Technical Report

UCAM-CL-TR-44
ISSN 1476-2986

Number 44

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Structural induction in LCF

Lawrence Paulson

November 1983

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Structural Induction in LCF

Lawrence Paulson

University of Cambridge

November 1983
Revised February 1984

Abstract

The fixed-point theory of computation can express a variety of recursive data types, including lazy types, conventional first-order (*strict*) types, mutually recursive types, and types with equational constraints. Lazy types contain infinite objects, regarded as the limit of a chain of finite objects. Structural induction for all these types follows from fixed-point induction, though induction for lazy types is only sound for a certain class of formulas.

The paper presents the derivation of structural induction for each type, and justifies the necessary axioms by furnishing models for them. It presents example type definitions of lazy lists, strict lists, syntax trees for expressions, and finite sets. Strict data types are proved to be flat in their partial ordering. Primitive recursion operators are introduced for each type, providing theoretical insights as well as a concise notation for defining total functions.

The research was done using LCF, an interactive theorem-prover for the fixed-point theory. The paper documents the theory of LCF data types, and surveys several LCF proofs involving structural induction. In order to be self-contained, it makes little reference to LCF details and includes a summary of the fixed-point theory.

Table of Contents

Structural Induction in LCF

Lawrence Paulson

University of Cambridge

November 1983
Revised February 1984

1. Introduction

Many computer scientists are aware of the fixed-point theory of computation, due to its importance in denotational semantics [22]. The theory is also good for reasoning about lazy evaluation, unbounded computation, partial functions, and higher-order functions. Unfortunately it remains obscure to most people, both in its foundations and in its relation to the more familiar first-order logic and set theory.

This paper summarizes experience with LCF (Logic for Computable Functions), a theorem-prover for the fixed-point theory. It does not try to motivate the logic itself, but shows how to define data types that allow structural induction [4].

In studying data structures, it is important to abstract away from what particular programming languages provide. The Cartesian product and discriminated union are fundamental data structuring operators. Making this idea more formal, Burstall and Goguen [5] define abstract types as word algebras or initial algebras. This paper considers types of that general form: all values are composed from a set of constructors, each with a given arity (formal parameter list). This includes common types such as lists and the natural numbers; in addition it includes lazy types such as infinite streams. It does not include function types. The fixed-point theory allows them, but apparently without any form of structural induction.

Boyer and Moore's theorem prover [2] accepts data structure definitions, introducing constructors and preparing a structural induction rule. All induction rules appeal to a general principle of well-founded induction. It has proved many difficult theorems, including the unique factorization of natural numbers into primes.

The LCF proof assistant [11,18] provides several packages for defining data types for structural induction. All induction rules appeal to fixed-point induction. LCF allows well-founded induction on similar logical foundations as Boyer and Moore, using natural number induction. A subsequent paper will deal with this complex topic.

This paper surveys the topic of inductive data types, presenting old and new results in a uniform framework and pointing out trouble spots. It tries to bridge the gap between the theory of types and the practice of using them. The remaining sections discuss

(2) the elements of *fixed-point theory*;

(3) *lazy* structures such as infinite lists, the basic Scott derivation [20];

(4) *strict* data structures such as finite lists;

(5) *mutually recursive* structures, such as abstract syntax trees for expressions;

(6) structures satisfying *equational constraints*, such as finite sets;

(7) proving that a strict data type is *flat*;

(8) *models* for lazy structures, strict structures, and structures with both lazy and strict constructors;

(9) defining functions by *primitive recursion*;

(10) *experience* in LCF research projects involving the implementation and use of structural induction.

The exposition is semi-formal. I have not written down the general case of n constructors with their arities, to avoid complex subscripting; the general case should be apparent from the examples. The paper is based on experience with proofs in LCF, but avoids mentioning LCF's commands or implementation. It does require some familiarity with fixed-point theory [1,14].

2. Elements of Fixed-Point Theory

This section is a brief summary of LCF's logic, PPLAMBDA [19]. PPLAMBDA evolved from an early logic of Scott [20], which Igarashi [13] extended to the full predicate calculus. Bird [1] and Manna [14] explain the theory of partial orderings, continuous functions, and fixed points.

PPLAMBDA is a natural deduction logic, with conventional rules for introducing and eliminating connectives [14]. A formula represents a logical sentence, while a term represents a computable value.

2.1. Types and continuous partial orderings

PPLAMBDA is a typed logic; every term has a type. If α and β are types, then other types include

the *function* type $\alpha \to \beta$ of continuous functions from α to β;

the *Cartesian product* type $\alpha \times \beta$ of pairs of elements from α and β,

the primitive type *tr* of the truth values.

You may introduce new types; for example, a later section will define the type (α)*list* for lists with elements of type α. The notation $t:\alpha$ states that the term t belongs to the type α. An operator such as $\to$, $\times$, or *list*, which builds a type from other types, is called a *type operator*.

Every type includes the value $\bot$ (bottom), representing the result of a non-terminating computation. This induces a continuous partial ordering (cpo) on terms, written $t \subset u$ and pronounced "t approximates u." Axioms include

$$\begin{array}{ll} \bot \subset x & (\textit{minimality of } \bot) \\ x \subset x & (\textit{reflexivity}) \\ x \subset y \wedge y \subset x \Rightarrow x \equiv y & (\textit{anti-symmetry}) \\ x \subset y \wedge y \subset z \Rightarrow x \subset z & (\textit{transitivity}) \end{array}$$

Strictly speaking, the expression "$\bot$" is ambiguous; we should always write down the type, "$\bot:\alpha$". However, the types can generally be inferred from the context. LCF performs this automatically, using Milner's type inference algorithm [9].

For logics involving undefined elements, "equivalence" refers to the equality *predicate*, where $\bot \equiv \bot$ is a true formula. The word "equality" is reserved for the computable equality *function*, where $\bot = \bot$ is a term whose value is $\bot$. The axioms imply several lemmas involving equivalence:

$$\begin{array}{ll} x \equiv x & (\textit{reflexivity}) \\ x \equiv y \Rightarrow y \equiv x & (\textit{symmetry}) \\ x \equiv y \wedge y \equiv z \Rightarrow x \equiv z & (\textit{transitivity}) \end{array}$$

2.2. Truth values

The type *tr* includes three distinct truth values: $\bot$ (bottom), TT (true), and FF (false). Axioms are

$$\begin{array}{ll} \forall p:tr\ .\ p \equiv \bot \vee p \equiv TT \vee p \equiv FF & (\textit{case analysis}) \\ TT \not\subset FF \wedge FF \not\subset TT \wedge TT \not\subset \bot \wedge FF \not\subset \bot & (\textit{distinctness}) \end{array}$$

Note: The cases axiom for truth-values implies that every truth-value is either defined or undefined. To intuitionists [10] this is an unacceptable instance of the excluded middle, implying that the halting problem is

decidable. The strong connections between computing and constructive logic [17] indicate the need for a constructive theory of computable functions.

2.3. Cartesian products

The axioms for Cartesian products state that every element, including $\perp$, can be uniquely expressed as a pair. For types α and β, the functions $FST{:}(\alpha{\times}\beta){\to}\alpha$ and $SND{:}(\alpha{\times}\beta){\to}\beta$ select components of pairs.

$$\begin{aligned} \forall xy{:}\alpha{\times}\beta\,.\,(FST\ xy,\ SND\ xy) &\equiv xy \\ FST(x,y) &\equiv x \\ SND(x,y) &\equiv y \end{aligned}$$

2.4. Functions

Functions may be written in lambda notation. If the variable x has type α, and the term t has type β, then the term $\lambda x.t$ has type $\alpha{\to}\beta$. Beta-conversion is an axiom scheme. For a variable x, and terms t and u, where x and u have the same type, let $t[u/x]$ denote the term that results from substituting of u for x in t, renaming bound variables of t to avoid clashes. For every x, t, and u, PPLAMBDA includes the axiom

$$(\lambda x.t)u \equiv t[u/x] \qquad (\textit{beta-conversion})$$

All functions are monotonic and continuous. The partial ordering on a function type is defined using the ordering on the range of the function, using the axioms

$$\begin{aligned} x{\sqsubseteq}y &\Rightarrow fx{\sqsubseteq}fy && (\textit{monotonicity}) \\ (\forall x.fx{\sqsubseteq}gx) &\Leftrightarrow f{\sqsubseteq}g && (\textit{extensionality}) \end{aligned}$$

Lemmas about functions include:

$$\begin{aligned} (\forall x.fx{\equiv}gx) &\Rightarrow f{\equiv}g && (\textit{extensionality}) \\ \lambda x.\perp &\equiv \perp && (\textit{undefined function}) \\ \lambda x.fx &\equiv f && (\textit{eta-conversion}) \end{aligned}$$

Every function f has a least fixed point $FIX\,f$. For every type α, $FIX{:}(\alpha{\to}\alpha){\to}\alpha$ is itself a continuous function in the logic, satisfying the axiom

$$FIX\,f \equiv f\,(FIX\,f)$$

The Fixed-Point Theorem (Kleene) states that FIX is the limit (least upper bound) of a chain of functions:

$$\begin{aligned} FIX\,f &\equiv \lim\{\perp;\ f\perp;\ f\,(f\perp);\ \cdots\} \\ &\equiv \lim_{n\to\infty}\{f^n\perp\} \end{aligned}$$

2.5. Fixed-point induction

The fundamental induction rule of PPLAMBDA is fixed-point induction:

$$\frac{P(\bot) \qquad \forall f.P(f) \Rightarrow P(fun\ f)}{P(FIX\ fun)} \qquad \textit{for chain-complete } P$$

Fixed-point induction on a variable f and formula $P(f)$ is sound whenever P is *chain-complete* with respect to f. For any ascending chain of values $g_1, g_2, \ldots$, if $P(g_i)$ holds for every i, then $P(g)$ must hold for the limit, g. The premises imply that P holds for every member of the chain $\bot$, $fun\,\bot$, $fun\,(fun\,\bot)$, $\cdots$. By chain-completeness, P holds for the limit, which is $FIX\ fun$.

Unfortunately chain-completeness is a *semantic* property, while conducting a formal proof using inference rules is a *syntactic* operation. In Scott's basic logic [20], the only formulas are conjunctions of inequivalences, which are always chain-complete. PPLAMBDA is a full predicate logic; a formula containing implication, negation, existential quantifiers, or predicates may not be chain-complete [1,13].

Both the Edinburgh [11] and Cambridge [19] implementations of LCF restrict fixed-point induction to formulas that satisfy a syntactic test that guarantees chain-completeness. Each test is too complex to describe in less than a page, yet neither accepts all the chain-complete formulas that come up in practice. Section 5, on mutual structural induction, presents such a formula. Thus current implementations of fixed-point induction are overly complex and insufficiently general. A possible solution would be to express and prove chain-completeness within the logic.

Chain-completeness matters even if you never use fixed-point induction directly. LCF derives structural induction from fixed-point induction. Structural induction on lazy types is only sound for chain-complete formulas.

2.6. Sets and flat types

The type *tr* may be thought of as the set $\{TT, FF\}$ of truth values, with $\bot$ adjoined. Other sets, such as the natural numbers $\{0,1,2,3,\cdots\}$, can be taken as primitive types by adjoining $\bot$. Such types have no partially defined elements: if $x \sqsubset y$, then either $x \equiv \bot$ or $x \equiv y$. Any type with this fundamental property is called *flat*, as diagrams of the partial orderings illustrate:

The types $\alpha\times\beta$ and $\alpha\to\beta$ are rarely flat, even if α and β are. For instance, $tr\times tr$ contains the partially defined element $(\bot,TT)$, where

$$(\bot,\bot)\sqsubseteq(\bot,TT)\sqsubseteq(TT,TT).$$

Infinite types such as functions and lazy lists have a complex partial ordering.

You can perform conventional reasoning about sets in PPLAMBDA by using only flat types. Section 8 presents sum and product type operators that preserve flatness; section 7 proves that strict type operators, such as ***list***, preserve flatness. Flat types have two important advantages:

- Structural induction over a flat type is sound for any formula. Since all chains are trivial, every formula is chain-complete.
- The equality function is total only on flat types. It is impossible to decide in finite time whether two infinite lists are equal; this intuition can be made rigorous. Any reasonable equality test must report that $y=y$ is TT, if y is defined. Then for any x that approximates y, monotonicity implies $(x=y)\sqsubseteq(y=y)$. Thus $x=y$ can only be $\bot$ or TT, never FF. This poses no problem in flat types, where x can only be $\bot$ or y.

2.7. Shorthand for defined quantification

Formulas involving flat types often involve quantification over defined values only. In this paper, such formulas are written with the help of abbreviations.

The formula $\forall_D\, x.P$ means the same as $\forall x.x\neq\bot \Rightarrow P$, and may be read, "$P$ holds for all defined x." Several variables may be quantified; for instance, $\forall_D\, x\, y.P$ means the same as $\forall_D\, x.\forall_D\, y.P$.

The formula $\exists_D\, x.P$ means the same as $\exists x.x\neq\bot\wedge P$, and may be read, "$P$ holds for some defined x." For several variables, $\exists_D\, x\, y.P$ means the same as $\exists_D\, x.\ \exists_D\, y.P$.

Unfortunately, current LCF implementations do not provide these quantifiers.

3. Lazy Data Types

Suppose we have a type α, with elements $x_1, \ldots, x_n$, and would like to define lists over α. It is natural to introduce the constructors *NIL* for the empty list, and *CONS* for adding an element to a list. Typical lists are

$$\begin{array}{l} NIL \\ CONS\ x\,(CONS\ x\ NIL) \\ CONS\ x_1\,(CONS\ x_2\,(CONS\ x_3\ NIL)) \end{array}$$

These are all finitely constructed. Induction on a type containing only finite objects is well understood, since it is only a slight generalization of traditional "mathematical induction" on numbers. Unfortunately the discussion of induction rules cannot begin with this easy case. The fixed-point theory is more amenable to defining lazy data types, which contain infinite and partially defined objects in addition to the usual finite ones.

3.1. Lazy lists

The example for this section is lazy lists, similar to those of the programming language KRC [23]. The constructors for lazy lists are, for any type α,

$$\begin{array}{l} LNIL : (\alpha)llist \\ LCONS : \alpha \rightarrow (\alpha)llist \rightarrow (\alpha)llist \end{array}$$

The type $(\alpha)llist$ includes finite lists like the ones above, with no requirement that the elements x_i be defined. There are also infinite lists, informally written as

$$ll_\infty \equiv LCONS\ x_1\,(LCONS\ x_2\,(LCONS\ x_3 \cdots))$$

The " $\cdots$ " indicates that ll_∞ is the limit (least upper bound) of a chain of finite lists ending with $\perp$:

$$\begin{array}{l} ll_0 \equiv \perp \\ ll_1 \equiv LCONS\ x_1\,\perp \\ ll_2 \equiv LCONS\ x_1\,(LCONS\ x_2\,\perp) \\ ll_3 \equiv LCONS\ x_1\,(LCONS\ x_2\,(LCONS\ x_3\,\perp)) \end{array}$$

Though mathematical induction concerns only finite objects, the *lazy induction* rule for $(\alpha)llist$ is sound even for infinite lists:

$$\frac{\begin{array}{c} P(\perp) \\ P(LNIL) \\ \forall x\ ll\ .\ P(ll) \Rightarrow P(LCONS\ x\ ll) \end{array}}{\forall ll\ .\ P(ll)} \qquad \textit{for chain-complete } P$$

For a lazy list of finite construction, the conclusion P holds by a finite number of applications of the $\bot$, *LNIL*, and *LCONS* premisses. Thus P holds for the chain $ll_0, ll_1, ll_2, \cdots$; by chain-completeness, P holds for their limit ll_∞. So P is true for both finite and infinite lazy lists.

3.2. Axioms

Induction proves a property for every element of a type. This is only possible if the elements of the type are sufficiently restricted. The *cases* axiom states that lazy lists are built only from the constructors $\bot$, *LNIL*, and *LCONS*:

$$\forall ll{:}(\alpha)llist \left[\begin{array}{rcl} ll & \equiv & \bot \vee \\ ll & \equiv & LNIL \vee \\ \exists x\ ll'.ll & \equiv & LCONS\ x\ ll' \end{array} \right]$$

Asserting that any infinite list is the limit of a chain of finite lists requires a *copying functional*, defined by cases on the three forms of list:

$$\begin{array}{lcl} LLIST\_FUN\ f\ \bot & \equiv & \bot \\ LLIST\_FUN\ f\ LNIL & \equiv & LNIL \\ LLIST\_FUN\ f\ (LCONS\ x\ ll) & \equiv & LCONS\ x\ (f\ ll) \end{array}$$

Write the function *FIX LLIST_FUN* as *COPY*. The definition of *FIX* implies $COPY \equiv LLIST\_FUN\ COPY$; expanding the above clauses gives

$$\begin{array}{lcl} COPY\ \bot & \equiv & \bot \\ COPY\ LNIL & \equiv & LNIL \\ COPY\ (LCONS\ x\ ll) & \equiv & LCONS\ x\ (COPY\ ll) \end{array}$$

This suggests that *COPY* recursively copies its argument, and should be the identity function for lazy lists. The *reachability* axiom asserts this:

$$FIX\ LLIST\_FUN\ ll\ \equiv\ ll$$

Note that *FIX LLIST_FUN* is the limit, for $n \to \infty$, of $LLIST\_FUN^n \bot$, and that

$$LLIST\_FUN^n \bot\ ll_\infty\ \equiv\ ll_n .$$

An infinite list such as ll_∞, because it equals $FIX\ LLIST\_FUN\ ll_\infty$, is the limit of the finite lists $ll_0, ll_1, ll_2, \cdots$. This is the desired interpretation of infinite structures — they do not exist in their entirety, but may be approximated to any finite degree. Obviously, the operator *llist* does not create flat types!

3.3. Derivation of induction

Most LCF proofs proceed backwards. The statement to be proved, called the *goal*, is reduced using inference rules to simpler and simpler subgoals. The following derivation shows how to reduce the conclusion of the lazy induction rule to its premisses.

Suppose that the property $P(ll)$ is chain-complete for lazy lists ll, and that we would like to prove

$$\forall ll{:}(\alpha)llist\,.\,P(ll).$$

By the reachability axiom, it is enough to show

$$\forall ll\,.\,P(FIX\ LLIST\_FUN\ ll)$$

The next step, fixed-point induction, requires proving that $\forall ll\,.\,P(f\ ll)$ is chain-complete in f. Suppose f is the limit of a chain of functions $f_0, f_1, f_2, \cdots$, and that $\forall ll.P(f_n\ ll)$ holds for all natural numbers n. It suffices to show $P(f\ ll')$ for every ll'. By continuity of function application, the chain of lazy lists $f_0 ll', f_1 ll', f_2 ll', \cdots$ has the limit $f\ ll'$. Since $P(f_n\ ll')$ holds for all n, and $P(ll)$ is chain-complete in ll, the limit $P(f\ ll')$ holds.

Now fixed-point induction gives the two subgoals

$$\forall ll\,.\,P(\bot\ ll) \qquad (\bot\ case)$$

$$(\forall ll\,.\,P(f\ ll)) \Rightarrow (\forall ll\,.\,P(LLIST\_FUN\ f\ ll)) \qquad (step\ case)$$

The $\bot$ case reduces to showing $P(\bot)$, which is the $\bot$ premiss of the lazy induction rule being derived. To prove the step case, assume the antecedent $\forall ll.P(f\ ll)$, and try to prove

$$\forall ll\,.\,P(LLIST\_FUN\ f\ ll).$$

The cases axiom breaks this into three goals, depending on whether ll is $\bot$, *LNIL*, or some *LCONS*:

$$P(LLIST\_FUN\ f\ \bot)$$
$$P(LLIST\_FUN\ f\ LNIL)$$
$$P(LLIST\_FUN\ f\ (LCONS\ x\ ll'))$$

Expanding out the definition of *LLIST_FUN* simplifies these to

$$P(\bot)$$
$$P(LNIL)$$
$$P(LCONS\ x\ (f\ ll'))$$

The $\bot$ and *LNIL* cases have been reduced to the desired form for the lazy induction rule, but the *LCONS* case needs more work. Appeal to the

assumption $\forall ll.P(f\ ll)$, which was set aside earlier. In particular $P(f\ ll')$ is true; making this explicit gives the goal

$$P(f\ ll') \Rightarrow P(LCONS\ x\ (f\ ll'))$$

Since we know nothing about the function f, we might as well remove it from the goal. The term $f\ ll'$, which appears twice, denotes some lazy list. Writing ll for $f\ ll'$, it suffices to prove the more general goal

$$\forall x\ ll\ .\ P(ll) \Rightarrow P(LCONS\ x\ ll)$$

which is the *LCONS* premiss of the lazy induction rule.

3.4. Discussion

Lazy induction dates back to Scott [20], but has never been published in this simple form. One refinement is the cases axiom using disjunction and existential quantifiers. The conventional approach requires a discriminator function *LNULL*, satisfying

$$\begin{array}{ll} LNULL\ \bot & \equiv \bot \\ LNULL\ LNIL & \equiv TT \\ LNULL(LCONS\ x\ ll) & \equiv FF \end{array}$$

In the derivation of induction, this replaces the appeal to the list cases axiom by an appeal to the truth-values cases axiom: consider whether *LNULL* ll returns $\bot$, *TT*, or *FF*. The conventional definition of *LLIST_FUN* is a conditional expression that tests its argument using *LNULL*, and takes it apart using destructor functions *LHEAD* and *LTAIL*. Expanding a call to *LLIST_FUN* requires reasoning about *LNULL*, *LHEAD*, *LTAIL*, and conditionals.

As Burstall [4] argued long ago, discriminator and destructor functions add needless complexity. The cases axiom is simpler than using a discriminator function, and generalizes naturally to larger structures. Milner's data type of trees [8] can be described with the cases axiom

$$\forall t{:}tree \left| \begin{array}{rcl} t & \equiv & \bot \vee \\ t & \equiv & TIP \vee \\ \exists op\ t_1 . t & \equiv & UNARY\ op\ t_1 \vee \\ \exists op\ t_1\ t_2 . t & \equiv & BINARY\ op\ t_1\ t_2 \end{array} \right|$$

The conventional method requires at least two discriminator functions, *IS_TIP* and *IS_UNARY*; uniformity suggests providing also *IS_BINARY*.

For an example proof using induction, consider a function to append two lazy lists:

$$\begin{aligned} &LAPPEND\ \bot\ ll_2 && \equiv \bot \\ &LAPPEND\ LNIL\ ll_2 && \equiv LNIL \\ &LAPPEND(LCONS\ x\ ll_1)\ ll_2 && \equiv LCONS\ x(LAPPEND\ ll_1\ ll_2) \end{aligned}$$

It is easy to prove that *LAPPEND* is associative, by induction on ll_1 in

$$LAPPEND(LAPPEND\ ll_1\ ll_2)ll_3 \equiv LAPPEND\ ll_1(LAPPEND\ ll_2\ ll_3)$$

This formula is chain-complete because it is an equivalence.

As yet there is little experience with infinite data structures. Few theorems have been proved, though Turner [23] and Burge [3] present many programming examples.

4. Strict Data Types

Lazy data types contain infinite objects that are not always wanted. Many simple properties hold only for finite objects. For example, it is straightforward to prove that reversing a finite list twice has no effect,

$$REVERSE(REVERSE\ l) \equiv l.$$

The analgous statement for lazy lists is false [6]; reversing an infinite list results in $\bot$. In their parser proof Cohn and Milner [8] construct a lazy type for parse trees, then use predicates to restrict their theorems to finite trees not containing $\bot$. This proof would be simpler with a type containing only finite objects.

4.1. Finite lists

The example for this section will be the type of ordinary finite lists. Its constructors are

$$\begin{aligned} &NIL : (\alpha)list \\ &CONS : \alpha \rightarrow (\alpha)list \rightarrow (\alpha)list \end{aligned}$$

To exclude partially defined lists, supply axioms stating that the constructors are strict:

$$\begin{aligned} CONS\ \bot\ l &\equiv \bot \\ CONS\ x\ \bot &\equiv \bot \end{aligned}$$

Formulate the cases axiom to avoid overlap between the cases. To be certain that the $CONS\ x\ l$ case is distinct from the $\bot$ case requires considering only defined x and l, which the quantifer $\exists_D$ concisely handles:

$$\forall l{:}(\alpha)list\left[\begin{array}{rcl} l & \equiv & \bot\vee \\ l & \equiv & NIL\vee \\ \exists_D\, x\ l'.\, l & \equiv & CONS\ x\ l' \end{array}\right]$$

The reachability axiom states that any infinite list is the least upper bound of partially defined lists. Since there are no partially defined lists, there are no infinite lists either. As for lazy lists, the axiom requires a copying functional:

$$\begin{array}{rl} LIST\_FUN\ f\ \bot & \equiv \bot \\ LIST\_FUN\ f\ NIL & \equiv NIL \\ \forall_D\, x\ l.\ LIST\_FUN\ f\ (CONS\ x\ l) & \equiv CONS\ x(f\ l) \end{array}$$

In the *CONS* case, the assertions that x and l are defined are essential to avoid contradicting the $\bot$ case. Put *TT* for x, put $\bot$ for l, and put $\lambda l'.NIL$ for f; the potential contradiction is

$$\begin{array}{rcl} LIST\_FUN\ f\ (CONS\ x\ l) & \equiv & LIST\_FUN\ f\ \bot \\ CONS\ TT((\lambda l'.NIL)l) & \equiv & \bot \\ CONS\ TT\ NIL & \equiv & \bot. \end{array}$$

To avoid such contradictions, always insist that x and l be defined whenever talking about $CONS\ x\ l$. This corresponds to the intuition that if a list is finitely constructed, then so are its parts.

4.2. Derivation of induction

In view of the above, the *CONS* premiss of induction should include the assumptions that x and l are defined. The desired rule is

$$\frac{\begin{array}{c} P(\bot) \\ P(NIL) \\ \forall_D\, x\ l.\, P(l) \Rightarrow P(CONS\ x\ l) \end{array}}{\forall l.P(l)}$$

Induction over finite lists is sound for any formula, but the formal derivation requires that $P(l)$ be chain-complete. Section 7, on flatness, shows how to prove that the type operator *list* preserves flatness. If α is flat, then all chains in $(\alpha)list$ are trivial, so every formula about strict lists is chain-complete.

The derivation of induction follows that for lazy lists. The first difference arises in the CONS case, in the goal

$$x \neq \bot \Rightarrow l' \neq \bot \Rightarrow P(f\ l') \Rightarrow P(CONS\ x(f\ l')).$$

The next step is to replace $f\ l'$ by the new variable l. The assertion $l' \neq \bot$ must be discarded. If $f\ l' \neq \bot$ could be proved, that would become $l \neq \bot$ after the substitution, giving the *CONS* premiss for list induction. Unfortunately there is no way to prove this, since we know nothing about the function f. The

resulting goal is

$$x \neq \bot \Rightarrow P(l) \Rightarrow P(CONS\ x\ l).$$

To strengthen this requires an ugly step. By the excluded middle, either $l \equiv \bot$ or $l \neq \bot$. If $l \equiv \bot$, then $P(CONS\ x\ l)$ follows from $P(\bot)$ and strictness of *CONS*. If $l \neq \bot$, then the goal reaches the proper form:

$$x \neq \bot \Rightarrow l \neq \bot \Rightarrow P(l) \Rightarrow P(CONS\ x\ l).$$

Note: The connective $\Rightarrow$ associates to the right; $A \Rightarrow B \Rightarrow C$ means $A \Rightarrow (B \Rightarrow C)$. Such a formula is like a curried function [17], which can be "applied" first to A, then to B.

4.3. Totality of functions producing lists

Strict proofs require careful treatment of termination. Consider the list append function:

$$\begin{aligned} & APPEND\ \bot\ l_2 && \equiv \bot \\ & APPEND\ NIL\ l_2 && \equiv NIL \\ \forall_D\, x\, l.\ & APPEND(CONS\ x\ l)\ l_2 && \equiv CONS\ x(APPEND\ l\ l_2) \end{aligned}$$

Here we could omit the assertions that x and l are defined, since the right side of the *CONS* clause collapses to $\bot$ if either x or l is undefined. But other common functions, like *MAP* below, require the assertions.

The assertions complicate proofs. Consider the associative law for *APPEND*; there is no way to prove the obvious formulation

$$APPEND(APPEND\ l_1\ l_2)l_3 \equiv APPEND\ l_1(APPEND\ l_2\ l_3)$$

Induct on l_1. After a few manipulations, the *CONS* goal becomes

$$APPEND(CONS\ x(APPEND\ l\ l_2))l_3 \equiv CONS\ x(APPEND\ l(APPEND\ l_2\ l_3))$$

Here we are stuck — there is no way to expand the leftmost *APPEND* before proving that $APPEND\ l\ l_2$ is defined. Although the induction rule states that l is defined, there is no assumption about l_2. We must start over, proving the weaker and uglier statement

$$l_2 \neq \bot \Rightarrow APPEND(APPEND\ l_1\ l_2)l_3 \equiv APPEND\ l_1(APPEND\ l_2\ l_3)$$

Beforehand we must prove that *APPEND* is a total function:

$$\forall_D\, l_1\, l_2.\ APPEND\ l_1\ l_2 \neq \bot$$

This is a trivial induction on l_1, but requires axioms stating that *NIL* and *CONS* construct defined lists:

$$NIL \neq \bot$$
$$\forall_D\, x\, l\,.\, CONS\ x\ l \neq \bot$$

When working with strict data types, it is a good idea to prove that any functions you introduce are total. Consider the functional that applies a function to every element of a list:

$$\begin{array}{rll} & MAP\ f\ \bot & \equiv \bot \\ & MAP\ f\ NIL & \equiv NIL \\ \forall_D\, x\, l\,. & MAP\ f\ (CONS\ x\ l) & \equiv CONS\,(f\ x)(MAP\ f\ l) \end{array}$$

It is meaningless to say that a functional is total; however *MAP* preserves totality. If f is a total function, then so is $MAP\ f$. The proof is a simple induction on l:

$$(\forall_D\, x\,.\, f\ x \neq \bot) \Rightarrow \forall_D\, l\,.\, MAP\ f\ l \neq \bot$$

To summarize the difficulties of proving theorems about strict data types: Before expanding a function definition, you must prove that the function's arguments are defined. If these arguments are the results of other functions, then you must prove that these functions are total. If these arguments are simply variables, then you must assume that the variables are defined.

Definedness conditions tend to accumulate; many theorems hold only when all free variables are assumed to be defined. In this situation it is inconvenient to have $\bot$ in the logic. If all of your functions are total, a logic based on conventional set theory would be more appropriate than PPLAMBDA.

5. Mutually Recursive Types

Several data types are *mutually recursive* if an element of one type may contain elements of the others. Abstract syntax trees for a programming language are often mutually recursive. For instance, a declaration may be part of a block, and a block may be part of a procedure, which may be part of a declaration. A variable may be part of an expression, and an expression may be part of a (subscripted) variable. Abstract syntax trees appear in both compilers and proofs.

5.1. Expression trees

The example data type for this section contains trees representing expressions like x or $f\,[x;y]$ or $f\,[g\,[x];g\,[y]]$. For simplicity the constructors will be lazy, allowing expressions such as $f\,[\bot;y]$ and $g\,[g\,[g\,[\cdots]]]$. The strict derivation is similar; add strictness axioms and put definedness assertions before uses of the constructors.

An expression is either a variable or a function applied to a list of expressions. This requires two mutually recursive types, *exp* for expressions and *elist* for expression lists. Suppose that we have a type *var* of variable symbols, and a type *fun* of function symbols. The constructors for type *exp* are

$$VAR : var \to exp$$
$$APPL : fun \to elist \to exp$$

and for the type *elist*,

$$ENIL : elist$$
$$ECONS : exp \to elist \to elist$$

Why not use an existing list type to define expression lists? Then *APPL* would have the type $fun \to (exp)llist \to exp$. The derivation of induction does not work for type definitions with recursion involving another type operator such as *llist*.

After defining the types *exp* and *elist*, you can easily prove that *elist* and $(exp)llist$ are isormorphic. Provide a function *EXPLL* to copy any *elist* as an $(exp)llist$, and a function *ELIST* to copy any $(exp)llist$ as an *elist*. Prove by induction that the two functions are isomorphisms:

$$\forall el{:}elist\,.\,ELIST(EXPLL\ el) \equiv el$$
$$\forall ll{:}(exp)llist\,.\,EXPLL(ELIST\ ll) \equiv ll$$

Then use *EXPLL* and *ELIST* to convert between the types $(exp)llist$ and *elist* as necessary. We can expect future theorem-provers to hide this routine construction.

5.2. An aside: local declarations

Here is a technique for presenting the axioms of mutual recursion in a readable form. Suppose that t is a complex term that appears in several places in the formula $P(t)$. As an informal shorthand, you might choose a new variable x, and write, "let $x{=}t$ in $P(x)$." Formally, it is straightforward to prove that $P(t)$ is logically equivalent to the formula

$$\forall x\,.\,x \equiv t \Rightarrow P(x).$$

Now suppose that the type of t is $\alpha{\times}\beta$; in other words, t denotes some pair. If the variables $x{:}\alpha$ and $y{:}\beta$ do not appear anywhere in $P(t)$, then, because $(FST\,t, SND\,t) \equiv t$, the following formulas are logically equivalent:

$$P(t)$$
$$P(FST\,t, SND\,t)$$
$$\forall x\ y\,.\,(x,y) \equiv t \Rightarrow P(x,y)$$

In words, "let $(x,y)=t$ in $P(x,y)$." Writing (x,y) on the left side of a declaration avoids writing numerous *FST*s and *SND*s in the body of P. This idea has been successful in programming languages. Many implementations of Lisp provide the "destructuring let;" LCF's meta-language ML [11] allows the binding of "varstructs." Note the similarity to defining functions by cases to avoid writing *NULL*, *HEAD*, and *TAIL*. Unfortunately, the use of implication causes problems later on.

5.3. Mutual induction

For a set of mutually recursive types, induction simultaneously proves a property of each type. If $P(e)$ is a proposition for expressions, and $PL(el)$ is one for expression lists, then the induction rule is

$$\frac{\begin{array}{c} P(\bot) \\ \forall v\,.\,P(VAR\ v) \\ \forall fn\ el\,.\,PL(el) \Rightarrow P(APPL\ fn\ el) \\ PL(\bot) \\ PL(ENIL) \\ \forall e\ el\,.\,P(e) \Rightarrow PL(el) \Rightarrow PL(ECONS\ e\ el) \end{array}}{\forall e.P(e) \qquad \forall el.PL(el)} \quad \textit{for chain-complete } P, PL$$

Each mutually recursive type has a separate cases axiom:

$$\forall e{:}exp \left[\begin{array}{rcl} e & \equiv & \bot \vee \\ \exists v\,.\,e & \equiv & VAR\ v \vee \\ \exists fn\ el\,.\,e & \equiv & APPL\ fn\ el \end{array}\right]$$

$$\forall el{:}elist \left[\begin{array}{rcl} el & \equiv & \bot \vee \\ el & \equiv & ENIL \vee \\ \exists x\ el'.el & \equiv & ECONS\ e\ el' \end{array}\right]$$

A single copying functional intertwines the recursive types. For expressions, it maps pairs of functions to pairs of functions, defining copying functions for *exp* and *elist* simultaneously:

$$(g\,,gl) \equiv EXP\_FUN(f\,,fl) \Rightarrow \left[\begin{array}{ll} g\ \bot & \equiv \bot \wedge \\ g(VAR\ v) & \equiv VAR\ v \wedge \\ g(APPL\ fn\ el) & \equiv APPL\ fn(fl\ el) \wedge \\ gl\ \bot & \equiv \bot \wedge \\ gl\ ENIL & \equiv ENIL \wedge \\ gl(ECONS\ e\ el) & \equiv ECONS(f\ e)(fl\ el) \end{array}\right]$$

The reachability axiom states that the fixed-point of *EXP_FUN* is a pair of identity functions:

$$(g\,,gl) \equiv FIX\ EXP\_FUN \Rightarrow (g\ e \equiv e \wedge gl\ el \equiv el)$$

The derivation of the mutual induction rule is similar to that for lazy lists, and should not require detailed commentary. The initial goal:

$$\forall e.P(e)\wedge\forall el.PL(el)$$

Using the axiom of reachability:

$$\forall g\ gl\ .\ (g,gl)\equiv FIX\ EXP\_FUN \Rightarrow (\forall e.P(g\ e)\wedge\forall el.PL(gl\ el))$$

Fixed-point induction produces two goals, with ggl as the induction variable. However, see the later note concerning chain-completeness:

$$\forall g\ gl\ .\ (g,gl)\equiv\perp \Rightarrow (\forall e.P(g\ e)\wedge\forall el.PL(gl\ el))$$

$$\begin{array}{l}(\forall g\ gl\ .\ (g,gl)\equiv ggl \Rightarrow (\forall e.P(g\ e)\wedge\forall el.PL(gl\ el))) \Rightarrow\\ \forall g\ gl\ .\ (g,gl)\equiv EXP\_FUN\ ggl \Rightarrow\\ \forall e.P(g\ e)\wedge\forall el.PL(gl\ el)\end{array}$$

The $\perp$ goal reduces to the $P(\perp)$ and $PL(\perp)$ premisses. The step goal, after specializing g, gl as f, fl in the antecedent, and substituting for ggl, becomes.

$$\begin{array}{l}(\forall e.P(f\ e)\wedge\forall el.PL(fl\ el)) \Rightarrow\\ \forall g\ gl\ .\ (g,gl)\equiv EXP\_FUN(f,fl) \Rightarrow\\ \forall e.P(g\ e)\wedge\forall el.PL(gl\ el)\end{array}$$

Putting aside the antecedents, and considering the separate cases for e and el:

$$\begin{array}{l}P(g\ \perp)\\ P(g\ (VAR\ v))\\ P(g\ (APPL\ fn\ el))\\ PL(gl\ \perp)\\ PL(gl\ ENIL)\\ PL(gl\ (ECONS\ e\ el))\end{array}$$

Substitution in these six goals, using the definition of EXP_FUN:

$$\begin{array}{l}P(\perp)\\ P(VAR\ v)\\ P(APPL\ fn(f\ el))\\ PL(\perp)\\ PL(ENIL)\\ PL(ECONS(f\ e)(fl\ el))\end{array}$$

Only the $APPL$ and $ECONS$ goals need further work. I leave the rest to you; formulate the induction hypotheses $P(e)$ and $P(el)$ using the same argument as for lazy lists.

5.4. Whoops!

There is a nasty problem with the use of fixed-point induction above. The induction formula is chain-complete if P and PL are, but violates most

proposed syntactic tests for chain-completeness [11,13,19]. The induction term *FIX EXP_FUN* appears inside an equivalence in a ***negative position***, the antecedent of an implication. Such formulas are rarely chain-complete.

We can salvage the derivation by extending the chain-completeness test to recognize the use of implications as declarations. But the test already handles a baroque combination of special cases. Or we can recast everything to use *FST* and *SND*, with induction on the chain-complete formula

$$\forall e\,.\,P(FST(FIX\ EXP\_FUN)e)\wedge\forall el\,.\,PL(SND(FIX\ EXP\_FUN)el).$$

The rest of the derivation of induction follows the one above. The *FST*s and *SND*s render the definition of *EXP_FUN* unreadable, with clauses such as

$$FST(EXP\_FUN\ ffl)(APPL\ fn\ el)\ \equiv\ APPL\ fn(SND\ ffl\ el)$$

Clearly we need a better way to establish chain-completeness.

6. Types with Equational Constraints

The data types presented so far have all been word algebras [5]; any structure can be uniquely decomposed. In computing there are many examples of types that satisfy equational constraints, such as commutative and associative laws. PPLAMBDA can express such types, and provide induction rules for them.

6.1. Finite sets

Programmers often use lists where finite sets are called for, so you might imagine that lists can also replace sets in proofs. Such a mistake cost me several months of work. Lists do not enjoy the algebraic laws that hold for the union and intersection of sets. It is true that the *APPEND* of lists resembles union, but *APPEND* is not commutative. A trivial proof about sets can become a long and tedious proof about lists.

The connection between lists and sets is more subtle. As the type $(\alpha)list$ has constructors *NIL* and *CONS*, the type $(\alpha)set$ has constructors

$$EMPTY : (\alpha)set$$
$$INCLUDE : \alpha\rightarrow(\alpha)set\rightarrow(\alpha)set.$$

Here *EMPTY* denotes the empty set ϕ, while *INCLUDE* x s denotes the set $\{x\}\cup s$. Sets satisfy two equations, stating that the multiplicity and order of elements is irrelevant:

$$INCLUDE\ x(INCLUDE\ x\ s)\ \equiv\ INCLUDE\ x\ s$$
$$INCLUDE\ x(INCLUDE\ y\ s)\ \equiv\ INCLUDE\ y(INCLUDE\ x\ s)$$

Note: Equational constraints are only allowed on strict types. There is no way to form equivalence classes of infinite objects; lazy sets make no computational sense.

6.2. Axioms

Sets are finitely constructed; induction should be possible. But we cannot derive induction in the usual way. If we try to turn lists into sets by adding equations, a contradiction arises in the definition of the copying functional, as in section 4. The assertion

$$CONS\ x(CONS\ x\ l) \equiv CONS\ x\ l,$$

along with the definition of *LIST_FUN*, implies

$$\begin{aligned} LIST\_FUN\ f\ (CONS\ TT(CONS\ TT\ NIL)) &\equiv LIST\_FUN\ f\ (CONS\ TT\ NIL) \\ CONS\ TT(f\ (CONS\ TT\ NIL)) &\equiv CONS\ TT(f\ NIL) \end{aligned}$$

Putting *LIST_FUN* $(\lambda y.\bot)$ for f gives:

$$\begin{aligned} CONS\ TT(CONS\ TT\ \bot) &\equiv CONS\ TT\ NIL \\ \bot &\equiv CONS\ TT\ NIL \end{aligned}$$

We can retain consistency by insulating the copying functional from the equations. This example will use the existing type $(\alpha)list$, with its copying functional and induction rule, and impose equations on elements of type $(\alpha)set$. Lists will be regarded as abstract syntax trees for sets. In the general case, you have to define two types: one with equations and one without.

To convert between lists and sets we introduce two functions:

$$\begin{aligned} &SET : (\alpha)list \to (\alpha)set \\ &LIST : (\alpha)set \to (\alpha)list \end{aligned}$$

The function *SET* takes a list $x_1,\ldots,x_n$ of elements, and constructs the set containing them:

$$\begin{aligned} &SET\ \bot && \equiv \bot \\ &SET\ NIL && \equiv EMPTY \\ \forall_D\, x\ l.\ &SET(CONS\ x\ l) && \equiv INCLUDE\ x(SET\ l) \end{aligned}$$

The function *LIST* converts any finite set s to the list of its elements, $x_1,\ldots,x_n$, in arbitrary order. The *denotation axiom* asserts that this list is correct; applying *SET* to it produces the original set s again:

$$\forall s : (\alpha)set\ .\ SET(LIST\ s) \equiv s$$

Consider the relation $=_s$ on lists, where $l_1 =_s l_2$ exactly when $SET\ l_1 \equiv SET\ l_2$. The type $(\alpha)set$ is isomorphic to equivalence classes of $(\alpha)list$

over $=_s$. We do not assert the dual statement $LIST(SET\ l) \equiv l$; this would force *SET* to preserve the structure of its argument, making $(\alpha)set$ isomorphic to $(\alpha)list$.

Since *INCLUDE* is strict, we need axioms of strictness and definedness:

$$INCLUDE\ \bot\ l\ \equiv\ \bot$$
$$INCLUDE\ x\ \bot\ \equiv\ \bot$$

$$EMPTY \neq \bot$$
$$\forall_D\, x\ l\ .\ INCLUDE\ x\ l \neq \bot$$

Clearly *SET* is a total function, by induction on lists. This will be needed later. Remember the advice from section 4 — you often must reason about totality when working with strict types.

Is it reasonable to postulate the function *LIST*, which can enumerate the elements of any set? Yes, if the element type α is a simple type such as the integers; sets of integers may be implemented on a computer as sorted lists. No, if α is a function type. In previous LCF studies, I have allowed sets only if the type α is flat. Better still, there should be some computable total ordering on α. Unfortunately, PPLAMBDA does not handle type conditions gracefully.

6.3. Derivation of induction

The induction rule for finite sets is derived from the one for strict lists:

$$\frac{\begin{array}{c} P(\bot) \\ P(EMPTY) \\ \forall_D\, x\ s\ .\ P(s) \Rightarrow P(INCLUDE\ x\ s) \end{array}}{\forall s.P(s)}$$

Suppose that α is a flat type, and that we would like to prove

$$\forall s : (\alpha)set\ .\ P(s).$$

By the denotation axiom, it is enough to show

$$\forall s\ .\ P(SET(LIST\ s)).$$

This reduces to a more general goal about lists:

$$\forall l\ .\ P(SET\ l)$$

The type $(\alpha)list$ is flat. Thus $P(SET\ l)$ is chain-complete in l; list induction produces the three goals

$$\frac{\begin{array}{c} P(SET\ \bot) \\ P(SET\ NIL) \\ \forall_D\, x\ l\ .\ P(SET\ l) \Rightarrow P(SET(CONS\ x\ l)) \end{array}}{}$$

Expanding with the definition of *SET* gives

$$\begin{array}{l} P(\bot) \\ P(EMPTY) \\ \forall_D\, x\ l\,.\, P(SET\ l) \Rightarrow P(INCLUDE\ x\ (SET\ l)) \end{array}$$

The $\bot$ and *EMPTY* goals are now in proper form for the induction rule; the *INCLUDE* goal needs a little more massaging. Using totality of *SET* on the assumption $l \neq \bot$ gives

$$\forall_D\, x\ l\,.\, SET\ l \neq \bot \Rightarrow P(SET\ l) \Rightarrow P(INCLUDE\ x (SET\ l))$$

Now every occurrence of the list variable l has the form $SET\ l$, which is some defined set. It suffices to prove

$$\forall_D\, x\ s\,.\, P(s) \Rightarrow P(INCLUDE\ x\ s).$$

This is the proper form of the *INCLUDE* premiss of set induction; the derivation is complete.

6.4. Defining functions on sets

This paper defines functions such as *APPEND*, *LLIST_FUN*, and *SET*, in a clausal style, by cases on the possible forms of input. This has the advantage of not requiring discriminator and destructor functions. The risk is that overlapping clauses may contradict each other.

Equations increase this risk. Any function on sets must be consistent with the set equations. Consider the definition of *UNION*, which resembles that of *APPEND*. It is consistent because it handles the *INCLUDE* case using *INCLUDE* itself.

$$\begin{array}{rl} & UNION\ \bot\ s_2 \equiv \bot \\ & UNION\ EMPTY\ s_2 \equiv EMPTY \\ \forall_D\, x\ s\,. & UNION (INCLUDE\ x\ s)\ s_2 \equiv INCLUDE\ x (UNION\ s\ s_2) \end{array}$$

A subtler example is the membership test, which requires a infix function *OR* on truth values:

OR	$\bot$	*TT*	*FF*
$\bot$	$\bot$	$\bot$	$\bot$
TT	$\bot$	*TT*	*TT*
FF	$\bot$	*TT*	*FF*

To test whether z is a member of the set s, compare z with each element of s:

$$\begin{array}{ll} & MEMBER\ z\ \bot & \equiv \bot \\ & MEMBER\ z\ EMPTY & \equiv FF \\ \forall_D\, x\ s\ .\ & MEMBER\ z\ (INCLUDE\ x\ s) & \equiv (z{=}x)\,OR\,(MEMBER\ z\ s) \end{array}$$

The definition is consistent because it handles the *INCLUDE* case using the function $include_M$, defined as

$$include_M\ x\ r\ \equiv\ (z{=}x)\,OR\,r,$$

which satisfies the same equations as *INCLUDE*. Section 9, on primitive recursion, make this idea more precise.

7. Flatness of Strict Types

Lazy data types have a much richer structure than strict ones. Consider lists of zeros and ones. Because strict lists have finite length, they can be embedded into the natural numbers using binary notation. So there are only countably many strict lists of zeros and ones. There are uncountably many lazy lists, since any real number between zero and one generates a distinct lazy list: its infinite binary expansion.

A flat type is one without partially defined elements. There are many partially defined lazy lists, such as $LCONS\ x\ \bot$. There are no partially defined strict lists; $CONS\ x\ \bot$ is (completely) undefined. The type $(\alpha)list$ is flat whenever α is, and this can be proved within the logic.

7.1. Axioms

To prove that a strict type operator like *list* preserves flatness requires axioms stating that values of the type can be uniquely decomposed. The various constructors must be *distinct*:

$$\forall_D\, x\ l\ .\ NIL \neq CONS\ x\ l$$

and *invertible*:

$$\forall_D\, x_1\ l_1\ x_2\ l_2.\ CONS\ x_1\ l_1 \sqsubseteq CONS\ x_2\ l_2 \Rightarrow x_1 \sqsubseteq x_2 \wedge l_1 \sqsubseteq l_2$$

If you introduce the discriminator and destructor functions *NULL*, *HEAD*, and *TAIL*, then distinctness of *NIL* and *CONS* follows from the distinctness of *TT* and *FF*, and invertibility follows from the monotonicity of *HEAD* and *TAIL*.

Distinctness is false for data types that satisfy equations among the constructors. The proof of flatness does not require distinctness, but only a weaker consequence called *maximality*:

$$NIL \sqsubseteq l \Rightarrow l \equiv NIL$$

$$\forall_D\, x_1\, l_1 .\, CONS\ x_1\ l_1 \subseteq l \Rightarrow\ \exists_D\, x_2\, l_2 .\, l \equiv CONS\ x_2\ l_2$$

7.2. Proof for strict lists

Define the type predicate *FLAT* as

$$FLAT(\alpha) \iff \forall xy{:}\alpha .\, x \subseteq y \Rightarrow \bot \equiv x \vee x \equiv y$$

The type $(\alpha)list$ can only be flat if the element type α is, for if $x{:}\alpha$ were partially defined, then so would be $CONS\ x\ NIL$. The desired theorem is

$$FLAT(\alpha) \Rightarrow FLAT((\alpha)list)$$

Assuming $FLAT(\alpha)$, it is enough to show

$$\forall l_1\ l_2{:}(\alpha)list\ .\ l_1 \subseteq l_2 \Rightarrow \bot \equiv l_1 \vee l_1 \equiv l_2$$

This formula is chain-complete in l_1, since the only negative occurrence of l_1 is on the left side of an inequivalence [11]. Structural induction is sound, and produces three subgoals. The $\bot$ goal is trivial:

$$\forall l_2 .\, \bot \subseteq l_2 \Rightarrow \bot \equiv \bot \vee \bot \equiv l_2$$

The *NIL* goal follows from maximality:

$$\forall l_2 .\, NIL \subseteq l_2 \Rightarrow \bot \equiv NIL \vee NIL \equiv l_2$$

The *CONS* goal requires more work. (To avoid ambiguity, I have renamed the variable l_2 as l_3 in the induction hypothesis.)

$$\begin{array}{l} x \not\equiv \bot \Rightarrow l \not\equiv \bot \Rightarrow \\ (\forall l_3 .\, l \subseteq l_3 \Rightarrow \bot \equiv l \vee l \equiv l_3) \Rightarrow \qquad (\textit{induction hypothesis}) \\ \forall l_2 .\, CONS\ x\ l \subseteq l_2 \Rightarrow \bot \equiv CONS\ x\ l \vee CONS\ x\ l \equiv l_2 \end{array}$$

Using the assumptions that x and l are defined and that $CONS\ x\ l \subseteq l_2$, maximality provides defined x' and l' such that $l_2 \equiv CONS\ x'\ l'$. Substitution gives the goal

$$\begin{array}{l} (\forall l_3 .\, l \subseteq l_3 \Rightarrow \bot \equiv l \vee l \equiv l_3) \Rightarrow \\ x' \not\equiv \bot \Rightarrow l' \not\equiv \bot \Rightarrow \\ CONS\ x\ l \subseteq CONS\ x'\ l' \Rightarrow \\ \bot \equiv CONS\ x\ l \vee CONS\ x\ l \equiv CONS\ x'\ l' \end{array}$$

Using the assumption that $CONS\ x\ l \subseteq CONS\ x'\ l'$, invertibility implies that $x \subseteq x'$ and $l \subseteq l'$. Because the type α is flat and $x \subseteq x'$, either $\bot \equiv x$ or $x \equiv x'$:

(1) If $\bot \equiv x$, then the strictness of *CONS* implies the conclusion $\bot \equiv CONS\ x\ l$. (This is the only part of the proof that does not hold for lazy constructors.)

(2) If $x \equiv x'$, then the induction hypothesis implies that either $\bot \equiv l$ or $l \equiv l'$. Again strictness solves the $\bot$ case, leaving the case where $x \equiv x'$ and $l \equiv l'$. Substitution implies the conclusion, $CONS\ x\ l \equiv CONS\ x'\ l'$.

This completes the proof. Though presented here in English, it can be entirely expressed using LCF tactics. You can easily extend it to mutually recursive strict data types; prove simultaneously that all the types are flat.

7.3. Proof for finite sets

An equational type such as $(\alpha)set$ is flat whenever its underlying type, here $(\alpha)list$, is flat. The proof does not require induction.

Suppose that the type α is flat. To prove flatness of $(\alpha)set$, it is enough to show, for all sets s_1 and s_2, that

$$s_1 \subseteq s_2 \Rightarrow \bot \equiv s_1 \vee s_1 \equiv s_2.$$

Suppose that $s_1 \subseteq s_2$. Monotonicity implies $LIST\ s_1 \subseteq LIST\ s_2$. Flatness of lists implies that either $\bot \equiv LIST\ s_1$ or $LIST\ s_1 \equiv LIST\ s_2$.

(1) If $\bot \equiv LIST\ s_1$, then $SET\ \bot \equiv SET(LIST\ s_1)$, implying $\bot \equiv s_1$.

(2) If $LIST\ s_1 \equiv LIST\ s_2$, then $SET(LIST\ s_1) \equiv SET(LIST\ s_2)$, implying $s_1 \equiv s_2$.

That's all there is to it!

8. Models of Recursive Types

The preceeding sections have introduced numerous axioms, asserting cases, strictness, definedness, distinctness, and invertibility. These can be proved as theorems for data types constructed from familiar primitives. (Section 9 will justify the axiom of reachability.)

The strict data types presented above can be expressed as sums of products of themselves (if recursive) and other types. Lazy types require an additional type operator, for delaying the evaluation of arguments of lazy constructors. Types need not be fully strict or fully lazy; constructors can take any combination of strict or lazy arguments.

8.1. Sum types

If α and β are types, then every value of the sum type $\alpha \oplus \beta$ is either $\bot$ or has the form $INL\ x$ or $INR\ y$, for defined $x{:}\alpha$ and $y{:}\beta$. This is a *coalesced* sum — the bottom element of $\alpha \oplus \beta$ is identified with those of α and β. In other words, *INL* and *INR* are strict constructors. The sum operator preserves flatness, since it is a special case of the strict types of section 4. It satisfies

cases,

$$\forall xy{:}\alpha\oplus\beta\ .\ xy\equiv\bot\vee(\ \exists_{\mathrm{D}}\ x{:}\alpha\ .\ xy\equiv INL\ x)\vee(\ \exists_{\mathrm{D}}\ y{:}\beta\ .\ xy\equiv INR\ y)$$

and strictness,

$$\begin{aligned} INL\ \bot\ &\equiv\ \bot \\ INR\ \bot\ &\equiv\ \bot \end{aligned}$$

and also definedness, invertibility, and distinctness.

8.2. Product types

Section 2 presents one product type, the *Cartesian* product used to define mutual recursion. It is superior to other products for this purpose because of two characteristics:

(1) every element is a pair; notably, the bottom element $\bot$ is the pair $(\bot,\bot)$.

(2) every pair (x,y) can be uniquely decomposed to yield x and y again.

The Cartesian pairing constructor (the comma) is neither lazy nor strict. If it were lazy, then (x,y) would differ from $\bot$ for all x and y, violating (1). If it were strict, then $(x,\bot)$ and $(\bot,y)$ would equal $\bot$ for all x and y, violating (2).

The Cartesian product is not suited for constructing models of data types; the *strict* (coalesced) product is better. If α and β are types, then every element of the strict product $\alpha\otimes\beta$ is either $\bot$ or has the form x/y, for defined $x{:}\alpha$ and $y{:}\beta$. The strict product operator preserves flatness, since it is another example of the strict types of section 4. It satisfies a cases axiom,

$$\forall xy{:}\alpha\otimes\beta\ .\ xy\equiv\bot\vee\ \exists_{\mathrm{D}}\ x{:}\alpha\ y{:}\beta\ .\ xy\equiv x/y$$

and strictness axioms,

$$\begin{aligned} \bot/y\ &\equiv\ \bot \\ x/\bot\ &\equiv\ \bot \end{aligned}$$

Another advantage of strict over Cartesian products appears when reasoning about product types in non-lazy programming languages. To represent finite lists of pairs, use the type $(\alpha\otimes\beta)\mathit{list}$; this will be flat if α and β are. The Cartesian product, in $(\alpha\times\beta)\mathit{list}$, will not produce a flat type.

8.3. Lifted types

Making a constructor lazy requires delaying the evaluation of the constructor's arguments. The domain theory models delayed evaluation as a lifted type, consisting of a type with a new bottom element affixed. If α is a type, then every element of the lifted type $(\alpha)u$ is either $\bot$ or has the form $UP\,x$, for some $x{:}\alpha$. Note that $UP\bot$ is partially defined, distinct from $\bot$. A

lifted type is an example of a lazy data type, satisfying the cases axiom

$$\forall ux{:}(\alpha)u \;.\; ux \equiv \bot \vee \exists x{:}\alpha \;.\; ux \equiv UP\,x$$

8.4. The void type

The trivial type *void* contains only the element $\bot$. As *void* has no constructors, it is both strict and lazy. Its cases axiom is $\forall z{:}void.z \equiv \bot$. This type serves as a building block for nullary constructors such as *NIL*.

8.5. Defining a model in LCF

You can model a recursive type as a solution of domain equations involving itself and the type operators above. Scott proved that such solutions exist [22]. The desired ***abstract*** type is defined to be isomorphic to a ***representing*** type involving sums, products, liftings, and *void*. For the strict list type $(\alpha)list$, the representing type is

$$(\alpha)rep \;=\; (void)u \;\oplus\; \alpha \otimes (\alpha)list$$

where $(void)u$ represents *NIL* and $\alpha \otimes (\alpha)list$ represents *CONS*. Assert that the abstract and representing types are isomorphic by declaring the functions

$$ABS\_LIST{:}(\alpha)list \to (\alpha)rep$$
$$REP\_LIST{:}(\alpha)rep \to (\alpha)list$$

and then stating

$$ABS\_LIST(REP\_LIST\ l) \;\equiv\; l$$
$$REP\_LIST(ABS\_LIST\ r) \;\equiv\; r$$

Define the constructors as

$$NIL \;\equiv\; ABS\_LIST(INL(UP()))$$
$$CONS\ x\ l \;\equiv\; ABS\_LIST(INR(x/l))$$

Monotonicity implies that the isomorphisms are strict; strictness implies that they are total. The properties of cases, strictness, definedness, distinctness, and invertibility follow from those for sums and products.

The model for a lazy type is similar, using the lifting operator for all the lazy arguments of constructors. For lazy lists, the representing type is

$$(void)u \;\oplus\; (\alpha)u \otimes ((\alpha)list)u$$

and the lazy version of *CONS* is

$$LCONS\ x\ ll \;\equiv\; ABS\_LLIST(INR((UP\,x)/(UP\,ll)))$$

For mutually recursive types, set up domain equations for each type independently of the others. The solution of these equations simultaneously

provides models for all the types. Cartwright and Donahue [6] discuss models of data types in greater detail.

9. Primitive Recursion and Initiality

A data type constructed from sums and products need not have an induction rule. Previous sections use a ***reachability axiom*** to exclude spurious elements that would invalidate induction. Primitive recursion is a more natural way to obtain induction, revealing a connection between LCF data types and initial algebras.

Primitive recursion is also a concise notation for defining functions. Gordon [12] and Burge [3] give many examples; a few appear below.

9.1. Lazy types

In *constructive type theory* [17], propositions are regarded as types; the type of any term is a formula. Induction is a special case of definition by primitive recursion. PPLAMBDA accomodates this idea. For lazy lists, defining a function h by primitive recursion means stating what h is to compute in the cases where its argument is *LNIL* or *LCONS* x ll; in the *LCONS* case, the value may depend on both x and the recursive call $h\ ll$.

Asserting induction is equivalent to asserting that primitive recursion defines a *unique* function; any two functions that agree on *LNIL* and *LCONS* x ll must agree on all lazy lists. The following axiom, by virtue of the $\Longleftrightarrow$ connective, asserts both existence and uniqueness of primitive recursive functions. The primitive recursive operator, *LLIST_REC*, is a higher-order function; it combines *lnil* and *lcons* to make a function on lazy lists:

$$h \equiv LLIST\_REC(lnil, lcons) \Longleftrightarrow \left[\begin{array}{ll} h\ \bot & \equiv \bot\wedge \\ h\ LNIL & \equiv lnil\wedge \\ \forall x\ ll\ .\ h(LCONS\ x\ ll) & \equiv lcons\ x\ (h\ ll) \end{array}\right]$$

An obvious use of primitive recursion is to concisely define functions such as *LAPPEND*:

$$LAPPEND\ ll_1\ ll_2 \equiv LLIST\_REC(ll_2, LCONS)\ ll_1$$

A more surprising use of primitive recursion is to justify the odd-looking reachability axiom. If *lnil* is *LNIL*, and *lcons* is *LCONS*, then h must satisfy

$$\begin{array}{ll} h\ \bot & \equiv \bot \\ h\ LNIL & \equiv LNIL \\ h(LCONS\ x\ ll) & \equiv LCONS\ x\ (h\ ll) \end{array}$$

Call any function h that satisfies these equations a *copier*. The uniqueness of primitive recursion implies that every copier is equivalent to $LLIST\_REC(LNIL, LCONS)$.

As section 3 points out, the function $FIX\ LLIST\_FUN$ is a copier. So is the identity function I, which for all x satisfies $I\,x \equiv x$. These functions are equivalent,

$$FIX\ LLIST\_FUN \equiv I,$$

implying the reachability axiom.

9.2. Strict types

For strict lists, use the quantifier $\forall_D$; the *CONS* case is only considered for defined arguments:

$$h \equiv LIST\_REC(nil, cons) \iff \begin{bmatrix} h\,\bot & \equiv & \bot\wedge \\ h\ NIL & \equiv & nil\wedge \\ \forall_D x\ l\,.\,h(CONS\ x\ l) & \equiv & cons\ x\,(h\ l) \end{bmatrix}$$

Section 4 suggests that you always prove functions total when working with strict types. This is easy if you define functions by primitive recursion. Prove, by induction on l, that *LIST_REC* produces a total function if *nil* is defined and *cons* is total:

$$(nil \neq \bot \Rightarrow \wedge \forall_D x\ r\,.\,cons\ x\ r \neq \bot) \Rightarrow$$
$$\forall_D l\,.\,LIST\_REC(nil, cons)\,l \neq \bot$$

If you define *MAP* in terms of *LIST_REC*,

$$MAP\ f \equiv LIST\_REC(NIL, \lambda x.CONS(f\ x)),$$

then *MAP* preserves totality because *LIST_REC* does. No induction is needed.

9.3. Mutually recursive types

The primitive recursion functional for the mutually recursive types *exp* and *elist* resembles the copying functional. It defines a pair of functions simultaneously:

$$(h, hl) \equiv EXP\_REC(var, appl, enil, econs) \iff$$
$$\begin{bmatrix} h\,\bot & \equiv & \bot\wedge \\ \forall v\,.\,h(VAR\ v) & \equiv & var\ v\wedge \\ \forall fn\ el\,.\,h(APPL\ fn\ el) & \equiv & appl\ fn(hl\ el)\wedge \\ hl\,\bot & \equiv & \bot\wedge \\ hl\ ENIL & \equiv & enil\wedge \\ \forall e\ el\,.\,hl(ECONS\ e\ el) & \equiv & econs\,(h\ e)(hl\ el) \end{bmatrix}$$

Burge [3, pages 119–124] gives examples of primitive recursive functions on mutually recursive trees and forests.

9.4. Types with equational constraints

To be consistent, primitive recursion must take account of any equations imposed on the type. For a functional *SET_REC* analogous to *LIST_REC*, a function defined as $SET\_REC(empty, include)$ will be consistent if *include* satisfies the same equations as *INCLUDE*:

$$\left[\forall x\ y\ r \begin{bmatrix} include\ x\ (include\ x\ r) & \equiv & include\ x\ r \wedge \\ include\ x\ (include\ y\ r) & \equiv & include\ y\ (include\ x\ r) \end{bmatrix}\right]$$
$$\Rightarrow$$
$$h \equiv SET\_REC(empty, include) \iff \begin{bmatrix} h\ \bot & \equiv \bot \wedge \\ h\ EMPTY & \equiv empty \wedge \\ \forall_D\ x\ s\ .\ h\ (INCLUDE\ x\ s) & \equiv include\ x\ (h\ s) \end{bmatrix}$$

We can now express the set functions *UNION* and *MEMBER* in one line each:

$$UNION\ s_1\ s_2 \equiv SET\_REC(s_2, INCLUDE)\ s_1$$

$$MEMBER\ z\ s \equiv SET\_REC(FF, \lambda x\ r.\ (z = x)\ OR\ r)$$

Note: Thanks to the definedness assertions before *INCLUDE*, the function *include* need not be strict. In fact, it is legitimate to assume that x and y are defined in the equations. The variable r represents the result of h. A more complex axiom for *SET_REC*, with assumptions that *empty* is defined and *include* is total, would guarantee that h was total and r was defined. Such tedious reasoning about totality is necessary, for instance, if *MEMBER* uses a conditional expression instead of the symmetric *OR*. LCF users often wish that $\bot$ were not around.

9.5. Initial algebras in LCF

There is an interesting connection between primitive recursion and initial algebras. Burstall and Goguen [5] describe an initial algebra as having two properties:

no confusion:

Different terms get different values. This is the same as saying that primitive recursive functions exist; it is consistent to specify different values for *NIL* and *CONS* $x\ l$, for each defined x and l.

no junk:

Every element is the value of some term. This is the same as saying that

primitive recursive functions are unique.

More formally, an algebra A is initial if, for any other algebra B with the same signature (the same number and arities of operators), there is a unique homomorphism from A to B. If $(\alpha)list$ is regarded as an algebra A with operators *NIL* and *CONS*, then *nil* and *cons* can be regarded as operators of the algebra B. The unique homomorpism from A to B is simply the unique primitive recursive function determined by *nil* and *cons*.

The connection also holds for data types satisfying equations. There exists a unique homomorphism from an initial algebra A to any other algebra B with the same signature, provided that B also satisfies the equations. The pragmatic, ad-hoc methods of this paper seem to define essentially the same data types as the initial algebra method.

Conventional treatments of initial algebras allow only first-order (strict) data types; emerging higher-order theories of initiality may provide an understanding of lazy data types. But for informal reasoning, I find primitive recursion more natural than homomorphisms.

Note: In constructive type theory [17], the *cons* function for primitive recursion may use the argument l, as well as x and $h\,l$. I prefer that *cons* take only the two arguments x and $h\,l$. This gives *CONS* and *cons* the same arity, allowing the connection with homomorphisms. To define a function h_3 by primitive recursion on nil_3 and on a function $cons_3$, taking the three arguments x, l, and $h\,l$, let

$$\begin{aligned} nil &\equiv (NIL, nil_3) \\ cons\ x\ (l,r) &\equiv (CONS\ x\ l, cons_3\ x\ l\ r). \end{aligned}$$

Then *LIST_REC*(*nil*,*cons*) returns the pair $(l, h_3\,l)$.

10. Experience in Case Studies

LCF is a *programmable* theorem-prover. Its meta-language, ML, is a higher-order functional programming language with a polymorphic type system [11]. By writing ML programs, users have automated routine parts of the theorem-proving process — in particular, the construction of theories of data types and the derivation of structural induction. Many LCF studies involve induction.

Edinburgh LCF users have traditionally defined recursive types in terms of strict sums, Cartesian products, lifted types, and *void*, which are sufficient to define lazy data types like lists [11, pages 79, 87–100]. For historical rather

than theoretical reasons, Edinburgh LCF does not provide disjunction or existential quantifiers. The cases axiom cannot be asserted directly, but its effect can be obtained by defining destructor and discriminator functions. At first the user had to develop new types individually, a tedious and error-prone process. In her proof of a simple compiler, Cohn [7] spent months constructing theories of syntax trees.

Now Milner [8] has implemented a package of ML programs to assert axioms and derive the induction rule for lazy recursive data types, given a description of the constructors. It achieves the effect of a cases axiom by repeated case analysis on the branches of an n-ary sum. This package is tremendously important, not merely for saving labor, but for demonstrating that complex theory construction can be implemented in ML. Many other data type definition packages descended from Milner's. Several projects are using the package. Cohn and Milner [8] have verified a simple parser, and Sokolowski [21] has proved the soundness of Hoare proof rules.

Monahan [16] has implemented strict products in Edinburgh LCF, and has developed a package that allows any combination of strict and Cartesian products in models. Monahan's program can define both strict and lazy types; it also provides discriminator and destructor functions.

I have extended Milner's package to use strict sums and products, with possible lifting of any argument of a constructor function. This allows any combination of strict and lazy constructors, exactly as in Section 8. Furthermore, the program can define mutually recursive types. It unfortunately runs extremely slowly, largely due to the complex inferences needed to simulate the cases axiom from the properties of strict sums and products. I have proved a few simple theorems about substitution using mutually recursive types, but for this purpose a single type is simpler [18].

In developing Cambridge LCF from Edinburgh LCF, I have provided disjunction and existential quantifiers and used them to implement a structure definition package. Like my earlier package, it allows strict and lazy constructors, but not mutual recursion. After constructing any strict type, it proves that the type is flat. Internally it is simpler than the earlier package, and runs much faster. Cambridge LCF does not provide sums, strict products, or lifted types, since these are definable using the package. It still provides Cartesian products.

Using Cambridge LCF, I have performed Manna and Waldinger's verification of the unification algorithm [15], including all the preliminary theorems about

substitution. The proofs involve strict data types for pairs, lists and expressions, developed automatically using the structure package. Equational types for finite sets and substitutions were developed manually, using the methods of section 6. The continual need to reason about totality was most annoying; Boyer and Moore's theorem prover [2] considers totality only when defining a new function. But it is encouraging that such a proof can be completed at all.

There have been few proofs involving lazy types, where totality should not be a problem. Perhaps some of the many projects now under way will shed some light on this topic.

Acknowledgements: I would like to thank M.J.C. Gordon for daily discussions, R. Burstall for some stimulating conversations about initiality, and also J. Goguen, G. Huet, R. Milner, and R. Waldinger.

References

[1] R. Bird, *Programs and Machines: An Introduction to the Theory of Computation*, (Wiley, 1976).

[2] R. Boyer and J. Moore, *A Computational Logic* (Academic Press, 1979).

[3] W.H. Burge, *Recursive Programming Techniques*, Addison-Wesley, 1975.

[4] R.M. Burstall, Proving properties of programs by structural induction, *Computer Journal* **12** (February 1969), pages 41-48.

[5] R.M. Burstall and J.A. Goguen, Algebras, theories and freeness: an introduction for computer scientists, Report CSR-101-82, University of Edinburgh, 1982.

[6] R. Cartwright and J. Donahue, The semantics of lazy (and industrious) evaluation, ACM Symposium on Lisp and Functional Programming (1982), pages 253-264.

[7] A.J. Cohn, *Machine Assisted Proofs of Recurson Implementation*, Report CST-6-79, PhD. Thesis, University of Edinburgh, 1980.

[8] A.J. Cohn and R. Milner, On using Edinburgh LCF to prove the correctness of a parsing algorithm, Report CSR-113-82, University of Edinburgh, 1982.

[9] L. Damas, Principal type-schemes for functional programs, ACM Symposium on Principles of Programming Languages (1982), pages 207-212.

[10] M. Dummet, *Elements of Intuitionism*, Oxford University Press, 1977.

[11] M.J.C. Gordon, R. Milner, and C. Wadsworth, *Edinburgh LCF* (Springer-Verlag, 1979).

[12] M.J.C. Gordon, On the power of list iteration, *Computer Journal* **22** (1979), pages 376-379.

[13] S. Igarashi, Admissibility of fixed-point induction in first order logic of typed theories, Report STAN-CS-72-287, Stanford University, 1972.

[14] Z. Manna, *Mathematical Theory of Computation* (McGraw-Hill, 1974).

[15] Z. Manna and R. Waldinger, Deductive synthesis of the unification algorithm, *Science of Computer Programming* **1** (1981) pages 5-48.

[16] B. Monahan, Forthcoming PhD. thesis, University of Edinburgh.

[17] B. Nordström, Programming in constructive set theory: some examples, ACM conference on Functional Programming Languages and Computer Architecture (1981), pages 141-153.

[18] L. Paulson, Recent developments in LCF: examples of structural induction, Report 34, Computer Laboratory, University of Cambridge (1983).

[19] L. Paulson, The revised logic PPLAMBDA: a reference manual, Report 36, Computer Laboratory, University of Cambridge (1983).

[20] D. Scott, A type-theoretic alternative to CUCH, ISWIM, OWHY, Unpublished notes (1969).

[21] S. Sokołowski, An LCF proof of the soundness of Hoare's logic, Report CSR-146-83, University of Edinburgh, 1983.

[22] J.E. Stoy, *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory*, MIT Press, 1977.

[23] D.A. Turner, Recursion equations as a programming language, in: J. Darlington, P. Henderson, D.A. Turner, editors, *Functional Programming and its Applications* (Cambridge University, 1982), pages 1-28.