's present address: Gesellschaft fuer Mathematik und Datenverarbeitung Bonn, 5205 St.Augustin, Schloss Birlinghoven, West Germany

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract DAHC 15-73-C-0435 and the Deutsche Akademische Austauschdienst. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Stanford University, ARPA, Deutsche Akademische Austauschdtenst, or the U.S. Government.

Con tents

1. Introduction

In this paper we are concerned with the use of data structures in generating correct programs from formal problem statements.

Present experimental systems for automatic program synthesis (see [BuL], [MaW] for recent work) are based on a rather large amount of knowledge in the form of individual axioms and problem solving methods. At each step in the 'synthesis process the system has to search for an applicable piece of knowledge in the data base. One of the main problems is the automatic construction of iterative loops or recursive calls. However, it can be observed that the structure of the data is reflected more or less in the structure of any program operating on them, both in the analysis of subcases and (iterative or recursive) loops. In fact, if a recursion or iteration is possible (and reasonable) at all it is because of a corresponding data structure. So it is safe to say that the generation of a program is always guided by an underlying domain structure, Thus, by "strengthening" the guide lines we can avoid the system having to "retrieve" anew the underlying structure each time it is synthesiiing a program. Organizing the knowledge about the data domain and representing it in such a way that it directly assists a system in constructing a program can possibly eliminate some complicated problem solving processes,

In the case of recursive data types the relationship between program structure and data structure is particularly obvious. For this kind of data types the Logic for Computable Functions (LCF) [Mi1, Mi2, WM J provides a natural, basis for reasoning about program generation, since both the problem and the prospective structure can be expressed in the same formal system. Obviously, the crucial point is to find an appropriate representation of the data structure. A large portion of this paper is devoted to this problem; it attempts to develop a sufficient mathematical framework for dealing with abstract data types within LCF. Based on this theory methods of function specification are investigated that are directly derivable from the data structure representation and do not require general problem solving methods. They include extensions of the term language of LCF, in particular a calculus for (a restricted kind of) sets and restricted quantification, and certain "definition schemes"; both kinds are based, on concepts obtained by interpreting, data types as algebras.

The definition techniques are meant to be a step towards a "problem specification language" that allows easy and concise definition of functions on a level of abstraction that is close to the intuitive conception of the user. This approach to program specification bears a resemblance with what has been called "very high level" or "non-procedural" programming languages. Indeed, programming language features similar to some of the constructions to be discussed here have been proposed elsewhere (e.g. [Ea]) and are available in SETL. However, we are not dealing with a programming language, but a formal system that permits formal reasoning. Emphasis is given to interpreting the added constructions in terms of LCF in order to make feasible meaning preserving transformations of ex presssions. Only the fact that every LCF term also has an interpretation as a computation rule for the function denoted by it, allows us to regard it as a kind of program.

The following section provides the logical and mathematical framework as needed in the subsequent sections. It gives a short overview of the type free version of LCF and the mathematical theory of subdomains. Section 3 discusses the axiomatization of abstract data types, their representation in LCF, and the interpretation of types as heterogeneous algebras. Section 4 is devoted to introducing elements of a specification language, which include (computable) sets, set operations and bounded quantification. The algebraic concepts of section 3 lead to methods for defining and simplifying functions *over* data types. In section 5, the definition methods are demonstrated in an example that is based on the data types of LCF terms and is taken from a LCF implementation. Finally, possible directions of future work are indicated in the concluding section.

The paper is intended to be essentially self-contained. The letters "T.P." that can often be found instead of a proof are meant to indicate that a prove has been generated by means of the interactive theorem prover for LCF. The amount of user interaction required to generate a proof is not indicated; in general, the proofs for simple lemmas can be generated fully automatically. The automatic theorem prover component of the system employed for proof generation will be described in detail in a forthcoming paper [He].

2. The type-free Logic for Computable Functions

The Logic for Computable Functions (LCF) was invented by D. Scott (unpublished) and, in a modified form, mechanized by R. Milner [Mi1, Mi2]. Using this interactive proving system the logic has subsequently been applied to various problems in the Mathematical Theory of Computation: schematology, formalization of syntax and semantics of programming languages, proving properties of programs and the correctness of interpreters and compilers (cf. [AAW, N2] for more recent work on PASCAL and LISP and comprehensive references). In these experiments LCF proved very useful for formalizing and proving problems involving (possibly partial) recursive functions,

In the following the reader is assumed to be, familiar at least with the basics of LCF. For the sake of self-containment, a syntax of the language is given in appendix A.1.1.

2.1 Type-f ree LCF

In this subsection the type free version of LCF (or tfLCF for short) is described briefly as needed for the further development. This version of the logic was developed by D. Scott, R. Milner and R. Weyhrauch [unpublished notes). Most-of the material and the ideas presented here is essentially due to them; part of it can also be found in [Sc2].

Essentially, tfLCF axiomatizes one of Scott's models for the λ-calculus [Sc1]: the domain I which is constructed over the 4-element lattice T of truth values:

$$\begin{tikzcd}
 & T \arrow[dl, dash] \arrow[dr, dash] & \\
tt \arrow[dr, dash] & & ff \arrow[dl, dash] \\
 & A &
\end{tikzcd}$$

The main characteristic of the domain I is that it is isomorphic to its domain of continuous functions; thus, each element of I can also be regarded as a function from I to I,

The language of the logic itself is essentially the same as for the typed version, (see appendix A.1.1), with two exceptions:

(a) the restrictions for building expressions that result from the types are abolished;
(b) besides the 4 truth values, the language includes constants I for the "universe", i.e. the domain of the model, and T for the domain of truth values.

The main problem in extending the semantics of expressions to the type free case is defining the meaning of the conditional $p \supset q,r$ for *any* term p. This is done by mapping the elements of I onto the truth values (this will be made clearer in the following subsection). The meaning of $T \supset x,y$ is not further specified except that $T \supset x,x$ • x. However, it turns out that it can be taken as the join of x and y (see below).

For details about axioms and inference rules of the logic, see appendix A.1.2.

The element ⊥ is called ***undefined,*** and the element ⊤ is called ***overdefined;*** all other elements are called *defined.* A predicate ∂ can be defined in I such that

$$\partial(x) \equiv tt \ \ iff \ \text{"x is defined,"} \qquad \partial(\bot) \equiv \bot, \qquad \partial(\top) \equiv \top$$

i.e., ∂ yields the distinction between defined and non-defined elements in I. 6 is definable in the logic by a mapping onto the S-element lattice $\{\bot, tt, \top\}$. The definition depends on the fact that the truth values are isolated elements in the lattice I. For details see appendix A.2.1.

A function f is called strict if it returns ⊥ or ⊤ whenever the argument is A or ⊤ resp,, that is if the following wff is true of f:

$$\text{v x. } (\partial(x) \supset \partial(f(x)), tt) \equiv \partial(f(x))$$

f is called i-strict if $f(\bot) \equiv \bot$, and T-strict, if $f(\top) \equiv \top$. f is called total if it never returns A. or ⊤ for a defined argument, i.e., if

$$\partial(x) \Rightarrow \partial(f(x)) \equiv tt$$

holds. Thus, if a function f is strict and total then $\partial(x) \equiv \partial(f(x))$.

Any function f can be made into a strict one by first applying ∂ to the argument: For

$$f' := [\lambda x. \partial(x) \supset f(x), A]$$

we obviously have

$$f'(x) \equiv \begin{array}{ll} \top & \text{if } x \equiv \top \\ A & \text{if } x \equiv \bot \\ f(x) & \text{otherwise} \end{array}$$

In the next section a functional **str** will be defined that turns any function into a strict one.

In the following we define some standard **operators** on 1 that will be used throughout the paper.

0	$:= [Xx\ y\ z. x(y(z))]$	function composition
pair	$:= [\lambda\ x\ y\ z. z \supset x,y]$	ordered pair
π_1	$:= [\lambda\ x. x(tt)]$	projection onto first component
π_2	$:= [\lambda\ x. x(ff)]$	projection on to second component
X	$:= [\lambda\ x\ y\ z. pair(x(z\ tt), y(z\ ff))]$	Cartesian product

+	$:\equiv [\lambda\, a\, b\, x.\, \pi_1(x) \supset pair(tt, a(\pi_2\, x)), pair(ff, b(\pi_2\, x))]$	disjoint union
id	$:\equiv$ [Xx.x] .	identity function
$\sqcup$	$:\equiv$ [Xx y. $\top \supset x,y$]	join
A	$:\equiv$ [Xx y. $(\partial(x) \sqcup \partial(y)) \supset x \supset y, ff$]	logical and
V	$:\equiv$ [Xx y. $(\partial(x)$ u $\partial(y)) \supset$ x $\supset tt, y$]	logical or
.	$:\equiv$ [Xx, x $\supset ff, tt$]	negation

For $pair(x,y)$ we also use the notation $\langle x,y\rangle$.

The standard properties of these functions are easily derivable; for example,

Vx $y. \pi_1(\langle x,y\rangle)$ $\bullet$ x Vx $y. \pi_2(\langle x,y\rangle) \equiv$ Y

Milner and Weyhrauch have shown that $\sqcup$ has all the properties of the join operation in a lattice, also with respect to the partial order defined by $\sqsubseteq$. In particular, $tt \sqcup ff \equiv \top$ and $x \sqcup \top \equiv \top$ for all $x \in I$.

A *strict conditional* $:\supset$, i.e. $\top :\supset x,y \equiv \top$ for all x and y , is definable in terms of the normal conditional $\supset$:

$:\supset \;:\equiv \lambda z$ x $y.$ z $\supset (z \supset$ x, T), $(z \supset \top,$ y)

Since the normal conditional will not be used in this paper except in the join operation, we will henceforth use the character $\supset$ to denote the strict conditional.

The propositional connectives are strict in all arguments; they extend the standard functions (for two-valued logic) to four truth values in such a way that the standard relationships like $x \vee y \equiv .(\neg x$ A $\neg y)$ still hold.

2.2 Retracts, Domains, Types

The typefree logic essentially axiomatizes the "universal" domain 1. However, one would like to talk also about domains other than I, like "lists" or "integers." It turns out that they can be "embedded" into the universal domain; there are subdomains of I that correspond to those particular domains in a sense to be made precise in the following section. As Scott [Sc2] has shown, I is so rich in subdomains that one can find a corresponding subdomain for all those domains or "data types" computer scientists are normally interested in.

The standard way of defining a subdomain is by using retracts. A retract is an idempotent function, i.e., an $f \in I$ with $f \circ f \equiv f$. The idempotency property implies that all elements in the range of a retract f remain unchanged, i.e. the range of f is exactly its set of fixed points. subdomain D_f of I (the range of f). This domain D_f can be shown to be a complete lattice. In the remainder of this paper, the term "domain" always nieans "subdomain of I as defined by a retract." Very often the domain and the function (the retract) defining it will be confused by using the same notation for both; however, from the context it will be clear what exactly is meant. For emphasis, we will say *retraction* if we mean the *function* in particular.

The category of retracts

It may be helpful to look at retracts from a categorical point of view. The retracts of I form a category R in the following way:

- The objects of R are the retractions in I.
- A functions $f \in I$ is a morphism from the retraction r to the retraction s iff $f \circ r \equiv s$.
- Composition of morphisms in R is just composition of functions in I.

Obviously, R is a category. Note that two functions f and g in I will be identical morphisms in R if they agree on their source domain ("source retract"), i.e. if $f \circ r \equiv g \circ r$. An identity on a retract r is a function F with $F \circ r \equiv r$. We write Id_r for the identity on r.

Let r, s be retracts. D_r is called **subdomain** of D_s iff $s \circ r \equiv r$, i.e. iff the fixed points of r are also fixed points of s. D_r is called retract of D_s iff $s \circ r \equiv r$ and $r \circ s \equiv r$.

- A particular retract is the truth value domain **T**. Trivially, the universal domain I is also a retract. However, the property of being a retract cannot be proved; the corresponding retractions

$$T \equiv [\lambda x.\, x \supset tt, ff]$$

and

$$I \equiv [\alpha J.\, T \cup (J \to J)]$$

are rather part of the axiomatization of tfLCF. Obviously, "retract" and "retract of I" mean the same thing.

It should be noted that R is *not* the category of those subdomains of I that are defined by retracts; different retractions can define the same domain but will be different objects in R. For example, the retractions T and istrue (see appendix) both define the domain $\{\perp, tt, ff, \top\}$ but are completely different functions. However, T and istrue are *isomorphic* in the category R. Incidentally, if two domains are isomorphic one of them need not be a retract of the other: for instance, it is

$$T \circ istrue \equiv istruo \qquad \text{and} \qquad istrue \circ T \equiv T$$

i.e. T and istruo are subdomains of each other, but

$$istrue \circ T \not\equiv istrue \qquad \text{and} \qquad T \circ istrue \not\equiv T$$

Thus neither of **T** and **istrue** is a retract of the other. **This** discussion shows how retractions that define the same domain are related in the category R:

Corollary 2.1: *For retractions* r *and* s*, if* $D_r \equiv D_s$ *then* r *and* s *are isomorphic, that is, **from** a structural **point** of view, they cannot 'be **distiguished**.*

The category R has many useful closure properties we are going to exploit.

Lemma 2.2: R is *closed under* $+, \times$ *and* $\rightarrow$, i.e. *if* a *and* b are *retracts, then so are* $a+b$, ax b, *and* $a \rightarrow b$.

Proof: by T.P.

Lemma 2.3: R *is cartesiati-closed.*

Proof; We have to prove that $[r \rightarrow [s \rightarrow t]]$ and- $[r \times s \rightarrow t]$ are isomorphic in R for **any retracts** r, s and t. Let F, G be defined by

$$F \equiv [\lambda\ f\ x.\ f(\pi_1\ x)(\pi_2\ x)\ ]$$
$$G \equiv [\lambda\ g\ r\ s.\ g(\langle r,s \rangle)]$$

The T.P. proofs for

$$G \circ F \equiv Id_{[R \rightarrow [S \rightarrow P]]}.$$

and

$$F \circ G \equiv Id_{[R \cdot S\ \ \cdot B\ \ P]}$$

are almost straightforward.

Lemma 2.3 is the basis for what is commonly called "currying". It allows to restrict attention to monadic functions.

Let the function **str** be defined by

$$str := [\lambda f\ x.\ \partial(x) \supset f(x), \perp]$$

By T.P. we can show that str 'turns any function into one that is strict (with respect to the first **argument)** and that it is a retract. This shows that the set of strict functions is a proper subdomain of I.

A domain is called flat if it contains, besides $\perp$ and $\top$, only **pairwise** incomparable elements, **For** flat domains there is a computable equality relation "$=$" with:

$$\partial(x) \equiv tt,\quad \partial(y) \equiv tt,\quad x \equiv y \qquad \vdash\ x{=}y \equiv tt$$
$$x{=}y \equiv tt \qquad \vdash\ x \equiv y,\quad \partial(x) \equiv tt,\quad \partial(y) \equiv tt$$

In many cases it is very convenient to use the (computable) equality instead of the equivalence $\equiv$ 'since it may appear inside a term and thus gives greater expressive power.

Lemma 2.4: If $F \equiv [\alpha f.\ [Xc.\ t(f(c))]]$ *and* $G \equiv [Xc.\ [\alpha g.\ t(g)$ J]] , *where* t *is any term, then* $F \equiv G$.

Proof: T.P.

Essentially, the lemma means that constant parameters can be bound "globally", i.e. they need not be passed on with every call. The lemma will be used quite often in the remainder of the paper without being referred to explicitly.

3. Abstract Data Types in the Type Free Logic

In this section we introduce data types and discuss the representation of data structures in tfLCF. We investigate <properties of data types by looking at them from a more algebraic point of view, which allows us to derive various function definition schemes. The basic function definition method is illustrated in an example dealing with the translation of arithmetical expressions from infix to postfix form.

What is intuitively meant by the notion *abstract data type?* There is a common understanding that, in programming, a data type is not just a set, but also comprises information about the structure of the elements and how to construct them and to operate on them. This can be done in an *abstract* way, i.e. the only information available is the set of primitive operations (constructors, selectors, recognizers) and relationships between them; it does not matter what the elements of the type look like and how the primitive operations are implemented. In the context of a formal calculus the relationships between the primitive operations are expressed by axioms.

The presentation concentrates on generic recursive types; however, in subsection 3.4 extensions to non-free types are discussed.

3.1 Data Type Definitions

We start with discussing *free data types.* The type system will be extended later to comprise a wider class *of* types, A *type definition* is made by listing alternative subtypes. A *subtype* is either a constant or a composed type. *Composed data types* are defined best by their *abstract syntax* [Mc], using *constructors, selectors* and *recognirers* to describe the structure of the type. In a more formal BNF-like notation (using "constr" for constructor, "sel" for selector, and "dt" for data type):

```
<type_def>     t  <type_name> := <subtype> { | <subtype> }*
<subtype>      t <constant> | <comptype>
<comptype>     t <constr> ( <sel1>:<dt1>,.....,<seln>:<dtn> )
<constant>     ← <identifier>
```

with the restriction that the names of all constructors in a type definition and all selectors in one composed type have to be distinct. A data type definition may be recursive, that is, any of the dt_i in a subtype may be the name of the type to be defined. Also mutually. recursive data type definitions are permitted.

For example, the data type "sequence (linear list) of atoms" can be defined using this formalism by

```
Seq   := emptyseq | mkseq(hd:atom, tl:Seq).
```

Strictly speaking, this data type definition is a *type scheme,* that is,

```
seq     := emptyseq | mkseq(hd:dtype, tl:seq)
```

defines a type "sequence of elements of type dtypo" for *any* data type **dtype**. This will be made more precise in the following subsection. Beside **seq** we will use other standard data types (type schemes) like *binary trees, natural numbers,* and *pairs*, defined by

```
bintree := mkbt(sub:dtype) | comp(fir:bintree, sec:bintree)
nnum    := zero | suc(nn:nnum)
dpair   := mkpair(fir:dtype_1, sec:dtype_2)
```

3.2 Representing Data Structures in tfLCF

In section 2 it was explained that retracts can be regarded as the "types" in the type-free logic, The data types are now to be represented in LCF in such a way that the resulting terms are retractions.

What exactly is implied by a data type definition? Intuitively, a data type should have the following properties:

a) A data type is the disjoint union of subtypes. A subtype is either a constant or a composed subtype. For each subtype there is a predicate (characteristic function) which will be named "is_const" or "is_<constr>" resp. These *recognizers* permit to decide membership in one of the subtypes relative to the whole data type.

b) Each constructor is a one-to-one function; in particular, the corresponding selector functions allow to "retrieve" the respective arguments of a constructor.

c) A subtype has to be embedded explicitly into *the* type by a constructor function. For example, "atoms" are not lists unless they are "converted" into lists. This helps us to keep all data types disjoint.

These statements can be expressed more precisely in terms of LCF axioms.

Definition 3.1 (Axioms for generic data types):

The data type definition

$$\text{type} := \text{constant}_1 | \dots | \text{constant}_m | \text{comptype}_1 | \dots | \text{comptype}_n$$

with

$$\text{comptype}_k := \text{comp}_k(\text{sel}_{k1}:\text{dt}_{k1}, \dots, \text{sel}_{kjk}:\text{dt}_{kjk}) \quad \text{for } k=1,\dots,n$$

is considered to correspond to the fixed point equation

$$(1)\quad \text{typo} = [\alpha F.[\lambda x.\ \begin{aligned}&\text{is\_constant}_1(x) \supset x, \dots, \text{is\_constant}_m(x) \supset x,\\ &\text{is\_comp}_1(x) \supset \text{comp}_1(\text{dt}_{11}(\text{sel}_{11}(x)), \dots, \text{dt}_{1j1}(\text{sel}_{1j1}(x))), \dots,\\ &\text{is\_comp}_n(x) \supset \text{comp}_n(\text{dt}_{n1}(\text{sel}_{n1}(x)), \dots, \text{dt}_{njn}(\text{sel}_{njn}(x))),\\ &\perp\ ]]\end{aligned}$$

where $dt'_{kj}=F$ if $dt_{kj}=type$ and' $dt'_{kj}=dt_{kj}$ otherwise, and for $i,j=1,..,m; k,l=1,..,n$

(2) $\partial(\text{constant}_i) \equiv tt$
$\text{is\_constant}_i(\text{constant}_i) \equiv tt$

(3) $\partial(\text{comp}_k(dt_{k1}(x_1),..,dt_{kjk}(x_{jk}))) \equiv \partial(dt_{k1}(x_1)) \wedge .. \wedge \partial(dt_{kjk}(x_{jk}))$
$\partial \circ \text{comp}_i \equiv \text{is\_comp}_i \circ \text{comp}_i$

(4) $\forall x . \text{is\_comp}_k(x) \Rightarrow \text{is\_comp}_l(x) \equiv ff$
$\forall x . \text{is\_const}_i(x) \Rightarrow \text{is\_const}_j(x) \equiv ff$
$\forall x . \text{is\_const}_i(x) \Rightarrow \text{is\_comp}_k(x) \equiv ff$

(5) $\partial \circ \text{is\_comp}_k \equiv \partial \circ \text{is\_comp}_l$ for $k \neq l$
$\partial \circ \text{is\_const}_i \equiv \partial \circ \text{is\_const}_j$ for $i \neq j$
$\partial \circ \text{is\_const}_i \equiv \partial \circ \text{is\_comp}_k$

(6) $\partial(\text{comp}_i(x_1,...,x_{ni})) \Rightarrow \text{sel}_{ir}(\text{comp}_i(x_1,...,x_{ni})) \equiv x_r$ for $r=1,..,n_i$

Axiom (1) is a mere transcription of the type definition. It contains the basic information about the type structure; therefore, it will be called the *characterizing function* of the type. The goal is to prove that it is a retract. However, this cannot be done without further specifying the primitives occurring in (1) by adding axioms expressing the statements a)-c). They make sure that the recognizers for the subtypes are complementary (axiom 4) and that they are defined exactly for the elements of the type (axiom 5). Axioms no. 6 state the *generic* nature of the type (This is equivalent to saying that constructor and selector functions are essentially tupling and projections). All constructors are assumed to be strict in each argument and total for arguments of correct types (axioms 3).

Example: The data type definition for (homogeneous) sequences

Seq ::= emptyseq | mkseq(hd:Atom, tl:Seq)

will generate the axioms

(S1) $\text{seq} \equiv [\alpha S. [\lambda x. \text{is\_emptyseq}(x) \supset \text{emptyseq},$
$\text{is\_mkseq}(x) \supset \text{mkseq}(\text{atom}(\text{hd}(x)), S(\text{tl}(x))),$
$\perp]]$

where atom is the characterizing function for the data type Atom,

(S2) $\partial(\text{emptyseq}) \equiv tt$
(S3) $\text{is\_emptyseq}(\text{emptyseq}) \equiv tt$
(S4) $\forall x\, y. \partial(\text{mkseq}(\text{atom}(x), \text{seq}(y))) \equiv \partial(\text{atom}(x)) \wedge \partial(\text{seq}(y))$
(S5) $\partial \circ \text{mkseq} \equiv \text{is\_mkseq} \circ \text{mkseq}$
(S6) $\text{is\_emptyseq} \equiv \neg \circ \text{is\_mkseq}$
(S7) $\forall x\, y. \partial(\text{mkseq}(x,y)) \Rightarrow \text{hd}(\text{mkseq}(x,y)) \equiv x$
(S8) $\forall x\, y. \partial(\text{mkseq}(x,y)) \Rightarrow \text{tl}(\text{mkseq}(x,y)) \equiv y$

Any type t_i that occurs in the definition of another type t_j is considered a *base type* for t_j. The notion "base type" does not imply "basic" or "simpler"; on the contrary, since mutual dependence of data types is permitted, t_j itself may be a base type for t_i (hence the relation "is-base-type-of" is only a quasi-order). If data types are mutually dependent the corresponding characterizing functions form a system of mutually recursive functions. Those base types that do not depend on the type to be defined are called *generating types* (in fact, they generate the type in an algebraic sense; see the following sub-section).

At this point it has to be clarified what it means for a characteristic function to be a retraction, The logical type of typo is (type→type). In order to make it a function in I the primitives (constants, constructors and selectors etc.) have to be specified as elements of I. This amounts to defining a *model* of (the axioms describing) the data type in I. However, the retraction property can be proved just from the axioms; more precisely, what can be proved is that type is an "abstract retraction", meaning that 'every model is a retraction. All the models will be isomorphic retractions (in the categorical sense), Thus the, abstract retraction represents an *equivalence* class of objects in R.

It should be noted that the standard representation of data types in a LCF-like language is by "domain equations" (involving + and × ; see [Sc2]). For example, the data type Seq of sequences of atoms is completely specified by the least fixed point of the equation

(i) Seq • ● mptysoq :+ (Atom :× Seq)

(where :+ and :× are strict versions of + and ×) given a representation of the type Atom and the constant emptyseq. However, we do not follow this line. The syntax of data type definitions, as given in the preceding section, involves constructors and selectors; they are the primitives for defining functions operating on the data type. But they do not appear in (i); in fact, (i) is an *implementation* of the data type Seq (by functions in I) rather than an abstract definition; the primitives are hidden in the construction of sum and product. In order to keep the previous higher level of abstraction the primitives have to be axiomatized, as has been done for special types in [Ne1]. Although an axiomatization as in definition 3.1 is by far less elegant than a definition like (i) it is more appropriate for the purpose of program specification.

Incidentally, this discussion shows that every (generic) data type has the following standard model: The data type definition

$$\text{type} := \text{constant}_1 \mid \ldots \mid \text{constant}, \mid \text{comptype}_1 \mid \ldots \mid \text{comptype},$$

with

$$\text{comptype}_k := \text{comp}_k(\text{sel}_{k1}:\text{dt}_{k1}, \ldots, \text{sel}_{kjk}:\text{dt}_{kjk}) \quad \text{for } k=1,\ldots,n$$

is simply translated into

$$\text{type} := \text{tt} :+ \ldots :+ \text{tt} :+ \text{comp}_1 :+ \ldots :+ \text{comp}_n$$

with

$$\text{comp}_k := (\text{dt}_{k1} :\times \; . \; , :\times \text{dt}_{kjk}) \quad \text{for } k=1,\ldots,n$$

It is easy to figure out what the primitives look like, and the proof that the axioms of definition 3.1 hold is straightforward.

Lemma 3.2: type *is strict.*

Proof: Simple consequence of the fact that all functions involved are strict (by definition).

Theorem 3.3: **type** *is a retract* if *all base types of* type *are retracts.*

Proof: See appendix **A.3.**

From the characterizing function of a **type** we are going to derive a variety of function definition schemes and functions. **In** particular, *a type predicate* (characteristic function) is-type **can be derived which** yields **tt** exactly for the defined elements of the **type,** i.e. which satisfies:

$\forall x.$ **is_type(x) ⇒** typo(x) ≡ x
$\forall x.$ **is_type(x) ⇒ ∂(x) ≡ tt**

and

type(X) **≡ x, ∂(x) ≡ tt |- is_type(x) ≡ tt**

This predicate will be discussed in greater detail in the following subsection.

For a recursive data type we can derive the standard structural induction rule from the characterizing function (the retraction).

Theorem 3.4 (Structural Induction):
For a recursive data ***type*** typo *defined by*

type := constant $_1$ | . . . | constant, | **comptype$_1$** | . . . | comptype,

with

comptype$_k$:= comp$_k$(sel$_{k1}$:dt$_{k1}$, . . . ,sel$_{kj_k}$:dt$_{kj_k}$) for **k=1,...,n**

there *is an induction rule that allows to conclude*

Q |- $\forall x.$ **is_type(x) ⇒ P(x)**

(where ***the conclusion is*** *meant to be a wff involving x each of whose* ***awff's*** *is* ***prefixed by*** $\forall x.$ **is_type(x)**. . *)* from *the* ***following*** *antecedents:*

(a) For each constant **constant$_i$ (i=1,..,m)**

Q |- P(constant$_i$)

(b) For each composed subtype $\mathrm{comptype}_k$ $(k=1,\dots,n)$

$$Q, P(y_{kjp}), \dots \; \partial(\mathrm{comp}_k(\dots)) \equiv tt \;\; \vdash \;\; P(\mathrm{comp}_k(\dots))$$

with an antecedent $P(y_{kjp})$ *for each recursion arguments, i.e. for those arguments of* comp_k *with* $dt_{kj} \equiv \mathrm{type}$.

Proof; see appendix A.3.

For example, the induction rule corresponding to the characterizing function **seq** is

$$\frac{P(\mathrm{emptyseq}) \qquad P(y),\; \partial(\mathrm{mkseq}(x,y)) \equiv tt \;\; \vdash \;\; P(\mathrm{mkseq}(x,y))}{\forall x.\; \mathrm{is\_seq}(x) \Rightarrow P(x)}$$

Note that the constructor arguments in the induction step need not be restricted by type predicates (the restriction is implied by the definedness predicate). A discussion of other forms of the induction rule that involve the retraction can be found in appendix A.3.

As mentioned above, the type definition for **seq** is a *type scheme,* defining a data type for any type **dtype.** This means that in the corresponding retraction seq the retract atom can be replaced by any other retract. We therefore can define the functional

$$\begin{aligned} \mathrm{seqof} :\equiv [\lambda\, \mathrm{type}.\, [\alpha S.\, [\lambda x.\; & \mathrm{is\_emptyseq}(x) \supset \mathrm{emptyseq}, \\ & \mathrm{is\_mkseq}(x) \supset \mathrm{mkseq}(\mathrm{type}(\mathrm{hd}(x)), S(\mathrm{tl}(x))), \\ & \bot]] \end{aligned}$$

By theorem 3.3 **seqof(type)** is a retract for *any* retract type. In other word, **seqof** *maps retracts on retracts.* Obviously, *any* generic type construction yields such a mapping on retracts. Properties of these **functionals** will be studied in the following subsection.

3.3 Algebraic Interpretation of Data Types

Interpreting data types in terms of universal algebra helps to clarify certain concepts and properties. As a data type may involve several subtypes-and functions of heterogeneous type the appropriate notion is that of "heterogeneous algebra" (Birkhoff and Lipsom [BI]; see *also* Higgins [Hi]).

Definition 3.5 (Heterogeneous algebra)

A heterogeneous algebra A consists of

- a family $(A_j)_{j\in J}$ of (non-empty) sets; the A_j are called *phyla.*
- a family $(f_k)_{k\in K}$ of operations; for each f_k there is an associated tupel $\mathrm{index}_k = (j_{k1}, \dots, j_{kn_k}; j_{kn_k+1})$ of elements of J.

The index of the operation f_k indicates the phyla from which the arguments of f_k are taken (i.e. the argument types) and the target phylum. nk is the *arity* of f_k (possibly 0). In the present framework, the index is simply the type of the operation.

The triple $(J,K,(\text{index}_k)_{k\in K})$ is called the *signature* of the algebra; it characterizes the basic structure. Algebras are called *similar* if they have the same signature. As the structure of similar algebras is comparable, it is possible to define structure-preserving mappings between them.

Definition 3.6 (Homomorphism)

Let $A = ((A_j)_{j\in J}, (f_k)_{k\in K})$ and $B = ((B_j)_{j\in J}, (g_k)_{k\in K})$ be similar algebras. A homomorphism h from A to B is a family of mappings $(h_j)_{j\in J}$ such that hi maps A_j into B_j and for each $k\in K$

$$h_{j_{nk+1}}(f_k(a_1, \ldots, a_{nk})) = g_k(h_{j_1}(a_1), \ldots, h_{j_{nk}}(a_{nk}))$$

where $\text{index}_k = (j_1, \ldots, j_{nk}; j_{nk+1})$.

As mentioned in the previous section, a data type (or: its domain) forms a complete lattice. So, the appropriate algebraic structure is that of a *heterogeneous lattice-algebra.* Although the lattice structure of the domains considered here is not very interesting - apart from the elements $\perp$ and $\top$ the domains are flat - we have to take it into account by requiring that all functions preserve the lattice structure. However, because of the simple structure it is sufficient to require that all functions involved in the algebraic structure, i.e. the operations, are strict and total. Similarly, all the mappings h_j constituting a homomorphism have to be strict (it is, however, not necessary to assume totality), Henceforth, these assumption will be made throughout the remainder of the paper,

Example. The data type Seq, regarded as a heterogeneous algebra, consists of the two phyla atom and seq and operations

emptyseq:	$0 \to$ seq	(nullary)
mkseq:	atom$\times$seq $\to$ seq.	

The axioms in the previous section indicate that a data type corresponds to an absolutely free (or "generic") algebra which is generated by the constants, the base types and the constructor functions as operations. As it is well-known in algebra, an absolutely free algebra has characterizing universal properties:

(1) *There is (upto isomorphisms) only one absolutely free algebra for given generating base types and operations.*

(2) *Any homomorphism from an absolutely free algebra F into another algebra A of the same type (in the algebraic sense) is determined uniquely by functions mapping the generating sets into A.*

Properties (1) and (2) can be proved in LCF for each (free) data type without relying on an algebraic interpretation. It is these properties we are going to exploit.

In order to define a homomorphism it is sufficient to map the base types into target sets and the constructors onto operations on the target structure. Then, by property (2) there is a unique function that homomorphically extends the base function(s). Due to the fact that in LCF homomorphisms can be "pushed through" conditionals, homomorphic extension is representable by a simple modification of the function characterizing the data type: we have only to replace the base type retracts by the base functions and the constructors by the operations on the target algebra.

To continue our example based on the data type **seq**, we notice that a homomorphism from **seq** into an appropriate algebra is determined completely by

a) a constant that is the image of **emptyseq,**
b) a function that maps the base type atom into the corresponding set, and
c) a binary operation on the target algebra.

In LCF, this is written as the functional

```
Sfun :≡ [λ f const op. [α s. [λx. is_emptyseq(x) ⊃ const,
                                  is_mkseq(x) ⊃ op(f(hd(x)), S(tl(x))),
                                  ⊥ 113
```

Assume R is the **target** structure with phyla R_1 and R_2, c an element of R_2, op : $R_1 \times R_2 \rightarrow R_2$ a binary operation and fun a function from atom to R_1. Then property (2) above yields the following theorem:

Theorem 3.7: F :≡ **Sfun(fun,c,op)** *is the unique homomorphic extension of* fun *with respect to c and* **op,** *i.e., it is' the* ***only*** *homomorphism* *from* **seq** *to* R *with* **F(emptyseq) ≡** c and F ∘ **mkseq ≡** (λx **y.op(fun(x),F(y))].**

The proof is straightforward; it crucially depends on the "freeness" of the type definition (i.e. axioms (S7), (S8)) which is necessary to establish the homomorphism property of F.

A simple example is the type predicate (characteristic function) for **seq**: The function **is_seq: seq→T** with

is_seq(x)≡tt *iff* **seq(x)≡x** *and* **∂(x)≡tt**

is definable simply by extending the type predicate is-atom of the generating base type to a homomorphism into **T**:

is_seq ≡ Sfun(is_atom,tt,∧)

A further property of the homomorphic extension functional Sfun is that it carries over a structural induction rule from the source domain. **E.g** for sequences:

```
P(c)       P(y), ∂(op(x,y ))≡ tt |- P(op(x,y))
----------------------------------------------
            ∀ z. is_seq(z) ⇒ P(F(z))
```

thus permitting induction on (target) domains originally not structured appropriately.

The mathematical content of the discussion on interpreting data types as absolutely free algebras and the homomorphic extension functionals amounts to a well-established fact known from category theory: the correspondence between free constructions (free objects) and representable functors. The pair (Sdom,Sfun) defines a functor from R into a subcategory of R of "suitably structured" retracts, The point is that this correspondence can be established within the framework of LCF. Due to the fact that everything is represented as LCF terms, objects and morphisms as well as functors, it allows to carry out mechanically assisted proofs rather easily. For the time being, theorems like the one mentioned above have to be proved in LCF for each data type separately, although the structure of the proof is always the same. However, there is some hope that formal proofs of general statements about, e.g., all generic data types will be feasible using a metatheory of LCF being developed on the basis of representing the LCF notions as data types (see section 6 for part of the data type definition).

The usefulness of homomorphisms as a structuring principle has been observed elsewhere, in particular in the context of program translation [Mo, MiW]. However, though homomorphic extension is a rather powerful scheme for function definition it is by far not powerful enough. It turns out that properties similar to those proved for homomorphic extension can be shown for a more general class of definition schemes; this will be discussed in section 5.

3.4 Non-generic Data Types

Although the class of generic data types covers many of the structures needed in programming it is not comprehensive enough. Relaxing the restriction to generic structures is tantamount to, in algebraic terms, allowing to add further relations to a type definition. In a way, the generic data types can be regarded as the "context-free types," and adding relations as "introducing context." In the context of this paper it is sensible to consider only relations that are expressible as recursive predicates.

The general method will be discussed by means of an example. Let norep(x) be a predicate on sequences which is true iff x does not contain repetitions of elements (the explicite definition is straightforward). Then the data type norepseq of "sequences without repetitions" is just the restriction of seq by norep. The new type can be represented in the following way: Whenever an element is added to a sequence it is checked first if it already occurs in it, in which case nothing is done. That is, if the constructor mkseq is modified to

mknorepseq :≡ [λx y. norep(mkseq(x,y)) ⊃ mkseq(x,y), y]

then all sequences constructed by mknorepseq have the "no-repetition" property, i.e. they satisfy the predicate norep. This is just another application of the homomorphic extension functional: The range of the function norepseq := Sfun(dtype, emptyseq, mknorepseq)

is exactly the desired subset of sequences without repetitions (it is obvious that only elements of seq are constructed). In other words, norepseq represents *ordered sets* of *elements of type* dtype. Obviously, norepseq is a *retract* of seq; Since norepseq defines a subdomain of seq it is also a retract of I, which means that norepseq makes sense (in the present context) as a data type. This construction for new retracts works at lease in the case where a new type is defined by a restrictive predicate. The full extend of the method, however, needs to be explored further. It is conjected that *any* data type (given a reasonable definition in terms of computability) is representable as a retract of a generic type; this would parallel the fact that, in formal language theory, any recursively enumerable set is the image of context free sets under suitable mappings.

3.5 An Example: Infix to Post-fix Translation

As an example we show how to generate a function that translates arithmetical expressions from infix to postfix notation (the example was suggested by J.Allen). The abstract syntax of the structures is defined by

```
exp    := mktexp(te:term) | mksexp(su1:exp, su2:term)
term   := mkfterm(tf:fact) | mkpterm(pr1:term, pr2:fact)
fact   := mkvfact(fv:var) | mkefact(fe:exp)
```

and

```
post   := mkvpost(pv:var) | mksum(s1:post, s2:post) | mkprod(p1:post, p2:post)
```

which may be thought of as abstraction from the “concrete” infix grammar

```
<exp>  := <exp> '+ <term> | <term>
<term> := <term> 'x <fact> | <fact>
<fact> := <var> | '( <exp> ')
```

and the postfix grammar

```
<post> := <var> | <post> <post) '+ | <post> <post> 'x
```

Now, the problem-is to find a function that translates variables into variables and infix-sums and infix-products into postfix-sums and postfix-products resp. This is a simple example of a homomorphism between heterogeneous algebras. The algebra Exp includes the 4 phyla exp, term, fact and var, the algebra Post the phyla post and var. The homomorphism maps exp, term and fact into the phylum post and var onto var, that is, the homomorphism consists of 4 mappings

id:	var → var
Texp:	exp → post
Tterm:	term → post
Tfact:	fact → post

These mappings have to respect the corresponding algebraic operations

```
mksexp:    exp×term → exp   <-> mksum  : post×post → post
mkpterm:   term×fact → term <-> mkprod: post×post → post
mkvfact:   var → fact       <-> mkvpoct: var → post
```

i.e., they must satisfy equivalences

$$\mathrm{Texp}(\mathrm{mksexp}(x,y)) \equiv \mathrm{mksum}(\mathrm{Texp}(x),\ \mathrm{Tterm}(y))$$

etc. Since the distinction between exp, term and fact disappearb in Post, the "operations" corresponding to mktexp, mkfterm and mkefact are just identities on post. Having established all the algebraic correspondences, homomorphic extension immediately yields the desired functions (slightly simplified):

```
Texp   ≡ [∝E. [λx. is_texp(x) ⊃ Tterm(te(x)),
                   is_sexp(x) ⊃ mksum(E(su$_1$(x)), Tterm(su$_2$(x))),
                   ⊥]]

Tterm  ≡ [∝F. [λx. is_fterm(x) ⊃ Tfact(tf(x)),
                   is_pterm(x) ⊃ mkprod(F(pr$_1$(x)), Tfact(pr$_2$(x))),
                   ⊥]]

Tfact  ≡ [λx. is_vfact(x) ⊃ mkvpost(var(fv(x))),
              is_efact(x) ⊃ Texp(fe(x)),
              ⊥]
```

4. Elements of a Problem Specification Language

This section is devoted to discussing a rudimentary "problem specification language." The language consists of the terms of typed LCF, augmented by certain, constructions that are considered natural or helpful for concise specification of problems or, more precisely, functions over data types. The main extension is a first-order like calculus that enables to talk about sets and quantification in a way consistent with the computational logic. is an extension of the LCF terms in their typed form. Using the definition techniques developed in the preceding section, the added constructions are interpreted as LCF terms which gives them the intended meaning as computation rules or "programs."

4.1 Sets, Set Operations and Quantification

Syntax

Types. The language is typed, i.e. a type is associated with each term. There is a predefined type: T, the domain of truth values. New types can be defined explicitly as data types (see below). For each type t we have a type setof(t) denoting the powerset type "sets of elements of type t". More formally:

Definition 4.1 (Types):

(1) Tisatype.

(2) Data types are types.

(3) If t_i and t_j are types, then $(t_i \rightarrow t_j)$ is a type (the type of functions from t_i to t_j).

(4) If t is a type then setof(t) is a type.

(5) These are all the types.

Types built by (4) are called set *types.* No data type is a set type. Although types are not sets, we use the type name also to denote the set of individuals of that type. There are no equalities between types; different type expressions, in particular different type names, denote different types.

Note the distinction between "types" and "data types". Types are the sorts in the logic, whereas the notion data type is used more in the sense of data types in programming languages which involves certain assumptions about the (internal) structure of the typed objects. By (2) in definition 4.1 data types are assumed to coincide with certain logical types.

Terms, We use the notation s:t to denote a term s of type t. If t is a set type then s:t is called *set* term. All LCF terms (1 - 6) are terms of our language (cf. [Mi 13 and appendix A.1.1). Beside the LCF terms the language includes terms for expressions involving sets and bounded quantification (7 - *9).*

Definition 4.2 (Terms):

(1) The constants ⊥, tt, ff, ⊤ are terms of type T.

(2) Any identifier is a term.
(3) If $s:t_1 \rightarrow t_2$ and $x:t_1$ are terms then $s(x):t_2$ is a term.
(4) If $x:t_1$ is an identifier and $s:t_2$ is a term then $[\lambda x.s(x)]: t_1 \rightarrow t_2$ is a term.
(5) If $p:T, q,r:t$ are terms then $(p \supset q,r):t$ is a term.
(6) If $x:t$ is an identifier and $s:t$ a term then $[\alpha x.s]:t$ is a term.
(7) If $x:t$ is an identifier and $S:setof(t)$ a set term then $(x \epsilon S):T$ is a term.
(8) If $x:t$ is an identifier, $S:setof(t)$ a set term and $P:t \rightarrow T$ a predicate term then $(\forall x \epsilon S.\ P(x)): T$ and $(\exists x \epsilon S.\ P(x)): T$ are truth value terms.
(9) These are all the terms.

As usual parentheses and brackets can be omitted as long as parsing is unambiguous. The notions *awff* and *wff* are used as in typed LCF (see appendix A.1.1).

Note that the use of the sign V for quantification-in (8) cannot be confused with the use of V in abbreviations for $\lambda x.t \equiv \lambda x.s$. The former *always* requires a restricting set whereas the wff-$\forall$ is never restricted.

Semantics
The aim is to interpret the extended typed language in the type free calculus. This is done by showing that every type corresponds to a retract in *tf*LCF. Since the representation of data types as retracts has already been discussed, it remains to show how sets are to be represented. Based on the set representation we then have to find interpretations for the set operations and quantifications.

The most common way of introducing sets into an environment of structures is by representing them as sequences (linear lists) of non-repeating elements. As we are not interested in axiomatizing set theory but rather look for convenient definition of function meanings we rely on such a representation in LCF (cf. [Ne1]). It will turn out later that sets are needed mainly as a conceptual intermediate step which can be eliminated in actual “programs”. Besides, representing sets by sequences fits nicely into the algebraic framework. Actually, what is to be represented is a rather restricted kind of sees: we are only dealing with homogeneous and computable (mainly even finite) sets. However, the required homogeneity is not really restrictive as one can always define the "sum type.”

The first step is to define a membership predicate $x \epsilon S$ for sequences, yielding tt if x occurs in S and ff otherwise (if it is defined). It is definable as a homomorphism from Seq into T by homomorphically extending equality on atoms:

$$\epsilon \equiv [\lambda x.\ Sfun([\lambda y. x \equiv y], ff, \vee)]$$

Note that ϵ is defined for appropriate types only; if x does not have the same type as the elements of S, $\equiv$ is undefined, thus also $x \epsilon S$.

Using the predicate $\in$ a function u_1 on atom $\times$ seq is definable by

$$u_1 :\equiv [\lambda x\ S.\ x \in S \supset S,\ mkseq(x,S)]$$

u_1 guarantees that elements already occurring in a sequence will not be added; sequences built up using u_1 are those directly representing sets. If Seq is the data type of sequences of elements of type t, the type setof(t) is the image of Seq under the homomorphism

$$set :\equiv Sfun(id,\ emptyseq,\ u_1).$$

Moreover, set is a *retract* on seq; it defines the same subdomain of seq as the function norepseq discussed in subsection 3.4.: setof(t) corresponds to the subset of sequences without repetitions of elements. (However, it is *not* a subalgebra of seq.) From this it follows that functions defined on Seq are equally defined on setof(t). Furthermore, the (generic) structure of Seq can be used for defining functions on setof(t). More specifically, we have the embedding $iset: setof(t) \to seq$ with

$$set \circ iset \equiv id_{set}\ .$$

Thus, any function $f: seq \to D$ can be restricted to set by composing with iset. In this way, the predicate $\in$ defined above becomes the set-theoretic element relation. Similarly, we obtain an interpretation of quantified terms by applying homomorphic extension to any predicate P. Let operators all and exist be defined by

$$all :\equiv [\lambda P.Sfun(P,\ ff,\ \wedge)$$
$$exist :\equiv [\lambda P.Sfun(P,\ ff,\ \vee)$$

Then

$$\forall x \in S.\ P(x) :\equiv all(P,S)$$

and

$$\exists x \in S.\ P(x) :\equiv exist(P,S).$$

Note that this form of quantification is well-defined if S and P are defined; since $\perp$ or $\top$ is never an element of a set, it will not appear in quantifications (and cause a non-defined truth value). Furthermore, a quantified term denotes a computable function if the predicate P and the term denoting the restricting set S are computable, which is guaranteed by the way terms can be built up.

Using these constructs, set inclusion is easily expressed by

$$S_i \subseteq S_j :\equiv \forall x \in S_i.\ x \in S_j$$

and- similarly set equality by the "extensionality" property

$$S_i \equiv S_j :\equiv (\forall x \in S_i.\ x \in S_j) \wedge (\forall x \in S_j.\ x \in S_i)$$

Note again that these relations will be undefined for sets over different types. The empty set is the image under set of the empty sequence; we will identify the former with the latter.

The function $\cup_1$, taken as a function from $t \times setof(t)$ to $setof(t)$, inserts a single element into a set; extending this function homomorphically in the first argument yields ordinary set union $\cup$. As a short hand notation we will use $\cup_n$ for n-ary union (n-l-fold composition of $\cup$).

Similarly, *set intersection* and *set difference* are definable by means of the function remove: $t1 \times setof(t1) \rightarrow setof(t1)$ that removes an element from a set. remove is defined by

$$remove := [Xx\ s.\ x \in s \supset rem(x,s),\ s]$$

where rem is the endomorphic extension of

$$rem_1 := [Xx\ y.\ x = y \supset \{\}, \{y\}].$$

If $\{x \mid x \in S\}$ is used as an equivalent notation for S, the term language can be extended to include sets that are characterized by predicates. However, one has to be careful: a set $\{x \mid P(x)\}$ need not be constructive even for computable P, if no domain is indicated. Therefore, predicates for set formation have to be restricted to those based on set expressions, i.e. elementary predicates $x \in S$. All other predicates have to be restrictive in the sense that they restrict a set to a subset ("filter predicates").

Definition 4.3 (admissible set predicates):

The set of admissible set predicates is defined by

(1) The elementary predicates $x \in S$ are admissible set predicates.

(2) If P is an admissible set predicate and Q any predicate, then $P \wedge Q$ is an admissible set predicate.

(3) If P and Q are admissible set predicates, then $P \vee Q$ and $P \backslash Q$ are admissible set predicates.

Lemma 4.4:

if P and Q are admissible set predicates, then

$$\{x \mid P(x) \vee Q(x)\} = \{x \mid P(x)\} \cup \{x \mid Q(x)\}$$

and

$$\{x \mid P(x) \wedge Q(x)\} = \{x \mid P(x)\} \cap \{x \mid Q(x)\}$$

It can be shown that the operations defined here have most of the standard properties. However, the well-known problems caused by only partial recursive predicates still remain. For example,

$$\neg(\forall x{\in}S.\ P(x)) \equiv \exists x{\in}S.\ \neg P(x)$$

is true only if P is *total* on the domain under consideration.

It is obvious that the representation of sets and set operations provide a model for a theory of (finite) sets. In particular, a first-order like calculus based on the restricted quantifiers is available for proving properties of functions. Note that this calculus is constructive in the sense that all expressions denote computable functions (cf. [Co]).

As the type system does not include basic set types, sets have to be generated from objects that are not sets. There is a canonical way of deriving set-valued functions from types. Recall that a type t_i is a base type for a data type t_j if it occurs in its definition. For each type t_i that is a base type for t_j a function

$$\text{set\_of\_}t_i\text{: } t_j \rightarrow \text{setof}(t_i)$$

is obtained by homomorphically extending the mapping base-type $\rightarrow$ singleton-set. More precisely, in the homomorphic extension constructors are replaced by set union (with appropriate arity); those parts of a structure that do not involve elements of type t_i are mapped onto the empty set. An example can be found in section 5.

4.2 Schemes for Function Definition

In section 3 we introduced a method called "homomorphic extension" for defining new functions over a data type. A particularly simple special case of this method is the *endomorphic extension* of a function. An *endomorphism* is a homomorphism from an algebra into itself. Since all the algebraic operations remain unchanged, the only parameters of endomorphic extension are the functions on the base types to be extended. A typical example is substitution of terms for variables. Recall the data type definition for binary trees over atoms from section 3:

bintree := mkbt(sub:atom) | comp(fir:bintree, sec:bintree)

where atom is the generating base type. The corresponding endomorphic extension functional is

```
BTend := [λf. [αE. [λx . is_mkbt(x) ⊃ f(atom(sub(x))),
                          is_comp(x) ⊃ comp(E(fir(x)), E(sec(x))),
                          ⊥]]]
```

Now, if we want to solve the problem

"Find a function varsubst: **bintree→bintree** *such that* varsubst *replaces all atoms in a binary tree by their values under the function* varsub: atom → bintree,"

then a solution is simply

varsubst ≡ **BTend(varsub),**

and this solution is even unique, as it was shown in section 3.

So far, we have been looking at homomorphisms only. Unfortunately, many interesting functions can not be represented as homomorphisms. But we can apply a similar definition technique to a larger class of functions simply by explicitly stating the non-homomorphic part of the function and using the extension functional for the homomorphic rest. This situation occurs often with data types which include several composed subtypes; an example can be found in the next section.

The functionals derived from a data type definition (for homomorphic, endomorphic extension etc.) not only permit definition of new functions in a concise way, they also facilitate proving properties. In fact, certain properties of those functions derive from properties of the function& like the induction proof rule already mentioned above.

Lemma 4.5:

If the argument junctions of an extension functional are strict/total then the resulting junction is strict/total.

Note that totality entails that any program derived from a function by "meaning-preserving" transformations terminates on defined inputs.

There are other definition schemes that hitherto have defied a natural algebraic interpretation. Consider, for example, the following form of function iteration. Let the expression

$$[\,\forall x \in S : f(x,z)]$$

be interpreted as "For each x in S apply [$\lambda y.\, f(x,y)$] to z." This can be made more precise by a recursion on the sequence representing S:

$$[\lambda S\ z.\ [\forall x \in S : f(x,z)]] \equiv [\alpha F.\ [\lambda S\ z.\ is\_emptyseq(S) \supset z, F(tl(S), f(hd(S), z))\,]]$$

However, this interpretation causes some problems. In order to be a conservative extension of the specification language as defined so far the given interpretation has to be consistent with the notions introduced previously. In particular, if two sets S and S' are equal one would expect

$$\forall x \in S: f(x,z) \equiv \forall x \in S': f(x,z)$$

This implies that the applications of the $f(x,\_)$ must be independent of the particular representation of S, i.e. the "hidden order" on S ; or, at least, it must be guaranteed that the sequence of applications of f can be executed in any order. This virtually restricts applicability of the construction; in many cases it may not be easy or even possible to verify this kind of commutativity. Although operators like function iteration are necessary to make the specification language powerful enough, they will not be discussed further in this paper.

4.3 Transformation of Function Definitions into Programs

So far we have been discussing methods for defining functions over structured data and their interpretation in LCF. Now, every LCF term also has an interpretation as a computation rule for the function denoted by it. Given such an interpreter for LCF this allows to compute all the functions definable in the language. However, the resulting computations would be quite inefficient, in particular because of nestings of unnecessary recursions resulting from direct interpretation of the constructs. Consider, for example, the expression

$$F(y) \equiv \forall x \in S(y).\ P(x)$$

where the type of y is the data type list as defined above and S the standard set-valued function **set_of_atom**. Since list is a recursive type, one recursion is required to compute S(y) and another one to compute the quantified expression; but WC can do much better by utilizing the underlying algebraic structure. Note that the value of F is determined by the values of P(x); moreover, we have

$$P \equiv F \circ \mathrm{mkbt}$$

which means that F is a homomorphic extension of P. Because of the uniqueness property it follows that

$$F \equiv \mathrm{BThom}(P, A)$$

where **BThom** is the homomorphic extension functional for **bintree**. This means that F can be replaced by an equivalent function that involves only one recursion. Apart from that, the explicit representation of the set S(y) is eliminated.

This is an example of how the algebraic concepts can be used to simplify function definitions considerably. It shows that the interpretation of the specification language is not a case of simple macro expansion, but a possibly non-deterministic process of simplifying expressions in a suitable way, which is similar to, e.g., theorem proving. More heuristic methods for recursion removal have been studied by R. Burstall and J. Darlington [BD].

The regular expression structure that results from defining functions by means of definition schemes is of advantage at all levels of program development. Apart from the techniques for proving properties about them (see above) it permits uniform application of optimizing transformations, like replacing recursion by iteration. Even at the implementation level it can be advantageous: For example, functions defined by endomorphic extension can be implemented in such a way that no additional storage (for data) is required (cf. selective updating in [Ho]). If it has been proved that the transformation and implementation techniques preserve meanings, then the "correctness" of resulting programs can be guaranteed. Meaning preserving transformations will be studied in greater detail in a subsequent paper.

5. An Example: Substitution with α-Conversion

5.1 The Data Types

In the example now to be discussed we have four data types, defined by

```
term :≡  mkcterm(constof:const) |
         mkvterm(varof:var) |
         mkapply(funeof:term, argeof:term) |
         mklambda(bvarof:bvar, termof:term) |
         mkmu(bvarof:bvar, termof:term) |
         mkcond(condof:term, trueeof:term, falseeof:term)

bvar :≡  mkbvar(varof:var)
const :≡ uu | tt | ff | 00
var is taken as basic and not further specified.
```

The reader will notice that these data types represent the abstract syntax of LCF terms. In algebraic terms the types form a heterogeneous algebra with the four phyla term, bvar, var, and const and operations

```
(op1) mkcterm:  const → term
(op2) mkvterm:  var → term
(op3) mkapply:  term×term → term
(op4) mklambda: bvar×term → term
(op5) mkmu:     bvar×term → term
(op6) mkcond:   term×term× term → term
(op7) mkbvar:   var → bvar
```

The generating phyla (data types) are **const** and **var**. Obviously the phyla var and bvar are isomorphic; the reason for introducing the data type **bvar** is that it is more convenient to separate the binding occurrences of variables from the other ones.

From the data type definitions the following characterizing functions are generated:

```
term :≡ [αF.[λx. is_const(x) ⊃ mkcterm(const(constof(x)),
                 is_mkvterm(x) ⊃ mkvterm(var(varof(x)),
                 is_mkapply(x) ⊃ mkapply(F(funeof (x), F(argeof(x))),
                 is_mklambda(x) ⊃ mklambda(bvar(bvarof(x)), F(termof(x))),
                 is_mkmu(x) ⊃ mkmu(bvar (bvarof (x)), F(termof (x))),
                 is_mkcond(x) ⊃ mkcond(F(condof(x)), F(trueof(x)), F(falseof(x))),
                 ⊥ ]]

bvar : ≡ [λx. mkbvar(var(varof (x)))]
```

In order to define a homomorphism we have to supply 7 operations of appropriate types. 6 of them correspond to the constructors occurring in the characteristic function term; the last one is to replace mkbvar. By substituting the characteristic function for bvar in term we obtain an expression that includes all operations and completely defines homomorphic extension. For endomorphic extension only the operations on the generating subtypes (i.e. op1,op2 and op7) are required. Let

termhom := [λ op1 op2 . . . op7. [αF. ..]]

and

termend := [λ op1 op2 op7. [αF. . .]]

be the functionals for homomorphic and endomorphic extension.

5.2 The Problem

We want to formalize the following (cf.[AW]):

Replace *any free occurence of the variable v in the expression (term) e by the term t after renaming bound variables in e suitably (i.e. so that no free variable in t will become bound in e) (a common notation is e[t/v]).*

What is described above is the basic conversion rule of the X-calculus as it is incorporated in the LCF system. It may be desirable to have a system that is smart enough to understand this description of substitution and to translate it from English into a programming language, At present, such a system is not available. It would require knowledge about what exactly is meant by "free occurrence", "replace", "renaming" etc. For the time being we have to be satisfied with specifying those notions in some kind of formal language and having a less ambituous system transform the specification statements into executable code. In any case, we need a formal definition in order to be able to prove anything about the function.

We construct a function subst: var× term×term → term by stepwise specifying the informal notion in our language. Let subst be defined by

subst := [Xv t e. substfree(v, t, renamebvar(e,t))]

where

substfree(v,t,e) := "replace all free occurences of v in e by t"
renamebvar(e,t) := "rename bound variables in e that occur free in t appropriately"

a) bound variables in term. The function boundvarsin: term → setof(var) returns a set of variables for which there is a binding occurrence in the term. This is just the standard set function set-of-bvar composed with the isomorphism variso from bvar to var, extended to sets. Here we can see how the separation of the type bvar from var facilitates definition of set-valued functions. set-of-bvar is the homomorphism defined by the operations

b_1 := b_2 := [Xx. {}]	(empty set)
b_3 := b_4 := b_5 := ∪	
b_6 := $∪_3$	
b_7 := [Ax. (x)]	(singleton map)

i.e., set-of-bvar $\equiv$ **termhom(b_1, . . ,b_7).**
Then

boundvarsin :≡ variso ∘ set-of-bvar

b) free *variables in term.* The standard function **set-of-vars** returns all **occuring** variables regardless of whether they are free or not. So we have to update that function appropriately to get a function **that** returns only free variables. If we had separated the λ- and cc-terms from the type term we could use a standard set-of-dtype function for defining **freevarsin.** Instead, we define it directly as 'as a homomorphism

freevarsin: term → **setof (var).**

Using the set-valued functions

f_1 :≡ [λx. {}]
f_2 :≡ f_7 :≡ [λx. {x}]
f_3 :≡ ∪
f_4 :≡ f_5 :≡ [λx y. y\x] (set difference)
f_6 :≡ $\cup_3$

the function is definable by

freevarsin :≡ **termhom(f_1, . . ,f_7)**

c) *Renaning bound variables.* We need a function newvar that "invents" new variables (which do not occur in **either e** or **t).** Strictly speaking the existence of newvar depends on a function that enumerates **all** variables and returns the first element with a certain property. In any **practical** implementation we "know" all the variable names available to the user, so a function that generates new names is available. In the abstract context it is sufficient to assume the existence of a strict and **total** function newvar that returns a variable **with** the property

$$\neg\, \mathbf{newvar(v,e,t)} \in \mathbf{varsin(e)} \cup \mathbf{varsin(t)} \cup \{v\}.$$

Using this function we can specify.renaming of bound variables:

renamevar(e,t) :≡ "rename *in* **e** *each variable that* occurs *free in* **t** *and bound in* **e**"

formally:

renamevar :≡ [λe t. [∀x ∈ **freevarsin(t)** ∩ **bvarsin(e)** : **rename(t,x,e)**]]
rename' :≡ [λt. [λx **e. termend(mkcterm,** replacevar, mkbvar ∘ replacevar)]]

where replacevar denotes the term **[λz. z=x ⊃ newvar(x,e,t), z].** Note that the use of the iteration **construction** is justified by the fact that renaming of bound variables can be done in any order; **all** resulting terms are equivalent.

d) *substfree*. "Free occurrence" means "not bound", i.e. "not in the range of a λ or ∝ binding that variable." *So*, in order to find occurrences of a variable v we have to search (recur in) the tree representing the term e. Whenever we come across a λ or ∝ (that is, a mklambda or mkmu) that binds v, we stop and return. Then any remaining occurrence of v is a free one and is to be replaced by t. In the formal language this is expressed by a construction using a modified functional for endomorphic extension:

```
substfree   :≡ [∝S. [Xv t. [ λe. is_mkvterm(e) ⊃ varof (e)=v ⊃ t,e,
                              (is_lambda(e)vis_mu(e)) A bvarof(e)=v ⊃ e,
                              term0(S(v,t),e)]]].
```

Here term0 is the operator on F that defines term, i.e. term ≡[∝F. [Xx. term0(F,x)]],

This finishes the formal specification of the substitution function. The collection of all the function definitions

```
subst          :≡ [Xv t e. substfree(v,t,renamevar(e,t))]
substfree      :≡ [ ∝S. [ Xv t. [ λe. is_mkvterm(e) ⊃ varof(e)=v ⊃ t,e,
                                  (is_lambda(e)vis_mu(e)) A bvarof(e)=v ⊃ e,
                                  term0(S(v,t),e)]]]
renamevar      :≡ [ λe t. [ Vx ∈ freevarsin(t) ∩ bvarsin(e) : rename(t,x,e)] ]
rename         :≡ [λt. [Xx e. termend(mkcterm, replacevar, mkbvar∘replacevar)]]
replacevar     :≡ [λz. z=x ⊃ newvar(x,e,t),z]
bvarsin        :≡ variso ∘ set-of-bvar
set-of-bvar    :≡ termhom([ Xx.{} ], [Xx.{)], ∪, ∪, ∪, ∪3,[λx.{x}])
freevarsin     :≡ termhom([λx.{}], [λx.{x}], ∪, \, \, ∪3, [λx.{x}])
```

is somewhat longer than the informal description in English, yet it is complete in the sense that a sufficiently smart system can transform it into a reasonably efficient program, using transformations of the sort indicated in the preceding section.

6. Concluding Remarks

In this paper, the representation of abstract data types in LCF and the algebraic interpretation of structures were discussed. This led to constructions that permit to specify functions operating on data structures in a concise way and close to what may be considered "natural." The methods were demonstrated in an example taken from the actual LCF system.

The construction methods considered here constitute only a first step towards an elaborated language that will allow easy and concise definition of complex functions as they are needed in, e.g., structure manipulating systems. There are many directions in which the work presented here has to be extended. Some have already been mentioned in the preceding sections: systematic extension 'of the system of data type; more general function definition schemes; general methods for transforming and optimizing function definitions, in particular for removal of redundant recursions; the translation of logical expressions into a "real-life" programming language. In the paper, only methods for explicit function definition have been discussed. However, it appears that techniques for solving equations that define functions implicitly can similarly be derived from the explicit representation of the data structure by a retract. The retract could serve for guiding the search for solutions and for structuring the resulting program. The development of such problem solving methods in the framework of LCF has to be left to future studies.

How much of the methods discussed here can be automated? It is obvious that the generation of the appropriate set of axioms, of function definition schemes and rules for structural induction from the data type definitions is straightforward and can be completely automated. Furthermore' many checks for simplifications and transformations can be done on a purely syntactic level accessible to automation, So it should be easy to incorporated all these features and special knowledge about the restricted set calculus into an interactive system for developing programs and proving theorems about them.

R. References

[A1] Allen, J.: *Anatomy of a Language: LISP.* Forthcoming book.

[AAW] Aiello, L., Aiello, M., and Weyhrauch, R.W.: *The Semantics of PASCAL in LCF.* Memo AIM-22 1, Stanford University, 1974.

[AW] Aiello, L. and Weyhrauch, R.W.: *LCFsmall: an implementation of LCF.* Memo AIM-241, Stanford University, 1974.

[BiL] Birkhoff, G., and J.D. Lipsom : *Heterogeneous algebras.* Journ.Comb.Theory 8 (1970), 115-133.

[BuD] Burstall, R.M., and J. Darlington : *Some transformations for developing recursive programs.* Proc. Int. Conf. on Reliable Software, Los Angeles, April 1975.

[BuL] Buchanan, J.R. and DC. Luckham: *On automating the construction of programs.* Memo AIM-236, Stanford University, 1974.

[Co] Constable, R.L.: *A constructive theory of - recursive' functions.* Technical Report 73-186, Cornell University, October 1973.

[Ea] Earley, J.: *High level operations in automatic programming.* Proc. Symp. on Very High Level Languages, SIGPLAN Notices 9.4 (1974).

[He] von Henke, F. W.: *Notes on automating theorem proving in LCF.* forthcoming.

[Hi] Higgins, P. J.: *Algebras with a scheme of operators.* Math. Nachr. 27 (1963), 115- 132.

[Ho] Hoare, C.A.R.: *Recursive Data Structures.* Memo AIM-223, Stanford University, 1973.

[Mc] McCarthy, J.: *A basis for a mathematical theory of computation.* in: Computer Programming and Formal Systems, (ed. Braffort and Hirschberg), North Holland (1963).

[MaW] Manna, Z. and R. Waldinger: *Knowledge and reasoning in program synthesis,* Techn.Note 98, Stanford Research Institute, Nov. 1974.

[Mi1] Milner, R.: *Logic for computable functions - description of an implementation.* Memo AIM-169, Stanford University, 1972.

[Mi2] Milner, R.: *Implementation and applications of Scott's logic for computable functions.* Proc. ACM Conference on Proving Assertions about Programs, New Mexico State University, Las Cruces, New Mexico, 1972.

[MiW] Milner, R. and Weyhrauch, R.W.: *Proving compiler correctness in a mechanized logic,* Machine Intelligence 7, ed. D. Michie, Edinburgh University Press, 1972.

[Mo] Morris, F.L.: *Correctness of Translations of Programming Languages - an algebraic approach.* Memo AIM- 174, Stanford University, 1972.

[Ne1] Newey, M.: *Axioms and Theorems for Integers, Lists and finite Sets in LCF.* Memo AIM-184, Stanford University, 1973.

[Ne2] *Newey,* M.: *'Formal semantics of LISP with applications to program correctness.* Memo A IM-257, Stanford University, 1975.

[Sc1] Scott, D.: *Continuous Lattices.* Proc. of the 1971 Dalhousie Conference, Springer Lecture Notes.

[Sc2] Scott, D.: *Data types as lattices.* Forthcoming Springer Lecture Notes.

[WM] Weyrauch, R. and Milner, R.: *Program semantics and correctness in a mechanized logic,* Proc. USA -Japan Computer Conference, Tokyo, Oct 1972.

A. Appendices

A.1 Logic for Computable Functions

A.1.1 Syntax of Typed LCF

The following is an extract taken from [Mi1].

Types At bottom tr and ind are types. Further if $\beta 1$ and $\beta 2$ are types then $(\beta 1 \rightarrow \beta 2)$ is a type. With each term of the logic there is an unambiguously associated type. For a term t we write $t:\beta$ to mean that the type associated with t is β.

Terms (metavariables s,t,s1,t1,...) The following are terms:

Identifiers (metavariables x,y) - sequences of upper or lower letters and digits. We assume that the type of each identifier is uniquely determined in some manner.

Applications - s(t) : $\beta 2$, where $s:\beta 1 \rightarrow \beta 2$ and $t:\beta 1$.

Conditionals - $(s \rightarrow t1,t2):\beta$, where s:tr and $t1,t2:\beta$.

X-expressions - $[\lambda x.s]$: $\beta 1 \rightarrow \beta 2$, where $x:\beta 1$ and $s:\beta 2$.

α-expressions - $[\alpha x.s]$: β , where $x,s:\beta$.

The intended interpretation of the d-expression $[\alpha f.s]$ is the minimal fixed-point of the function or functional denoted by $[\lambda f.s]$. For example:

$$[\alpha f.[\lambda x.(p(x) \rightarrow f(a(x)),b(x))]]$$

denotes the function defined recursively as follows:

f(x) <= if p(x) then f(a(x)) else b(x).

Constants The identifiers tt, ff denote truthvalues true and false. $\perp$ denotes the totally undefined object of any type: in particular, the undefined truthvalue.

Atomic well-formed formulae (awffs) The following is an awff:

$$s \subset t$$

where s and t are of the same type. The intended interpretation of $s \subset t$ is, roughly, that t is at least as well defined as, and consistent with, s.

Well-formed **formulae (wffs)** (metavariables P,Q,P1,Q1,...) Wffs are sets of zero or more awffs, written as lists with separating commas. They are interpreted as conjunctions. We use

$$s \equiv t$$

to abbreviate $s \subseteq t, t \subseteq s$.

Sentences Sentences are implications between wffs, written

$$P \vdash Q$$

or, if P is empty, just $\vdash Q$.

Proofs A proof is a sequence of sentences, each being derived from zero or more preceding sentences by a rule of inference.

The strict syntax for terms and awff's is relaxed in the machine implementation to allow a saving of parentheses and brackets. In addition, we use the abbreviation

$f(x,y)$	for $f(x)(y)$
v x. $t \equiv s$	for $\lambda x.t$ • Xx.8
$p :: q \equiv r$	for $p \supset q, \perp \equiv p \supset r, \perp$

Functions are used in infix notation where it is obvious what is meant.

A.1.2 Type free LCF

The type free version of LCF differs from the typed one essentially in the handling of truth values and conditional expressions. Apart from that it also specifies the structure of the domain. Besides the truth values there are constants T for the truth values retract and I for the universal domain. In the following the additional axioms and rules of inference are listed.

MAX $\vdash s \subseteq T$

COND $\vdash T \rightarrow s,s \equiv s$

$\vdash r \rightarrow s,t \equiv T(r) \rightarrow s,t$

$\vdash T(s)(t) \equiv T(s)$

$\vdash T \subseteq T \rightarrow T$

$\vdash I \equiv [\alpha J.\ T \sqcup (J \rightarrow J)]$

The CASES-rule is changed to

$$\text{CASE-S}\quad \frac{P \vdash Q\{\perp/x\} \quad P \vdash Q\{tt/x\} \quad P \vdash Q\{ff/x\} \quad P \vdash Q\{\top/x\}}{P \vdash Q\{T/x\}}$$

Other defined standard terms:

:⊃ := [Xx y z. x ⊃ x ⊃ y, T, z]

→ := [Xx y. λ z. y × z × x]

A.2 Special **Functions** in **tfLCF**

A.2.1 **Definedness** predicate

We want a predicate ∂ such that

$$\vdash \partial(x) \equiv tt,\ x \equiv \bot,\ x \equiv \top$$

Define ∂ by

$$\partial \equiv \mathbf{upt} \cup \mathrm{down}$$

where

$$\mathrm{down} \equiv [\lambda\, x.\ x :\supset \bot, \bot]$$

and

$$\mathbf{upt} \equiv [\alpha\, P.\, [\lambda\, x.\ (x \supset tt, tt) \cup P(x(\top))]] \qquad (\equiv [\lambda x.\ \mathrm{uptf}(x) \supset tt, tt])$$

down maps everything to $\bot$ except $\top$ which goes to $\top$:

$$\mathrm{down}(x) \equiv \top \vdash \top(x) \equiv \top \vdash x \equiv \top$$

upt maps everything to tt except $\bot$ which is mapped to $\bot$. The desired properties of ∂ are then obvious.

A.2.4 **istrue**

- Our aim is to give a function that
  - maps everything on a truth value and
  - gives the values tt and ff exactly for the arguments tt and ff resp.

This function will enable us to test effectively variables for "well defined" truth values. In the type-free logic, the simple conditional does not provide this function as it is defined "relative to the truth values retract T .". However, **we can define istrue** using a limit construction. That such a definition is possible at all is due to the fact that the truth values are isolated points in the **lattice I.**

Definition: $\mathbf{istrue} \equiv [\alpha\, S.\, [\lambda\, x.\ x :\supset tt \cup S(x\ \mathrm{T}),\ ff \cup S(x\ \mathrm{T})\ ]]$

It is easy to show by cases that

(1) $\quad \mathrm{T} \subset \mathbf{istrue}$

Since $\forall\, x.\ \mathrm{T}(x) \subset x$ we also have

(2) - $\quad \mathrm{T} \circ \mathbf{istrue} \subset \mathbf{istrue}$

From the definition follows immediately

(3a) $\quad \mathrm{istrue}(x) \equiv tt \vdash \mathrm{istrue}(x(\top)) \subset \mathrm{istrue}(x)$
(3b) $\quad \mathrm{istrue}(x) \equiv ff \vdash \mathrm{istrue}(x(\top)) \subset \mathrm{istrue}(x)$

also

(4) $T(x) \equiv TV \vdash istrue(x) \equiv TV$ for $TV \equiv \perp, \top$

Next we show by induction on **istrue**

(5) $\forall x.\ istrue(x) \sqsubseteq T(istrue(x))$

I. $\perp(x) \sqsubseteq \ldots$ ok.

II. Assume $\forall x.\ S(\kappa) \sqsubseteq T(istrue(x))$. We have to show
$x :\rightarrow tt \cup S(x\ T),\ ff \cup S(x\ \top) \sqsubseteq T(istrue(x))$.
By Cases T(x):

$T(x) \equiv \perp$: trivial
$T(x) \equiv \top$: implies $istrue(x) \equiv \top$, trivial.

$T(x) \equiv tt$:
$ihs \equiv tt \cup S(x\ \top) \sqsubseteq tt \cup T(istrue(x\ \top))$ by Ind.Hyp.
$\equiv T(tt \cup istrue(x\ \top))$ by L54 Mi-We
$\equiv T(istrue(x))$.

$T(x) \equiv ff$: analog

With (2) we have shown

(6) $istrue \equiv T \circ istrue$.

which means that the range of **istrue** is a set of truth-values.

On the other hand we already mentioned that

(6a) $istrue(tv) \equiv tv$

holds for each truth value **tv**. Thus, in a short notation

(7) $istrue \circ T \equiv T$

i.e. **istrue** is an identity on T. From (6) and (7) we deduce the retract property for **istrue**:

$$\begin{aligned} istruo \circ istruo &\equiv istruo \circ (T \circ istrue) \\ &\equiv (istrue \circ T) \circ istrue \\ &\equiv T \circ istrue \\ &\equiv istrue \end{aligned} \tag{8}$$

The ultimate goal is to show

(×) $\mathrm{istrue}(x) \equiv tt \vdash x \equiv tt$
$\mathrm{istrue}(x) \equiv ff \vdash x \equiv ff$

i.e. istrue is a truth-valued function that gives the values tt or ff exactly for tt and ff resp. In orde
to do so we introduce another truth-valued function:

Definition: $\mathrm{uptf} \equiv [\alpha\, P.\ [\lambda\, x.\ T(x) \cup P(x(T))]]$

By definition we have

(U1) $T \subset \mathrm{uptf}$

We prove the following facts about uptf:

(U2) $x \subset \mathrm{uptf}$

(9) $\mathrm{istrue} \subset \mathrm{uptf}$

(10) $\mathrm{istrue}(x) \equiv tt \vdash \mathrm{uptf}(x) \subset \mathrm{istrue}(x)$

(9) and (10) together show

(11) $\mathrm{istrue}(x) \equiv tt \vdash \mathrm{uptf}(x) \equiv tt$

With (U2) it follows that

(12) $x \subset tt$

on the other hand, since $tt \equiv T(x) \subset K$ we have

(13) $\mathrm{istrue}(x) \equiv tt \vdash x \equiv tt.$

The proof for the corresponding statement for ff follows the same line.

A.3 Structural induction

The basic idea of how to do structural induction in LCF is that it can actually be simulated if a recursive function "describing" the structure is available. For the kind of structures we are interested in in this paper the retraction constructed from the type definition serves this purpose. So, structural induction becomes a mere application of computational induction. The derivation of the induction rule as in theorem 3.x is done in two steps: 1) first derive a rule involving the retraction; 2) modify the rule in 1) by using the type-predicate. Since proving the rule in full generality would be rather tedious, it is demonstrate by means of the example seq.

Recall that seq is defined by the retraction

$$\text{seq} \equiv [\alpha S.\ [\lambda x.\ \text{is\_emptyseq}(x) \supset \text{emptyseq},\\ \text{is\_mkseq}(x) \supset \text{mkseq}(\text{atom}(\text{hd}(x)),\ S(\text{tl}(x))),\\ \perp]]$$

First, we prove the rule

$$\text{(R1)}\quad \frac{(a_1)\ P(\perp)\qquad (a_2)\ P(\top)\qquad (a_3)\ P(\text{emptyseq})\qquad (a_4)\ \forall x.P(y)\ \vdash\ \forall x.P(\text{mkseq}(x,y))}{P(\text{seq}(x))}$$

where x and y do not occur in P. By computational induction, we can deduce

$$\forall x.\ P(\text{seq}(x))$$

from

$$\forall x.\ P(\perp\ K)\qquad \text{and}\qquad \forall x.\ P(S\ K)\ \vdash\ \forall x.\ P(\text{tau}(S)(x)).$$

Since

$$P(\perp\ x) \Leftrightarrow P(\perp)$$

the base case is proved by premise (b_1).

Now assume

$$\forall x.\ P(S\ X)$$

In order to prove

$$\forall x.\ P(\text{tau}(S)(x))$$

we expand tau(S) to

$$\text{tau}(S)(x) \equiv \text{is\_emptyseq}(x) \supset \text{emptyseq},\\ \text{is\_mkseq}(x) \supset \text{mkseq}(\text{atom}(\text{hd}(x)),\ S(\text{tl}(x))),\\ A$$

and split into cases which then can be deduced from appropriate premises:

is_exptyseq(x) ≡ ⊥ :		tau(S)(x) ≡ ⊥	by premise (a_1)
	≡ tt :	≡ emptyseq	by (a_3)
	≡ T :	≡ T.	by (a_2)
	≡ ff : case split for is_mkseq(x) : only tt is interesting:		
		≡ mkseq(atom(y),S(x))	by (a_4)

Now, the rule

$$\frac{(b_1)\ P(\text{emptyseq})\qquad (b_2)\ P(y)\ \vdash\ P(\text{mkseq}(x,y))}{\forall x.\text{is\_seq}(x) \Rightarrow P(x)}$$

follows from (R 1) by virtue of the facts that the reiativiring type predicate eliminates the cases $\perp$ and T.