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

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited
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' 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 Austauschdienst, or the U. S. Government.

Reproduced in the U.S.A. Available from the National Technical Information Service, Springfield, Virginia 22151.
Contents

1. Introduction

2. The type-free Logic for Computable Functions
   2.1 Type-free LCF
   2.2 Retracts, Domains, Types

3. Abstract Data Types in the Type Free Logic
   3.1 Data Type Definitions
   3.2 Representing Data Structures in tfLCF
   3.3 Algebraic Interpretation of Data Types
   3.4 Non-generic Data Types
   3.5 An Example: Infix to Postfix Translation

4. Elements of a Problem Specification Language
   4.1 Sets, Set Operations and Quantification
   4.2 Schemes for Function Definition
   4.3 Transformation of Function Definitions into Programs

5. An Example: Substitution with α-Conversion
   5.1 The Data Types
   5.2 The Problem

6. Concluding Remarks

R. References

A. Appendices
   A.1 Logic for Computable Functions
   A.2 Special functions in tfLCF
   A.3 Structural Induction
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 synthesizing 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] 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.
Introduction

of expressions. 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].
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.

2.1 Type-free 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 \( \lambda \)-calculus [Sc1]: the domain \( I \) which is constructed over the 4-element lattice \( T \) of truth values:

\[
\begin{array}{c}
T \\
\downarrow \\
\downarrow \\
\downarrow \\
\top
\end{array}
\]

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 \( p \rightarrow q \) 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 \( T \rightarrow x \cdot y \) is not further specified except that \( T \rightarrow x \cdot x \cdot x \). However, it turns out that it can be taken as the join of \( x \) and \( y \) (see below).
The type-free Logic for Computable Functions

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

The element \( \perp \) is called *undefined*, and the element \( \top \) is called *overdefined*; all other elements are called *defined*. A predicate \( \delta \) can be defined in \( I \) such that

\[
\delta(x) = \top \quad \text{iff} \quad "x \text{ is defined}," \quad \delta(\perp) = \bot, \quad \delta(\top) = \top
\]

i.e., \( \delta \) yields the distinction between defined and non-defined elements in \( I \). \( \delta \) is definable in the logic by a mapping onto the 3-element lattice \( \{ \perp, \top, \bot \} \). 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 \( \perp \) or \( \top \) whenever the argument is \( \perp \) or \( \top \) respectively, that is if the following wff is true of \( f \):

\[
\forall x . ( \delta(x) \Rightarrow \delta(f(x))) = \top
\]

\( f \) is called \( \perp \)-strict if \( f(\perp) = \perp \), and \( \top \)-strict if \( f(\top) = \top \). \( f \) is called *total* if it never returns \( \perp \) or \( \top \) for a defined argument, i.e., if

\[
\delta(x) \Rightarrow \delta(f(x)) = \top
\]

holds. Thus, if a function \( f \) is strict and total then \( \delta(x) = \delta(f(x)) \).

Any function \( f \) can be made into a strict one by first applying \( \delta \) to the argument: For

\( f' := [\lambda x . \delta(x) \Rightarrow f(x), \perp] \)

we obviously have

\[
f'(x) =
\begin{cases} 
\top & \text{if } x = \top \\
\perp & \text{if } x = \perp \\
 f(x) & \text{otherwise}
\end{cases}
\]

In the next section a functional \( \text{str} \) will be defined that turns any function into a strict one.

In the following we define some standard operators on \( I \) that will be used throughout the paper.

\[
\begin{align*}
o & := [\lambda x y z . x(y(z))] & \text{function composition} \\
pair & := [\lambda x y z . z \Rightarrow x y] & \text{ordered pair} \\
\pi_1 & := [\lambda x . x(\top)] & \text{projection onto first component} \\
\pi_2 & := [\lambda x . x(\top)] & \text{projection onto second component} \\
x & := [\lambda x y z . \text{pair}(x(z \top), y(z \#))] & \text{cartesian product}
\end{align*}
\]
The type-free Logic for Computable Functions

\[ \begin{align*}
\lambda : & \equiv [\lambda x, b \cdot x \cdot n_1(x) \Rightarrow \text{pair}(b, a(n_2 x))], \text{pair}(\#, b(n_2 x))] \text{ disjoint union} \\
\text{id} : & \equiv [\lambda x. x] \quad \text{identity function} \\
u : & \equiv [\lambda x, y. x \Rightarrow x, y] \quad \text{join} \\
\land : & \equiv [\lambda x, y. \theta(x) \vee \theta(y) \Rightarrow x \Rightarrow y, \#] \quad \text{logical and} \\
\lor : & \equiv [\lambda x, y. \theta(x) \vee \theta(y) \Rightarrow x \Rightarrow \#, y] \quad \text{logical or} \\
\neg : & \equiv [\lambda x. x \Rightarrow \#, \#] \quad \text{negation}
\end{align*} \]

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

The standard properties of these functions are easily derivable; for example,
\[ \forall x, y. n_1(\langle x,y \rangle) = x \quad \forall x, y. n_2(\langle x,y \rangle) = y \]

Milner and Weyhrauch have shown that \(u\) has all the properties of the join operation in a lattice, also with respect to the partial order defined by \(\leq\). In particular, \(tt \sqcup \# \sqcup x \sqcup \top \sqcup \#\) for all \(x \leq 1\).

A strict conditional \(\Rightarrow\), i.e. \(T : \Rightarrow x, y : T\) for all \(x\) and \(y\), is definable in terms of the normal conditional \(\Rightarrow\).
\[ \Rightarrow : \equiv \lambda z. x, y : z : (z \Rightarrow x, \top), (z \Rightarrow \top, y) \]

Since the normal conditional will not be used in this paper except in the join operation, we will henceforth use the character \(\Rightarrow\) 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 \(x \lor y \Rightarrow \neg (\neg x \land \neg y)\) still hold.

2.2 Retracts, Domains, Types

The type-free logic essentially axiomatizes the "universal" domain \(I\). 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 \(f : I\) with \(f \circ f = f\). 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 type-free Logic for Computable Functions

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 means "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 \( f : I \) is a morphism from the retraction \( r \) to the retraction \( s \) iff \( f \circ r = s \).
- 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 \( R \) iff they agree on their source domain ("source retract"), i.e. if \( f \circ r = g \circ r \). An identity on a retraction \( r \) is a function \( F \) with \( F \circ r = r \). We write \( I d_r \) for the identity on \( r \).

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

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

\[
T = [\lambda x. x \approx \#] \\
I = [\lambda x. T \cup (J \to J)]
\]

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 retractions; different retractions can define the same domain but will be different objects in \( R \). For example, the retractions \( T \) and \( \text{istrue} \) (see appendix) both define the domain \( \{\#, \text{true}, \text{false}, T\} \) but are completely different functions. However, \( T \) and \( \text{istrue} \) 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

\[
T \circ \text{istrue} = \text{istrue} \quad \text{and} \quad \text{istrue} \circ T = T
\]

i.e. \( T \) and \( \text{istrue} \) are subdomains of each other, but

\[
\text{istrue} \circ T \neq \text{istrue} \quad \text{and} \quad T \circ \text{istrue} \neq T
\]
Thus neither of \( T \) and \( \text{istrue} \) 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 \( D = D_a \) then \( r \) and \( s \) are isomorphic, that is, from a structural point of view, they cannot be distinguished.

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 \( a + b \), \( a \times b \), and \( a \rightarrow b \).

**Proof:** by T.P.

**Lemma 2.3:** \( R \) is cartesian-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

\[
F = [\lambda f x. (r_1 x)(r_2 x)] \\
G = [\lambda g r s. g(r, s)]
\]

The T.P. proofs for

\[ G \circ F = \text{Id}_{[R \rightarrow (S, P)]]} \]

and

\[ F \circ G = \text{Id}_{[R \times S \rightarrow P]} \]

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 \( \text{str} \) be defined by

\[
\text{str} := [\lambda f x. \delta(x) \Rightarrow f(x), \bot]
\]

By T.P. we can show that \( \text{str} \) 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 \( \bot \) and \( T \), only pairwise incomparable elements. For flat domains there is a computable equality relation "\( = \)" with:

\[
\delta(x) = \bot, \quad \delta(y) = \bot, \quad x = y \quad \Rightarrow \quad x \text{eq} = \bot \\
x \text{eq} = \bot \\
x \text{eq} \quad y \quad \Rightarrow \quad x = y, \quad \delta(x) = \bot, \quad \delta(y) = \bot
\]
In many cases it is very convenient to use the (computable) equality instead of the equivalence ≡ since it may appear inside a term and thus gives greater expressive power.

Lemma 2.4: If \( F = [\lambda e. l(e)]) \) and \( G = [\lambda e. g(e)]) \), 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 \cite{Mc}, using constructors, selectors and recognizers 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):

\[
\begin{align*}
\text{type_def} & \leftarrow \text{name} \rightarrow \text{subtype} \mid \text{comptype} \\
\text{subtype} & \leftarrow \text{constant} \mid \text{comptype} \\
\text{comptype} & \leftarrow \text{constr} ( \text{sel} \mid : \text{dt} \mid ) \\
\text{constant} & \leftarrow \text{identifier}
\end{align*}
\]

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, 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

\[
\text{Seq} \leftarrow \text{emptyseq} \mid \text{mseq}(\text{hd:atom}, \text{tl:Seq}).
\]

Strictly speaking, this data type definition is a type scheme, that is,
Abstract Data Types in the Type Free Logic

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 retraction.

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

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

\[ \text{type} ::= \text{constant}_1 | \ldots | \text{constant}_m | \text{comptype}_1 | \ldots | \text{comptype}_n \]

with

\[ \text{comptype}_k ::= \text{comp}_n(\text{sel}_1; \text{dt}_1, \ldots, \text{sel}_k; \text{dt}_k) \quad \text{for } k=1, \ldots, n \]

is considered to correspond to the fixed point equation

\[ \text{type} = [\langle \lambda \text{F}(x) \mid \text{is_constant}_1(x) = x, \ldots, \text{is_constant}_m(x) = x, \text{is_comptype}_1(x) = \text{comp}_1(\text{dt}_1, \text{sel}_1(x)), \ldots, \text{dt}_n(\text{sel}_n(x)) \rangle, \ldots, \text{is_comptype}_n(x) = \text{comp}_n(\text{dt}_n, \text{sel}_n(x)), \ldots, \text{dt}_n(\text{sel}_n(x)) \rangle] \]

where \( \text{dt}_i F \) if \( \text{dt}_i = \text{type} \) and \( \text{dt}_i \neq \text{dt}_j \) otherwise, and for \( i,j = 1, \ldots, m; k = 1, \ldots, n \)
Abstract Data Types in the Type Free Logic

(2) \( \delta(\text{constant}) = \text{tt} \)
    \( \text{is}\_\text{constant}(\text{constant}) = \text{tt} \)

(3) \( \delta(\text{comp},(\text{dt}_{k_1}(x_1), \ldots, \text{dt}_{k_n}(x_n))) = \delta(\text{dt}_{k_1}(x_1)) \land \ldots \lor \delta(\text{dt}_{k_n}(x_n)) \)
    \( \delta \circ \text{comp} = \text{is}\_\text{comp} \circ \text{comp} \)

(4) \( \forall x. \text{is}\_\text{comp}_k(x) \Rightarrow \text{is}\_\text{comp}(x) = \text{ff} \)
    \( \forall x. \text{is}\_\text{const}_i(x) \Rightarrow \text{is}\_\text{const}(x) = \text{ff} \)
    \( \forall x. \text{is}\_\text{const}_i(x) \Rightarrow \text{is}\_\text{comp}_n(x) = \text{ff} \)

(5) \( \delta \circ \text{is}\_\text{comp}_k = \delta \circ \text{is}\_\text{comp} \)
    \( \delta \circ \text{is}\_\text{const} = \delta \circ \text{is}\_\text{const} \)
    \( \delta \circ \text{is}\_\text{const} = \delta \circ \text{is}\_\text{comp}_k \)

(6) \( \delta(\text{comp},(x_1, \ldots, x_n)) = \text{sel}_r(\text{comp}(x_1, \ldots, x_n)) = x_r \)
    \( \text{for } r=1, \ldots,n \)

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

\[ \text{Seq} ::= \text{emptyseq} \mid \text{mkseq(hd:Atom, tl:Seq)} \]

will generate the axioms

(51) \( \forall x. \text{is}\_\text{emptyseq}(x) \Rightarrow \text{emptyseq}, \)
    \( \text{is}\_\text{mkseq}(x) \Rightarrow \text{mkseq} \text{atom(hd(x)), S(tl(x)))}, \)

where \text{atom} is the characterizing function for the data type \text{Atom},

(52) \( \delta(\text{emptyseq}) = \text{tt} \)

(53) \( \text{is}\_\text{emptyseq}(\text{emptyseq}) = \text{tt} \)

(54) \( \forall x y. \delta(\text{mkseq(Atom(x), seq(y))) = \delta(\text{atom(x)} \land \delta(\text{seq(y))}) \)

(55) \( \delta \circ \text{mkseq} = \text{is}\_\text{mkseq} \circ \text{mkseq} \)

(56) \( \text{is}\_\text{emptyseq} = \lor \circ \text{is}\_\text{mkseq} \)

(57) \( \forall x y. \delta(\text{mkseq(x,y)}) \Rightarrow \text{hd(mkseq(x,y)) = x} \)

(58) \( \forall x y. \delta(\text{mkseq(x,y)}) \Rightarrow \text{tl(mkseq(x,y)) = y} \)

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, \( t \), itself may be a base type for \( t \), (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 \( \text{typ} \) is \( \text{typ} \). 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 \( \text{typ} \) 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 \( \times \); see [Sc2]). For example, the data type \( \text{Seq} \) of sequences of atoms is completely specified by the least fixed point of the equation

\[
(i) \quad \text{Seq} = \text{emptyseq} :+ (\text{Atom} :\times \text{Seq})
\]

(where \( :+ \) and \( :\times \) are strict versions of \( + \) and \( \times \)) given a representation of the type \( \text{Atom} \) and the constant \( \text{emptyseq} \). 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 \( \text{Seq} \) (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

\[
\text{type} := \text{constant}_1 | \ldots | \text{constant}_n | \text{comptype}_1 | \ldots | \text{comptype}_n
\]

with

\[
\text{comptype}_k := \text{comp}_k(\text{sel}_1^{\text{dt}_{k_1}} \ldots \text{sel}_{k_k}^{\text{dt}_{k_k}}) \quad \text{for} \quad k = 1, \ldots, n
\]

is simply translated into

\[
\text{type} := \text{id} | \ldots | \text{id} | \text{id} \times \text{comp}_1 | \ldots | \text{id} \times \text{comp}_n
\]

with

\[
\text{comp}_k := (\text{dt}_{k_1} \times \ldots \times \text{dt}_{k_k}) \quad \text{for} \quad k = 1, \ldots, n
\]

It is easy to figure out what the primitives look like, and the proof that the axioms of definition 3.1 hold is straightforward.
Abstract Data Types in the Type Free Logic

**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 \( \text{tt} \) exactly for the defined elements of the type, i.e. which satisfies:

\[
\begin{align*}
\forall x. \text{is_type}(x) & \Rightarrow \text{type}(x) = x \\
\forall x. \text{is_type}(x) & \Rightarrow \delta(x) = \text{tt}
\end{align*}
\]

and

\[
\text{type}(x) = x, \quad \delta(x) = \text{tt} \quad \vdash \quad \text{is_type}(x) = \text{tt}
\]

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 type defined by

\[
\text{type} := \text{constant}_1 | \ldots | \text{constant}_m | \text{comptype}_1 | \ldots | \text{comptype}_n
\]

with

\[
\text{comptype}_k := \text{comptyp}_1(s_1; d_{k_1}, \ldots, s_{k_2}; d_{k_2}) \quad \text{for} \quad k = 1, \ldots, n
\]

there is an induction rule that allows to conclude

\[
Q \vdash \forall x. \text{is_type}(x) \Rightarrow P(x)
\]

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

(a) For each constant constant\(_i\) (\( i = 1, \ldots, m \))

\[
Q \vdash P(\text{constant}_i)
\]
Abstract Data Types in the Type Free Logic

(b) For each composed subtype \( \text{comptype}_k \) \((k=1,\ldots,n)\)

\[ Q, \; P(y_{k_1}, \ldots, \sigma(\text{comptype}_k(\ldots))) \vdash \neg P(\text{comptype}_k(\ldots)) \]

with an antecedent \( P(y_{k_1}) \) for each recursion arguments, i.e. for those arguments of \( \text{comptype}_k \) with \( \sigma_k \neq \text{type} \).

Proof: see appendix A.3.

For example, the induction rule corresponding to the characterizing function \( \text{seq} \) is

\[
\frac{\forall x. \text{is_seq}(x) \Rightarrow P(x)}{P(\text{emptyseq}) \quad P(y), \; \delta(\text{mkseq}(x,y)) \Rightarrow \neg P(\text{mkseq}(x,y))}
\]

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 \( \text{seq} \) is a type scheme, defining a data type for any type \( \text{dtype} \). This means that in the corresponding retraction \( \text{seq} \) the retract atom can be replaced by any other retract. We therefore can define the functional

\[
\text{seqof} := \lambda \text{type}. \left( \lambda x. \begin{array}{l}
\text{is_emptyseq}(x) \Rightarrow \text{emptyseq}, \\
\text{is_mkseq}(x) \Rightarrow \text{mkseq}(\text{type}(\text{hd}(x)), \text{S}(\text{tl}(x))), \\
\lambda \end{array} \right)
\]

By theorem 3.3 \( \text{seqof}(\text{type}) \) is a retract for any retract type. In other word, \( \text{seqof} \) 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 [Bl]; see also Higgins [Hi]).

Definition 3.5 (Heterogeneous algebra)

A heterogeneous algebra \( A \) consists of
- a family \( \{A_i\}_{i \in J} \) of (non-empty) sets; the \( A_i \) are called phyla.
- a family \( \{f_k\}_{k \in \mathbb{N}} \) of operations; for each \( f_k \) there is an associated tuple \( \text{index}_k = (i_{k1}, \ldots, i_{km_k}, j_{hm_k}) \) 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 $A = ((A^j, (f_k)_{k\in K})$ and $B = ((B^j, (g_k)_{k\in K})$ be similar algebras. A homomorphism $h$ from $A$ to $B$ is a family of mappings $(h^j)_{k\in K}$ such that $h_j$ maps $A_j$ into $B_j$, and for each $k \in K$

$$h_{j\in K}((f_k(a_1, \ldots, a_{nk}))) = g_k(h_{j_1}(a_1), \ldots, h_{j_{nk}}(a_{nk}))$$

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 $\bot$ 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

- $\text{emptyseq} : 0 \to \text{seq}$ (nullary)
- $\text{mkseq} : \text{atom} \times \text{seq} \to \text{seq}$

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 (i.e. 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

\[ \text{Sfun} := [\lambda f \text{ const op. } \in S. [\lambda x. \text{ is_emptyseq}(x) \Rightarrow \text{const}, \text{is_mkseq}(x) \Rightarrow \text{op}(f(hd(x)), S(tl(x)))], \quad ] \]

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

**Theorem 3.7:** \( \text{F} := \text{Sfun}(\text{fun},c,\text{op}) \) is the unique homomorphic extension of \( \text{fun} \) with respect to \( c \) and \( \text{op} \), i.e., it is the only homomorphism from seq to \( R \) with \( \text{F}(\text{emptyseq}) = c \) and \( \text{F} \circ \text{mkseq} = [\lambda x. \text{op}(\text{fun}(x), \text{F}(y))]. \)

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 \( \text{is_seq} : \text{seq} \rightarrow T \) with

\[ \text{is_seq}(x) \equiv \text{tt} \quad \text{iff} \quad \text{seq}(x) = x \quad \text{and} \quad \text{\&}(x) = \text{tt} \]

is definable simply by extending the type predicate \( \text{is_atom} \) of the generating base type to a homomorphism into \( T \):

\[ \text{is_seq} = \text{Sfun}(\text{is_atom}, \text{tt}, \text{\&}) \]

A further property of the homomorphic extension functional \( \text{Sfun} \) is that it carries over a structural induction rule from the source domain. E.g. for sequences:

\[ P(c) \quad P(y), \text{\&}(\text{op}(x,y)) = \text{tt} \vdash P(\text{op}(x,y)) \]

\[ \forall z. \text{is_seq}(z) \Rightarrow P(\text{F}(z)) \]
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 \((\text{Sdom}, \text{Sfun})\) defines a functor from \(\mathcal{R}\) into a subcategory of \(\mathcal{R}\) 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 \([\text{Mo}, \text{Mi}W]\). 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 \(\text{norep}(x)\) be a predicate on sequences which is true iff \(x\) does not contain repetitions of elements (the explicit definition is straightforward). Then the data type \(\text{norepseq}\) of "sequences without repetitions" is just the restriction of \(\text{seq}\) by \(\text{norep}\). 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 \(\text{mkseq}\) is modified to

\[
\text{mknorepseq} := (\lambda x y. \text{norep} (\text{mkseq} (x,y)) \Rightarrow \text{mkseq} (x,y), y)
\]

then all sequences constructed by \(\text{mknorepseq}\) have the "no-repetition" property, i.e. they satisfy the predicate \(\text{norep}\). This is just another application of the homomorphic extension functional: The range of the function
Abstract Data Types in the Type Free Logic

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 1, which means that norepseq makes sense (in the present context) as a data type. This
construction for new retracts works at least 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 conjectured 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 Postfix 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 := mkexp(letterm) | mkseq(exp_x, exp_y, term)
term := mkterm(flact) | mkplerm(pr_1, term, pr_2, fact)
fact := mkvact(fvar) | mkfact(leterm)

and

post := mkvpost(pvar) | mksum(s, post, p^2) | mkprod(p, post, p^2, post)

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

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

and the postfix grammar

<post> := <var> | <post> <post> ' + ' | <post> <post> ' - '

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

\[
\begin{align*}
\text{mksexp} & : \exp \times \text{term} \to \exp \leftrightarrow \text{mksum} : \text{post} \times \text{post} \to \text{post} \\
\text{mkpterm} & : \text{term} \times \text{fact} \to \text{term} \leftrightarrow \text{mkprod} : \text{post} \times \text{post} \to \text{post} \\
\text{mkvfact} & : \text{var} \to \text{fact} \leftrightarrow \text{mkvpost} : \text{var} \to \text{post}
\end{align*}
\]

i.e., they must satisfy equivalences

\[
\text{Texp}(\text{mksexp}(x,y)) = \text{mksum}(\text{Texp}(x), \text{Tterm}(y))
\]

e tc. Since the distinction between \(\exp\), \(\text{term}\) and \(\text{fact}\) disappears in \(\text{Post}\), the "operations" corresponding to \(\text{mkexp}\), \(\text{mkterm}\) and \(\text{mkfact}\) are just identities on \(\text{post}\). Having established all the algebraic correspondences, homomorphic extension immediately yields the desired functions (slightly simplified):

\[
\begin{align*}
\text{Texp} & = [\lambda x. \text{isExp}(x) \Rightarrow \text{Tterm}((\text{te}(x))), \\
& \quad \text{isExp}(x) \Rightarrow \text{mksum}(\text{E}(\text{su}_1(x)), \text{Tterm}(\text{su}_2(x))), \\
& \quad \bot)] \\
\text{Tterm} & = [\lambda x. \text{isTerm}(x) \Rightarrow \text{Tfact}((\text{tf}(x))), \\
& \quad \text{isTerm}(x) \Rightarrow \text{mkprod}(\text{F}(\text{pr}_1(x)), \text{Tfact}((\text{pr}_2(x))), \\
& \quad \bot)] \\
\text{Tfact} & = [\lambda x. \text{isFact}(x) \Rightarrow \text{mkvpost}((\text{vfv}(x))), \\
& \quad \text{isFact}(x) \Rightarrow \text{Texp}((\text{e}(x))), \\
& \quad \bot]
\end{align*}
\]
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. It 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) T is a type.
(2) Data types are types.
(3) If t, and t_1 are types, then (t,->t_1) is a type (the type of functions from t to t_1).
(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. [Mil] 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 ⊥, ⊥, #, T are terms of type T.
(2) Any identifier is a term.
(3) If \( \downarrow z_1 \rightarrow t_2 \) and \( \downarrow x_1 \) are terms then \( s(x_1) t_2 \) is a term.
(4) If \( \downarrow x_1 \) is an identifier and \( \downarrow t_2 \) is a term then \( \lambda x_1 s(x_1) t_2 \) is a term.
(5) If \( p : T, q : r \) are terms then \( (p \rightarrow q) : T \) is a term.
(6) If \( \downarrow x_1 \) is an identifier and \( \downarrow t_1 \) a term then \( \downarrow x_1 s(x_1) t_1 \) is a term.
(7) If \( \downarrow x_1 \) is an identifier and \( \downarrow S \) is a set term then \( (x_1 S) : T \) is a term.
(8) If \( \downarrow x_1 \) is an identifier, \( \downarrow S \) is a set term and \( \downarrow P \) a predicate term then \( (\forall x_1 : S. \, P(x_1)) : T \) and \( (\exists x_1 : S. \, P(x_1)) : T \) are truth value terms.

As usual parentheses and brackets can be omitted as long as parsing is unambiguous. The notions \( \text{a} \) and \( \text{w} \) are used as in typed LCF (see appendix A.1.1).

Note that the use of the sign \( \forall \) for quantification in (8) cannot be confused with the use of \( \forall \) in abbreviations for \( \lambda x_1 x_1 \lambda x_1 \). The former always requires a restricting set whereas the \( \text{w} \)-\( \forall \) 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 \( \text{tL} \). 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. [Nel]). 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 sets: 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 \( \pi \) for sequences, yielding \( \text{T} \) if \( x \) occurs in \( S \) and \( \text{I} \) otherwise (if it is defined). It is definable as a homomorphism from \( \text{Seq} \) into \( \text{T} \) by homomorphically extending equality on atoms:

\[
\pi \equiv [\lambda x. \, \text{Seq}([\lambda y. \, x y], \Phi, \Psi)]
\]

Note that \( \pi \) is defined for appropriate types only; if \( x \) does not have the same type as the elements of \( S \), \( \pi \) is undefined, thus also \( x \).

\[
\epsilon := [\lambda x. \, \text{Seq}([\lambda y. \, x y], \# \), \Psi)]]
\]
Using the predicate \( \in \) a function \( U_1 \) on \( \text{atom} \times \text{seq} \) is definable by

\[
U_1 := [\lambda x. S. \ x(S = S, \ \text{mkseq}(x, S))]
\]

\( U_1 \) guarantees that elements already occurring in a sequence will not be added; sequences built up using \( U_1 \) are those directly representing sets. If \( \text{Seq} \) is the data type of sequences of elements of type \( t \), the type \( \text{setof}(t) \) is the image of \( \text{Seq} \) under the homomorphism

\[
\text{set} := \text{Sfun}(\text{id}, \text{emptyseq}, U_1).
\]

Moreover, \( \text{set} \) is a retract on \( \text{seq} \); it defines the same subdomain of \( \text{seq} \) as the function \( \text{no apseq} \) discussed in subsection 3.4.: \( \text{setof}(t) \) corresponds to the subset of sequences without repetitions of elements. (However, it is not a subalgebra of \( \text{seq} \).) From this it follows that functions defined on \( \text{Seq} \) are equally defined on \( \text{setof}(t) \). Furthermore, the (generic) structure of \( \text{Seq} \) can be used for defining functions on \( \text{setof}(t) \). More specifically, we have the embedding \( \text{iset}: \text{setof}(t) \to \text{seq} \) with

\[
\text{set} \circ \text{iset} = \text{id}_{\text{set}}.
\]

Thus, any function \( f: \text{seq} \to \text{D} \) can be restricted to \( \text{set} \) by composing with \( \text{iset} \). 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 \( \text{all} \) and \( \text{exist} \) be defined by

\[
\begin{align*}
\text{all} & := [\lambda P. \text{Sfun}(P, \ \text{if}, \ \wedge)] \\
\text{exist} & := [\lambda P. \text{Sfun}(P, \ \text{if}, \ \vee)]
\end{align*}
\]

Then

\[
\forall x(S. \ P(x)) := \text{all}(P, S)
\]

and

\[
\exists x(S. \ P(x)) := \text{exist}(P, S).
\]

Note that this form of quantification is well-defined if \( S \) and \( P \) are defined; since \( \bot \) 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

\[
S_i \subseteq S_j := \forall x(S_i. \ x(S_j)
\]

and similarly set equality by the "extensionality" property

\[
S_i = S_j := (\forall x(S_i. \ x(S_j)) \wedge (\forall x(S_j. \ x(S_i))
\]

Note again that these relations will be undefined for sets over different types. The empty set is the image under \( \text{set} \) of the empty sequence; we will identify the former with the latter.
The function \( u_1 \), taken as a function from \( \text{selof}(1) \) to \( \text{selof}(1) \), 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 \( U_n \) for \( n \)-ary union (\( n \)-fold composition of \( \cup \)).

Similarly, set intersection and set difference are definable by means of the function \( \text{remove}: \text{selof}(1) \rightarrow \text{selof}(1) \) that removes an element from a set. \( \text{remove} \) is defined by

\[
\text{remove} := (\lambda x s. x \in s) \Rightarrow \text{rem}(x, s)
\]

where \( \text{rem} \) is the endomorphic extension of

\[
\text{rem}_1 := (\lambda x y. x \neq y) \Rightarrow [\{], [\}].
\]

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 \( \forall x \in S \). 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 \( PAQ \) is an admissible set predicate.
3. If \( P \) and \( Q \) are admissible set predicates, then \( P \lor Q \) and \( P \land Q \) are admissible set predicates.

**Lemma 4.4:**

If \( P \) and \( Q \) are admissible set predicates, then

\[
\{ x \mid P(x) \lor Q(x) \} = \{ x \mid P(x) \} \cup \{ x \mid Q(x) \}
\]

and

\[
\{ x \mid P(x) \land Q(x) \} = \{ x \mid P(x) \} \cap \{ x \mid Q(x) \}
\]

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,

\[
\neg ( \exists x \in S. P(x)) = \exists x \in S. \neg P(x)
\]

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
extressions 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 \( t \)
is a base type for a data type \( t \), if it occurs in its definition. For each type \( t \), that is a base type for \( t \),
a function

\[
\text{set_of}_t: t \rightarrow \text{set} (t)
\]

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 \( t \), 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:

\[
\text{bintree} := \text{mkbt}(\text{subAtom}) \mid \text{comp} (\text{first:bintree, sec:bintree})
\]

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

\[
\text{BTend} := [\lambda. \text{subAtom} \Rightarrow \text{mkbt(\text{atom(x)}))}, \text{\lambda. \text{compu(x)} \Rightarrow \text{comp(\text{first(x)}, \text{sec(x)})}})
\]

Now, if we want to solve the problem

"Find a function \( \text{vars} : \text{bintree} \rightarrow \text{bintree} \) such that \( \text{vars} \) replaces all atoms in a binary
tree by their values under the function \( \text{vars} : \text{atom} \rightarrow \text{bintree}, \)"

then a solution is simply

\[
\text{vars} = \text{BTend} (\text{vars})
\]

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
cannot 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
Elements of a Problem Specification Language 4 - 6

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 functionals, like the induction proof rule already mentioned above.

Lemma 4.5:
If the argument functions of an extension functional are strict'total then the resulting function 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 

$$\forall x:S \cdot f(x,z)$$

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

$$\lambda z. [\forall x:S \cdot f(x,z)]$$

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

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

This implies that the applications of the \( f(x,\cdot) \) 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

\[ F(y) = \forall x (S(x), P(x)) \]

where the type of \( y \) is the data type list as defined above and \( S \) the standard set-valued function
\texttt{set_of_atom}. Since \texttt{list} is a recursive type, one recursion is required to compute \( S(y) \) and another one
to compute the quantified expression; but we 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

\[ P = F \circ \text{mkbl} \]

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

\[ F = B_{\text{Thom}}(P, \wedge) \]

where \( B_{\text{Thom}} \) is the homomorphic extension functional for \texttt{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.
An Example: Substitution with \( \alpha \)-Conversion

5. An Example: Substitution with \( \alpha \)-Conversion

5.1 The Data Types

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

\[
\begin{align*}
\text{term} & := \text{mkcterm}(\text{constof}:\text{const}) | \\
& \quad \quad \text{mkvterm}(\text{varof}:\text{var}) | \\
& \quad \quad \text{mkapply}(\text{funeof}:\text{term}, \text{argeof}:\text{term}) | \\
& \quad \quad \text{mklambda}(\text{bvarof}:\text{bvar}, \text{termof}:\text{term}) | \\
& \quad \quad \text{mkmu}(\text{bvarof}:\text{bvar}, \text{termof}:\text{term}) | \\
& \quad \quad \text{mkcond}(\text{condof}:\text{term}, \text{trueof}:\text{term}, \text{falseof}:\text{term}) \\
\text{bvar} & := \text{mkbvar}(\text{varof}:\text{var}) \\
\text{const} & := \text{uu} | \text{tt} | \text{ff} | \text{oo} \\
\text{var} & \text{is taken as basic and not further specified.}
\end{align*}
\]

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 \( \text{term}, \text{bvar}, \text{var}, \) and \( \text{const} \) and operations

\[
\begin{align*}
\text{(op1)} \quad \text{mkcterm}: \text{const} & \rightarrow \text{term} \\
\text{(op2)} \quad \text{mkvterm}: \text{var} & \rightarrow \text{term} \\
\text{(op3)} \quad \text{mkapply}: \text{term} \times \text{term} & \rightarrow \text{term} \\
\text{(op4)} \quad \text{mklambda}: \text{bvar} \times \text{term} & \rightarrow \text{term} \\
\text{(op5)} \quad \text{mkmu}: \text{bvar} \times \text{term} & \rightarrow \text{term} \\
\text{(op6)} \quad \text{mkcond}: \text{term} \times \text{term} \times \text{term} & \rightarrow \text{term} \\
\text{(op7)} \quad \text{mkbvar}: \text{var} & \rightarrow \text{bvar}
\end{align*}
\]

The generating phyla (data types) are \( \text{const} \) and \( \text{var} \). Obviously the phyla \( \text{var} \) and \( \text{bvar} \) are isomorphic; the reason for introducing the data type \( \text{bvar} \) 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:

\[
\text{term} := [\alpha F. [\lambda x. \text{is\_const}(x) = \text{mkcterm}(\text{constof}(x)), \\
\text{is\_mkvterm}(x) = \text{mkvterm}(\text{varof}(x)), \\
\text{is\_mkapply}(x) = \text{mkapply}(\text{funeof}(x), \text{argeof}(x)), \\
\text{is\_mklambda}(x) = \text{mklambda}(\text{bvarof}(x), \text{termof}(x)), \\
\text{is\_mkmu}(x) = \text{mkmu}(\text{bvarof}(x), \text{termof}(x)), \\
\text{is\_mkcond}(x) = \text{mkcond}(\text{condof}(x), \text{trueof}(x), \text{falseof}(x)), \\
\perp]]
\]

\[
\text{bvar} := [\lambda x. \text{mkbvar}(\text{varof}(x))]
\]

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 \( \text{term} \); the last one is to replace \( \text{mkbvar} \). By substituting the characteristic function for \( \text{bvar} \) in \( \text{term} \) 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
\[
\text{termhom} := [\lambda \text{op1 op2 op7. } [\times F. ..]]
\]
and
\[
\text{termend} := [\lambda \text{op1 op2 op7. } [\times F. ..]]
\]
be the functionals for homomorphic and endomorphic extension.

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

Replace any free occurrence of the variable \( v \) in the expression \( \text{term} \) by the term \( t \) after
renaming bound variables in \( \text{term} \) suitably (i.e. so that no free variable in \( t \) will become bound in \( \text{term} \)).

A common notation is \( \text{term}[t/v] \).

What is described above is the basic conversion rule of the \( \lambda \)-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 ambitious 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 \( \text{subst} : \text{var} \times \text{term} \times \text{term} \rightarrow \text{term} \) by stepwise specifying the informal notion in
our language. Let \( \text{subst} \) be defined by

\[
\text{subst} := [\lambda \text{v t a. substfree}(\text{v, t, renamebvar(a, t))}]
\]

where

\[
\text{substfree}(\text{v, t, a}) := \text{"replace all free occurrences of } \text{v in } \text{t by } \text{a"}
\]
\[
\text{renamebvar(a, t)} := \text{"rename bound variables in } \text{t that occur free in } \text{a appropriately"}
\]

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

\[
b_1 := \emptyset
\]
\[
b_2 := [\lambda x. \{\}] \\
b_3 := b_2 := b_5 := U \\
b_6 := U_3 \\
b_7 := [\lambda x. \{\}]
\]

(empty set)

(singleton map)
An Example: Substitution with $\alpha$-Conversion

i.e., $\text{set-of-bvar} = \text{termhom}(b_1 \ldots b_7)$.

Then

$$\text{boundvars} := \text{variso} \circ \text{set-of-bvar}.$$ 

b) free variables in term. The standard function $\text{set-of-vars}$ returns all occurring 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 $\lambda$- and $\alpha$-terms from the type $\text{term}$ we could use a standard set-of-dtype function for defining $\text{freevars}$. Instead, we define it directly as as a homomorphism

$$\text{freevars} : \text{term} \rightarrow \text{setof(var)}.$$ 

Using the set-valued functions

$$\begin{align*}
  f_1 & := \{ \lambda x. \emptyset \} \\
  f_2 & := f_7 := \{ \lambda x. \{ x \} \} \\
  f_3 & := \emptyset \\
  f_4 & := f_5 := \{ \lambda x. y. y \setminus x \} & \text{(set difference)} \\
  f_6 & := \emptyset
\end{align*}$$

the function is definable by

$$\text{freevars} := \text{termhom}(f_1 \ldots f_7).$$

c) Renaming bound variables. We need a function $\text{newvar}$ that "invents" new variables (which do not occur in either $e$ or $t$). Strictly speaking the existence of $\text{newvar}$ 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 $\text{newvar}$ that returns a variable with the property

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

Using this function we can specify renaming of bound variables:

$$\text{renamevar}(e, t) := \text{"rename in } e \text{ each variable that occurs free in } t \text{ and bound in } e\"}$$

formally:

$$\begin{align*}
  \text{renamevar} & := [ \lambda x \ e. [ \forall x \in \text{freevars}(t) \land \text{bvarsin}(e) : \text{rename}(l, x, e)]] \\
  \text{rename} & := [ \lambda l. [ \lambda x. \text{termend}(\text{mkterm, replacevar, mkbvar}, \text{renamevar})]]
\end{align*}$$

where $\text{replacevar}$ denotes the term $[ \lambda z. z \mapsto \text{newvar}(x, e, l), z]$. 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.
An Example: Substitution with \(\alpha\)-Conversion

An Example: Substitution with \(\alpha\)-Conversion

d) \(\text{substfree}\). "Free occurrence" means "not bound", i.e. "not in the range of a \(\lambda\) or \(\alpha\) 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 \(\lambda\) or \(\alpha\) (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:

\[
\text{substfree} := [\alpha S. (\lambda v. [\lambda e. \text{is_mklambda}(e) \Rightarrow \text{varof}(e)v \Rightarrow t, e, \\
(is\_lambda(e)\text{vis\_mu}(e)) \land \text{bvarof}(e)v \Rightarrow e, \\
\text{term}(S(v,t),e)))]
\]

Here \(\text{term}\) is the operator on \(F\) that defines \(\text{term}\), i.e. \(\text{term} = [\alpha F. (\lambda x. \text{term}(F,x))\)]

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

\[
\begin{align*}
\text{subst} & := [\lambda v t e. \text{substfree}(v,t,\text{renamevar}(e,t))] \\
\text{substfree} & := [\alpha S. (\lambda v t. [\lambda e. \text{is_mklambda}(e) \Rightarrow \text{varof}(e)v \Rightarrow t, e, \\
(is\_lambda(e)\text{vis\_mu}(e)) \land \text{bvarof}(e)v \Rightarrow e, \\
\text{term}(S(v,t),e)))] \\
\text{renamevar} & := [\lambda e t. (\forall x \in \text{freesvar}(t) \land \text{bvsvar}(e) : \text{rename}(t,x,e))] \\
\text{rename} & := [\lambda t. (\forall x e. \text{term}(\text{mkterm}, \text{replacevar}, \text{mkbvar}=\text{replacevar}))] \\
\text{replacevar} & := [\lambda. z x \Rightarrow \text{newvar}(x,e,t), z] \\
\text{bvsvar} & := \text{variso} \circ \text{sel-of-bvar} \\
\text{sel-of-bvar} & := \text{term}(\text{mk}(\lambda x.\{\}, \{x\}), U, U, U, U, U, U, U, U, U, U, U) \\
\text{freesvar} & := \text{term}(\text{mk}(\lambda x.\{\}, \{x\}), U, \backslash U, U, U, U, U, U, U, U, U, U)
\end{align*}
\]

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 incorporate all these features and special knowledge about the restricted set calculus into an interactive system for developing programs and proving theorems about them.
References

R. Reference:


References

Appendices

A. Appendices

A.1 Logic for Computable Functions

A.1.1 Syntax of Typed LCF

The following is an extract taken from [Mil].

Types At bottom tr and ind are types. Further if $\alpha_1$ and $\beta_2$ are types then $(\alpha_1 \rightarrow \alpha_2)$ 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 $\beta$.

Terms (metavariables $s,t,s_1,t_1,...$) The following are terms:

1. 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.
2. Applications - $s(t) : \alpha_2$, where $s : \alpha_1 \rightarrow \alpha_2$ and $t : \alpha_1$.
3. Conditionals - $(s \rightarrow t_1,t_2) : \beta$, where $s : \alpha_1 \rightarrow \beta$ and $t_2 : \beta$.
4. $\lambda$-expressions - $[\lambda x.s] : \alpha_1 \rightarrow \alpha_2$, where $x : \alpha_1$ and $s : \alpha_2$.
5. $\alpha$-expressions - $[\alpha x.s] : \beta$, where $x : \beta$.

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

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

denotes the function defined recursively as follows:

$f(x) = \text{if } p(x) \text{ then } f(a(x)) \text{ else } b(x)$.

Constants The identifiers t, # denote truthvalues true and false. \_ denotes the totally undefined object of any type; in particular, the undefined truthvalue.

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

$s \leq t$

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

Well-formed formulae (wffs) (metavariables $P,Q,P_1,Q_1,...$) Wffs are sets of zero or more awffs, written as lists with separating commas. They are interpreted as conjunctions. We use
to abbreviate set, etc.

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 wffs is relaxed in the machine implementation to allow a saving of parentheses and brackets. In addition, we use the abbreviation

\[
\begin{align*}
& f(x,y) \quad \text{for} \quad f(x)[y] \\
& \forall x. t = s \quad \text{for} \quad \lambda x.t = \lambda x.s \\
& p :: q = r \quad \text{for} \quad p \lor q \land p \Rightarrow r, \perp
\end{align*}
\]

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.

\[
\begin{align*}
\text{MAX} & \quad \vdash s \in T \\
\text{COND} & \quad \vdash T \rightarrow s, s = s \\
& \quad \vdash r \rightarrow s, t = T(r) \rightarrow s, t \\
& \quad \vdash T(s)(t) = T(s) \\
& \quad \vdash T = T \rightarrow T \\
& \quad \vdash I = \{ \in, T \cup (J \rightarrow J) \}
\end{align*}
\]

The CASES-rule is changed to

\[
\begin{align*}
P & \vdash Q(\perp/x) \quad P \vdash Q(\mu/x) \quad P \vdash Q(\#/x) \quad P \vdash Q(T/x) \\
\text{CASES} & \quad \vdash Q(T/x)
\end{align*}
\]
Other defined standard terms:

\[ \equiv := \lambda x \, y \, z. \, x \equiv y, \, T, \, z \]

\[ \rightarrow := \lambda x \, y. \, \lambda z. \, y \, z \, z \, z \, x \]
Appendices

A.2 Special Functions in tfLCF

A.2.1 Definedness predicate

We want a predicate \( d \) such that

\[ |- d(x) \Rightarrow \bot, x \Rightarrow \bot, x \Rightarrow \top \]

Define \( d \) by

\[ d = \text{upt} \cup \text{down} \]

where

\[ \text{down} = [\lambda x. x \Rightarrow \bot, \bot] \]

and

\[ \text{upt} = [\lambda P. [\lambda x. (x \Rightarrow \top) \cup P(x(T))]] \quad [\lambda x. \text{upt}(x) \Rightarrow \top, \top] \]

down maps everything to \( \bot \) except \( T \) which goes to \( T \):

\[ \text{down}(x) \Rightarrow T \quad \Rightarrow T(x) \Rightarrow T \quad \Rightarrow x \Rightarrow T \]

upt maps everything to \( \top \) except \( \bot \) which is mapped to \( \bot \). The desired properties of \( d \) 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 \( \top \) and \( \bot \) exactly for the arguments \( \top \) and \( \bot \) 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 \( 1 \).

Definition: istrue = [\( \lambda S. [\lambda x. x \Rightarrow \top \cup S(x(T)), \bot \cup S(x(T))] \)]

It is easy to show by cases that

\[ T \in \text{istrue} \]

Since \( \forall x. T(x) \Rightarrow x \) we also have

\[ T \circ \text{istrue} \subseteq \text{istrue} \]

From the definition follows immediately

\[ \text{istrue}(x) \Rightarrow \top \quad \Rightarrow \text{istrue}(x(T)) \subseteq \text{istrue}(x) \]

\[ \text{istrue}(x) \Rightarrow \bot \quad \Rightarrow \text{istrue}(x(T)) \subseteq \text{istrue}(x) \]
also

(4) \( T(x) = TV \models \text{istrue}(x) = TV \) for \( TV = \bot, T \)

Next we show by induction on istrue

(5) \( \forall x. \text{istrue}(x) \subseteq T(\text{istrue}(x)) \)

I. \( \bot(x) = \ldots \) ok.

II. Assume \( \forall x. S(x) \subseteq T(\text{istrue}(x)) \). We have to show \( x : \mapsto \text{iff } S(x) \supseteq T(\text{istrue}(x)) \).

By cases \( T(x) \):

\( T(x) = \bot \) : trivial
\( T(x) = T \) : implies istrue(x) = T, trivial.

\( T(x) = \text{iff} \):

\[ \begin{align*}
\text{lhs} &= \text{iff } U S(x) = \text{iff } U T(\text{istrue}(x)) \\
&= T(\text{iff } U \text{istrue}(x)) \quad \text{by L54 Mi-We} \\
&= T(\text{istrue}(x)).
\end{align*} \]

With (2) we have shown

(6) \( \text{istrue} = T \circ \text{istrue} \)

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

On the other hand we already mentioned that

(6a) \( \text{istrue}(tv) = tv \)

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

(7) \( \text{istrue} \circ T = T \)

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

\[ \begin{align*}
\text{istrue} \circ \text{istrue} &= \text{istrue} \circ (T \circ \text{istrue}) \\
&= (\text{istrue} \circ T) \circ \text{istrue} \\
&= T \circ \text{istrue} \\
&= \text{istrue}
\end{align*} \]
The ultimate goal is to show

\[(x)\quad \text{istrue}(x) = \texttt{tt} \quad \neg x = \texttt{tt} \]
\[(x)\quad \text{istrue}(x) = \texttt{ff} \quad \neg x = \texttt{ff} \]

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

**Definition:** $\text{uptf} = [\in P. [\lambda x. T(x) \cup P(x(T))]]$

By definition we have

\[(U1)\quad T \subseteq \text{uptf} \]

We prove the following facts about $\text{uptf}$:

\[(U2)\quad x \subseteq \text{uptf} \]
\[(9)\quad \text{istrue} \subseteq \text{uptf} \]
\[(10)\quad \text{istrue}(x) = \texttt{tt} \quad \neg \text{uptf}(x) = \text{istrue}(x) \]

(9) and (10) together show

\[(11)\quad \text{istrue}(x) = \texttt{tt} \quad \neg \text{uptf}(x) = \texttt{tt} \]

With (U2) it follows that

\[(12)\quad x = \texttt{tt} \]

on the other hand, since $\texttt{tt} = T(x) \subseteq x$ we have

\[(13)\quad \text{istrue}(x) = \texttt{tt} \quad \neg x = \texttt{tt} \]

The proof for the corresponding statement for \texttt{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.
Appendices

Recall that \( \text{seq} \) is defined by the retraction

\[
\text{seq} = [\forall S. (\lambda x. \text{is\_emptyseq}(x) \Rightarrow \text{emptyseq}, \\
\quad \text{is\_mkseq}(x) \Rightarrow \text{mkseq}(@\text{atom}(\text{hd}(x)), S(\text{tl}(x))), \\
\quad \bot)]
\]

First, we prove the rule

\[
\begin{align*}
(\alpha_1) & \quad P(\bot) \quad (\alpha_2) \quad P(T) \quad (\alpha_3) \quad P(\text{emptyseq}) \quad (\alpha_4) \quad \forall x. P(y) \mid\mid \forall x. P(\text{mkseq}(x,y)) \\
\hline
\hline
\text{(RI)} & \quad P(\text{seq}(x))
\end{align*}
\]

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

\[
\forall x. P(\text{seq}(x))
\]

from

\[
\forall x. P(\bot x) \quad \text{and} \quad \forall x. P(S x) \mid\mid \forall x. P(\text{tau}(S)(x)).
\]

Since

\[
P(\bot x) \Leftrightarrow P(\bot)
\]

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 \( \text{tau}(S) \) to

\[
\text{tau}(S)(x) = \text{is\_emptyseq}(x) \Rightarrow \text{emptyseq}, \\
\text{is\_mkseq}(x) \Rightarrow \text{mkseq}(\text{atom}(\text{hd}(x)), S(\text{tl}(x))), \\
\bot
\]

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

\[
\begin{align*}
\text{is\_emptyseq}(x) & \quad \bot : \quad \text{tau}(S)(x) = \bot \quad \text{by premise } (\alpha_1) \\
& \quad \text{tt} : \quad \text{emptyseq} \quad \text{by } (\alpha_3) \\
& \quad \text{T} : \quad \text{T} \quad \text{by } (\alpha_2) \\
& \quad \# : \text{case split for } \text{is\_mkseq}(x) : \text{only } \text{tt} \text{ is interesting:} \\
& \quad \quad \quad \text{mkseq}(\text{atom}(y), S(x)) \quad \text{by } (\alpha_4)
\end{align*}
\]

Now, the rule

\[
\begin{align*}
(\beta_1) & \quad P(\text{emptyseq}) \quad (\beta_2) \quad P(y) \mid\mid P(\text{mkseq}(x,y)) \\
\hline
\hline
\forall x. \text{is\_seq}(x) \Rightarrow P(x)
\end{align*}
\]

follows from \((\text{RI})\) by virtue of the facts that the relativizing type predicate eliminates the cases \( \bot \) and \( T \).
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.