Deriving Structural Induction in LCF

Lawrence Paulson
Computer Laboratory
University of Cambridge
Corn Exchange Street
Cambridge CB2 3QG, England

Abstract

The fixed-point theory of computation allows a variety of recursive data structures. Constructor functions may be lazy or strict; types may be mutually recursive and satisfy equational constraints. Structural induction for these types follows from fixed-point induction; induction for lazy types is only sound for a subclass of formulas.

Structural induction is derived and discussed for several types, including lazy lists, finite lists, syntax trees for expressions, and finite sets. Experience with the LCF theorem prover is described.

The paper is a condensation of "Structural Induction in LCF" [12].

1. Introduction

Many computer scientists know of the fixed-point theory of computation, due to its importance in denotational semantics [16]. The theory is also good for reasoning about lazy evaluation, unbounded computation, and higher-order functions. Unfortunately it remains obscure to most people, both in its foundations and in its relation to more familiar logics.

This paper discusses the theory and practice of data types and structural induction [3] in LCF, a theorem prover for the fixed-point theory [7]. It presents old and new results in a uniform framework and points out trouble spots. I explored much of this material while verifying the unification algorithm [13].

Burstall and Goguen [4] define abstract types as initial algebras. This paper considers types of that general form: all values are composed from a set of constructors [2]. This includes lists, the natural numbers, and lazy types such as infinite streams. It does not include function types. The remaining sections discuss

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

(3) *lazy* structures such as infinite lists, the basic Scott derivation [14];

(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) *models* for recursive data types as sums of products;

(8) defining functions by *primitive recursion*;

(9) *experience* in LCF research involving structural induction.

The exposition is semi-formal. I have not written down the general case of n constructors with their argument lists, to avoid complex subscripting. The general case should be apparent from the examples. The paper requires little familiarity with LCF, but some familiarity with fixed-point theory [1,9,16].

2. Elements of Fixed-Point Theory

This section is a brief summary of LCF's logic, PPLAMBDA [11]. PPLAMBDA is a natural deduction logic, with conventional rules for introducing and eliminating connectives [9]. A formula represents a logical sentence, while a term represents a computable value.

2.1. Types and complete partial orderings

Every PPLAMBDA 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, containing the truth values TT and FF.

You may introduce new types; for example, a later section will define the type $(\alpha)list$ 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*.

To represent the result of a non-terminating computation, every type includes the value $\perp$ (bottom), which is the least element under a complete partial ordering, $\subseteq$. The formula $t \subseteq u$ is pronounced "t approximates u."

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

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.

2.2. Functions

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 $(\lambda x.t)u \equiv t[u/x]$, of *beta-conversion*.

All functions are monotonic ($x \subseteq y$ implies $f\ x \subseteq f\ y$) and continuous. The partial ordering on a function type satisfies $f \subseteq g \iff (\forall x. f\ x \subseteq g\ x)$.

Every function f has a least fixed point $FIX\ f$, which is the limit of a chain of functions:

$$FIX\, f \;\equiv\; \lim\{\bot;\, f\bot;\, f\,(f\bot);\; \cdots\} \;\equiv\; \lim_{n\to\infty}\{f^n\bot\}.$$

For every type α, $FIX{:}(\alpha\to\alpha)\to\alpha$ is itself a continuous function, satisfying the axiom $FIX\, f \;\equiv\; f\,(FIX\, f\,)$.

2.3. Fixed-point induction

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

$$\frac{\begin{array}{c}\textit{chain-complete}\ P\\ P(\bot)\quad \forall f.P(f\,) \Rightarrow P(fun\ f\,)\end{array}}{P(FIX\ fun)}$$

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*** process. In Scott's basic logic [14], 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 [1,8].

Both the Edinburgh [7] and Cambridge [11] implementations of LCF restrict fixed-point induction to formulas that satisfy a complex syntactic test for chain-completeness. Deriving mutual induction (section 5) requires a chain-complete formula that both tests reject. A possible solution would be to prove chain-completeness within the logic. Since LCF derives structural induction from fixed-point induction, structural induction on lazy types is only sound for chain-complete formulas.

2.4. 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 types by adjoining $\bot$. These ***flat*** types have no partially defined elements: if $x \sqsubseteq y$, then either $x \equiv \bot$ or $x \equiv y$. PPLAMBDA expresses flatness of the type α as

$$\forall xy{:}\alpha\,.\, x \sqsubseteq y \;\Rightarrow\; \bot \equiv x \vee x \equiv y.$$

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 $(\bot, TT)$. 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, which can be constructed using strict type operators such as *list*, strict product, and strict sum. 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 can only be total for flat types. If $y=y$ is TT for all defined y, then for any x that approximates y, monotonicity implies $(x=y)\subseteq(y=y)$. Thus $x=y$ can only be $\perp$ or TT, never FF. This poses no problem in flat types, where x can only be $\perp$ or y.

2.5. Shorthand for defined quantification

The formula $\forall_D\, x.P$ denotes $\forall x.x\neq\perp \Rightarrow P$, and may be read, "$P$ holds for all defined x." Several variables may be quantified; for instance, $\forall_D\, x\, y.P$ denotes $\forall_D\, x.\forall_D\, y.P$. The formula $\exists_D\, x.P$ denotes $\exists x.x\neq\perp\wedge P$, and may be read, "$P$ holds for some defined x." For several variables, $\exists_D\, x\, y.P$ denotes $\exists_D\, x.\ \exists_D\, y.P$.

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

$$NIL \qquad CONS\ x(CONS\ x\ NIL) \qquad CONS\ x_1(CONS\ x_2(CONS\ x_3\ NIL))$$

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 type for this section is lazy lists. For a type α, constructors are

$$LNIL : (\alpha)llist \qquad LCONS : \alpha\to(\alpha)llist\to(\alpha)llist$$

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 of a chain of finite lists ending with $\perp$:

$$\begin{aligned} ll_0 &\equiv \perp \\ ll_1 &\equiv LCONS\ x_1\perp \\ ll_2 &\equiv LCONS\ x_1(LCONS\ x_2\perp) \end{aligned}$$

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

$$\frac{\overset{\textit{chain-complete } P}{P(\perp) \quad P(LNIL) \quad \forall x\ ll\,.\,P(ll) \Rightarrow P(LCONS\ x\ ll)}}{\forall ll\,.\,P(ll)} \qquad \text{(lazy induction rule)}$$

For a lazy list of finite construction, the conclusion P holds by a finite number of applications of the $\perp$, $LNIL$, and $LCONS$ premisses. Thus P holds for every element of 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

We can formalize these intuitions as axioms, and derive lazy induction as a theorem. I present the axioms and commentary mixed together; for emphasis, each axiom is flagged in the right margin.

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 $\perp$, $LNIL$, and $LCONS$:

$$\forall ll{:}(\alpha)llist\,.\, ll \equiv \perp \vee\ ll \equiv LNIL \ \vee\ \exists x\ ll'.ll \equiv LCONS\ x\ ll' \qquad \text{(cases axiom)}$$

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:

$$\begin{aligned} &LLIST\_FUN\ f\ \perp &&\equiv \perp \\ &LLIST\_FUN\ f\,LNIL &&\equiv LNIL \qquad\qquad \text{(copying functional axiom)} \\ &LLIST\_FUN\ f\,(LCONS\ x\ ll) &&\equiv LCONS\ x(f\ ll) \end{aligned}$$

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

$$COPY \perp \equiv \perp \qquad COPY\ LNIL \equiv LNIL \qquad COPY\,(LCONS\ x\ ll) \equiv LCONS\ x(COPY\ ll)$$

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

$$FIX\ LLIST\_FUN\ ll\ \equiv\ ll \qquad \text{(reachability axiom)}$$

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

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

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

The soundness of induction can now be proved, though only as a meta-theorem. Since PPLAMBDA does not allow quantifiers over propositions, the soundness of an inference rule cannot be stated within the logic.

Theorem. *The cases, copying functional, and reachability axioms imply that the lazy induction rule is sound.*

Proof. In keeping with LCF style, the conclusion of the lazy induction rule will be reduced to its premisses. Suppose that $P(ll)$ is chain-complete for lazy lists ll. It suffices to prove $\forall ll\,.\,P(ll)$. By the reachability axiom, it is enough to prove

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

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 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

$$\forall ll\,.\,P(\bot\ ll) \qquad (\bot \text{ goal})$$

$$(\forall ll\,.\,P(f\ ll)) \Rightarrow (\forall ll\,.\,P(LLIST\_FUN\ f\ ll)). \qquad (\text{step goal})$$

The $\bot$ goal reduces to showing $P(\bot)$, which is the $\bot$ premiss of the lazy induction rule being derived. To prove the step goal, assume the antecedent $\forall ll.P(f\ ll)$, and try to prove $P(LLIST\_FUN\ f\ ll)$. The cases axiom breaks this into three goals, for ll may be $\bot$, $LNIL$, or $LCONS$:

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

Unfolding the definition of $LLIST\_FUN$ simplifies these to

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

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

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

The term $f\ ll'$ denotes some lazy list. Writing ll for $f\ ll'$, it suffices to prove

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

which is the $LCONS$ premiss of the lazy induction rule. ■

3.4. Discussion

Scott derived lazy induction years ago [14], but I have seen no published proof in this simple form. One refinement is the cases axiom using disjunction and existential quantifiers. The conventional approach requires a discriminator function $LNULL$, satisfying

$$LNULL\ \bot \equiv \bot \qquad LNULL\ LNIL \equiv TT \qquad LNULL(LCONS\ x\ ll) \equiv FF$$

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 $\perp$, 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 [3] 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 [6] can be described with the cases axiom

$$\forall t{:}tree\,.\,t\equiv\perp \vee t\equiv TIP \vee \exists op\ t_1 . t\equiv UNARY\ op\ t_1 \vee \exists op\ t_1\ t_2 . t\equiv BINARY\ op\ t_1\ t_2$$

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

4. Strict Data Types

Lazy types contain infinite objects that are not always wanted. For example, reversing a finite list l twice has no effect,

$$REVERSE(REVERSE\ l)\ \equiv\ l,$$

but reversing any infinite lazy list results in $\perp$.

4.1. Finite lists

The example for this section is finite lists, with constructors

$$NIL : (\alpha)list \qquad\qquad CONS : \alpha \to (\alpha)list \to (\alpha)list$$

To exclude partially defined lists, supply ***strictness axioms*** for the constructors:

$$CONS\ \perp\ l\equiv\perp \wedge CONS\ x\ \perp\equiv\perp \qquad\qquad \text{(strictness axiom)}$$

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

$$\forall l{:}(\alpha)list\,.\,l\equiv\perp \vee l\equiv NIL \vee \exists_D\ x\ l'\,.\,l\equiv CONS\ x\ l' \qquad\qquad \text{(cases axiom)}$$

As for lazy lists, induction requires a copying functional:

$$\begin{array}{rll} & LIST\_FUN\ f\ \perp & \equiv \perp \\ & LIST\_FUN\ f\ NIL & \equiv NIL \\ \forall_D\,x\,l. & LIST\_FUN\ f\,(CONS\ x\ l) & \equiv CONS\ x(f\ l) \end{array} \qquad \text{(copying functional axiom)}$$

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

$CONS\ TT\ NIL \equiv CONS\ TT((\lambda l'.NIL)l) \equiv LIST\_FUN\ f\ (CONS\ x\ l) \equiv LIST\_FUN\ f\ \perp \equiv \perp.$

The reachability axiom states that any infinite list is the limit of partially defined lists. Since there are no partially defined lists, there are no infinite lists either:

$$FIX\ LIST\_FUN\ l\ \equiv\ l \qquad \text{(reachability axiom)}$$

4.2. Derivation of induction

The *CONS* premiss of induction includes the assumptions that x and l are defined:

$$\frac{\begin{matrix} chain\text{-}complete\ P \\ P(\perp) \quad P(NIL) \quad \forall_D\, x\ l\,.\, P(l) \Rightarrow P(CONS\ x\ l) \end{matrix}}{\forall l.P(l)} \qquad \text{(strict induction rule)}$$

Theorem. *The strictness, cases, copying functional, and reachability axioms imply that the strict induction rule is sound.*

Proof (sketch). As for lazy lists, it suffices to prove $\forall l.P(l)$ from the premisses of the rule. Apply the reachability axiom and fixed-point induction. The $\perp$ case is easy. Argue by cases in the step case, $P(LIST\_FUN\ f\ l)$. Unfolding the definition of the copying functional reduces the $\perp$ and *NIL* cases to premisses of the list induction rule. The first departure from the proof for lazy lists arises in the *CONS* case:

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

The next step is to replace $f\ l'$ by the new variable l. The assertion $l' \neq \perp$ must be discarded. If $f\ l' \neq \perp$ could be proved, that would become $l \neq \perp$ 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

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

To strengthen this requires further case analysis: either $l \equiv \perp$ or $l \neq \perp$. If $l \equiv \perp$, then $P(CONS\ x\ l)$ follows from $P(\perp)$ and strictness of *CONS*. If $l \neq \perp$, then the goal reaches the proper form:

$$x \neq \perp \Rightarrow l \neq \perp \Rightarrow P(l) \Rightarrow P(CONS\ x\ l).\blacksquare$$

4.3. Totality of functions producing lists

Termination requires careful treatment. Consider the append function:

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

Here we could omit the assertions that x and l are defined, since the right side of the *CONS* clause collapses to $\perp$ if either x or l is undefined. But other common functions require the assertions, which complicate proofs. Consider the associative law for *APPEND*; there is no way to prove the naive statement

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

Try induction 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 allows assuming that l is defined, there is no assumption about l_2. We must start over, proving the weaker and uglier statement

$$l_2 \not\equiv \bot \Longrightarrow APPEND(APPEND\ l_1\ l_2)l_3 \equiv APPEND\ l_1(APPEND\ l_2\ l_3)$$

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

$$\forall_D\ l_1\ l_2 .\ APPEND\ l_1\ l_2 \not\equiv \bot$$

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

$$NIL \not\equiv \bot \ \wedge\ \forall_D\ x\ l .\ CONS\ x\ l \not\equiv \bot \qquad \text{(definedness axiom)}$$

With strict types, it is a good idea to prove that any functions you introduce are total. Consider the functional that applies a function to every list member:

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

It is rarely useful to prove that a functional is total. However, *MAP* preserves totality -- if f is a total function, then so is $MAP\ f$, by induction on l:

$$(\forall_D\ x .\ f\ x \not\equiv \bot) \Longrightarrow \forall_D\ l .\ MAP\ f\ l \not\equiv \bot$$

Many theorems about strict types hold only for defined values.

4.4. Proving flatness

The derivation requires that $P(l)$ be chain-complete, but induction over finite lists is sound for any formula. It is straightforward 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 chain-completeness test of Cambridge LCF recognizes this situation [11]; you can supply it with theorems stating that certain of your types are flat.

To prove in PPLAMBDA that a strict type operator like *list* preserves flatness requires axioms stating that constructions are unique:

$$\forall_D\ x\ l .\ NIL \not\subseteq CONS\ x\ l \ \wedge\ CONS\ x\ l \not\subseteq NIL \qquad \text{(distinctness axiom)}$$

$$\forall_D\ x_1\ l_1\ x_2\ l_2 .\ CONS\ x_1\ l_1 \subseteq CONS\ x_2\ l_2 \Longrightarrow x_1 \subseteq x_2 \wedge l_1 \subseteq l_2 \qquad \text{(invertibility axiom)}$$

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*. 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 $CONS\ x\ NIL$.

Theorem. *If the type α is flat, then the cases, strictness, definedness, distinctness and invertibility axioms, along with list induction, imply that the type* (α)list *is flat.*

Proof (sketch). We may assume that α is flat, and show

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

This formula is chain-complete in l_1, since the only negative occurrence of l_1 is on the left side of an inequivalence [7]. List induction produces three subgoals. The $\bot$ goal is trivial. The *NIL* goal follows by case analysis of l_2, using definedness and distinctness. For *CONS*, similar reasoning provides defined x' and l' such that $l_2 \equiv CONS\ x'\ l'$:

$$\begin{aligned}
&(\forall l_3 .\ l \sqsubseteq l_3 \Rightarrow \bot \equiv l \vee l \equiv l_3) \Rightarrow \\
&x' \not\equiv \bot \Rightarrow l' \not\equiv \bot \Rightarrow \\
&CONS\ x\ l \sqsubseteq CONS\ x'\ l' \Rightarrow \\
&\bot \equiv CONS\ x\ l \vee CONS\ x\ l \equiv CONS\ x'\ l'
\end{aligned}$$

Invertibility implies $x \sqsubseteq x'$ and $l \sqsubseteq l'$. Because α is flat and $x \sqsubseteq x'$, either $\bot \equiv x$ or $x \equiv x'$:

(1) If $\bot \equiv x$, then strictness implies $\bot \equiv CONS\ x\ l$.

(2) If $x \equiv x'$, then the induction hypothesis implies that $\bot \equiv l$ or $l \equiv l'$. Strictness solves the $\bot$ case. If $x \equiv x'$ and $l \equiv l'$, then $CONS\ x\ l \equiv CONS\ x'\ l'$. ■

5. Mutually Recursive Types

Several data types are *mutually recursive* if an element of any 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 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.

5.1. Expression trees

The example type for this section is 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 derivation for strict constructors is similar.

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 the type *exp* are

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

and for the type *elist*,

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

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*. It is easy to prove that *elist* and $(exp)llist$ are isomorphic. 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:

$$\forall el\,.\,ELIST(EXPLL\ el) \equiv el \ \wedge\ \forall ll\,.\,EXPLL(ELIST\ ll) \equiv ll$$

Use these isomorphisms to convert between $(exp)llist$ and *elist* as necessary.

5.2. An aside: local declarations

Here is a natural way to write the axioms of mutual recursion. 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 easy 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:

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

In words, "let $(x,y)=t$ in $P(x,y)$." Writing (x,y) on the left side of a declaration avoids writing *FST* and *SND* many times in the body of P, just as defining functions by cases avoids writing *NULL*, *HEAD*, and *TAIL*. Unfortunately, implication causes problems for fixed-point induction.

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 elists (expression lists), then the ***mutual induction*** rule is

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

Each mutually recursive type has a separate cases axiom:

$$\forall e{:}exp\,.\,e \equiv \bot \vee \exists v\,.\,e \equiv VAR\ v \vee \exists fn\ el\,.\,e \equiv APPL\ fn\ el$$

$$\forall el{:}elist\,.\,el \equiv \bot \vee el \equiv ENIL \vee \exists x\ el'.el \equiv ECONS\ e\ el' \qquad \text{(cases axiom)}$$

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 (copying functional axiom):

$$\begin{array}{l}(g,gl) \equiv EXP\_FUN(f,fl) \Longrightarrow \\ \quad g\ \bot \equiv \bot \wedge g(VAR\ v) \equiv VAR\ v \wedge g(APPL\ fn\ el) \equiv APPL\ fn(fl\ el) \wedge \\ \quad gl\ \bot \equiv \bot \wedge gl\ ENIL \equiv ENIL \wedge gl(ECONS\ e\ el) \equiv ECONS(f\ e)(fl\ el)\end{array}$$

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

$$(g,gl) \equiv FIX\ EXP\_FUN \Longrightarrow (g\ e \equiv e \wedge gl\ el \equiv el) \qquad \text{(reachability axiom)}$$

Theorem. *The cases, copying functional, and reachability axioms imply that the mutual induction rule for expressions/elists is sound.*

Proof. Assuming the premisses of the induction rule, it is enough to prove the conclusion, $\forall e.P(e) \wedge \forall el.PL(el)$. By the reachability axiom, it is enough to prove

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

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

$$\forall g\ gl\,.\,(g,gl) \equiv \bot \Longrightarrow (\forall e.P(g\ e) \wedge \forall el.PL(gl\ el)) \qquad (\bot \text{ goal})$$

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

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

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

Put aside the antecedents, and argue by cases on e and el:

$$\begin{array}{lll}P(g\,\bot) & P(g(VAR\ v)) & P(g(APPL\ fn\ el')) \\ PL(gl\,\bot) & PL(gl\ ENIL) & PL(gl(ECONS\ e'\ el'))\end{array}$$

Unfold the definition of EXP_FUN in these six goals:

$$\begin{array}{lll}P(\bot) & P(VAR\ v) & P(APPL\ fn(f\ el')) \\ PL(\bot) & PL(ENIL) & PL(ECONS(f\ e')(fl\ el'))\end{array}$$

Only the $APPL$ and $ECONS$ goals differ from the corresponding premisses of the mutual induction rule. The remainder of the proof resembles that for lazy lists. Instantiate the fixed-point induction hypothesis using e' and el'. Generalize the goals, putting e for $f\ e'$, and putting el for $fl\ el$. ■

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 [7,8,11]. The induction term $FIX\ EXP\_FUN$

appears inside an equivalence in a ***negative position***, the antecedent of an implication. We can salvage the derivation by extending the chain-completeness test, which 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 definition of *EXP_FUN* becomes unreadable, with clauses such as

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

Note: M.J.C. Gordon informs me that local declarations can be written without implication, since $P(t)$ is equivalent to $\exists x\,.\,x \equiv t \wedge P(x)$. But existential quantifiers also violate chain-completeness tests.

6. Types with Equational Constraints

The data types presented so far have all been word algebras [4] -- 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. I found equational types indispensable during the verification of the unification algorithm [13].

6.1. Finite sets

The example for this section is finite sets, which are related to finite lists. As the type $(\alpha)list$ has constructors *NIL* and *CONS*, the type $(\alpha)set$ has constructors

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

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:

$$\begin{aligned} INCLUDE\ x(INCLUDE\ x\ s) &\equiv INCLUDE\ x\ s \\ INCLUDE\ x(INCLUDE\ y\ s) &\equiv INCLUDE\ y(INCLUDE\ x\ s) \end{aligned}$$

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 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

$\perp \equiv CONS\ TT(CONS\ TT \perp) \equiv CONS\ TT\ NIL$

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:

$SET : (\alpha)list \to (\alpha)set \qquad LIST : (\alpha)set \to (\alpha)list$

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

$$\begin{array}{rll} & SET \perp & \equiv \perp \\ & SET\ NIL & \equiv EMPTY \\ \forall_D\, x\, l\, . & SET(CONS\ x\ l) & \equiv INCLUDE\ x(SET\ l) \end{array} \qquad \text{(quotient axiom)}$$

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

$$\forall s:(\alpha)set\, .\, SET(LIST\ s) \equiv s \qquad \text{(representative axiom)}$$

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:

$$INCLUDE \perp l \equiv \perp \wedge INCLUDE\ x \perp \equiv \perp \qquad \text{(strictness axiom)}$$

$$EMPTY \not\equiv \perp \wedge \forall_D\, x\, l\, .\, INCLUDE\ x\ l \not\equiv \perp \qquad \text{(definedness axiom)}$$

Clearly SET is a total function, by induction on lists. This is needed for the derivation of induction. Remember the advice from section 4 — you often must reason about totality when working with strict types. I can see no use for equational constraints on lazy types; the equivalence relation is likely to be undecidable.

Is it reasonable to postulate the function $LIST$, which can enumerate the elements of any set? Only if the elements have a simple type such as the integers; sets of integers may be implemented on a computer as sorted lists. My proof of unification [13] allows sets only if the element type is flat. Unfortunately, PPLAMBDA handles type conditions awkwardly.

6.3. Derivation of induction

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

$$\frac{\begin{array}{c} \textit{chain-complete } P \\ P(\bot) \quad P(EMPTY) \quad \forall_D\, x\ s\,.\,P(s) \Rightarrow P(INCLUDE\ x\ s) \end{array}}{\forall s. P(s)} \qquad \text{(set induction rule)}$$

Theorem. ***The strictness, definedness, quotient, and representative axioms, along with the induction rule for strict lists, imply that the set induction rule is sound.***

Proof. Assuming the premisses of the induction rule, it suffices to prove the conclusion, $P(s)$. By the representative axiom, it is enough to show $P(SET(LIST\ s))$; generalizing the goal gives $P(SET\ l)$. List induction produces three goals,

$$P(SET\ \bot) \quad P(SET\ NIL) \quad \forall_D\, x\ l\,.\,P(SET\,l) \Rightarrow P(SET(CONS\ x\ l)).$$

Unfolding the definition of SET gives

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

The $\bot$ and $EMPTY$ goals are now in proper form for the induction rule; the $INCLUDE$ goal needs massaging. The totality of SET gives

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

Putting s for the defined set $SET\ l$, it suffices to prove

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

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. 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:

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

A subtler example is the membership test. It requires a infix function $p\ OR\ q$ on truth values, strict in both p and q. To test whether z is a member of the set s, compare z with each element of s:

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

Viewed as a definition by primitive recursion (Section 8), it handles the *INCLUDE* case using $\lambda x\ r.(z=x) OR\ r$, where r represents the result of the recursive call. The definition is consistent because this lambda-abstraction satisfies the same equations as *INCLUDE*.

7. Models of Recursive Types

The preceeding sections have introduced numerous axioms, asserting cases, strictness, definedness, distinctness, and invertibility. These are theorems for types constructed from familiar primitives. Strict data types can be expressed as sums of products; lazy constructors require an additional type operator, for delaying the evaluation of arguments [5]. For types α and β, compound types for building models include

sum types

Every value of the sum $\alpha \oplus \beta$ is either $\perp$ or $INL\ x$ or $INR\ y$, for defined $x{:}\alpha$ and $y{:}\beta$. This is a *coalesced* sum — INL and INR are strict constructors.

product types

Every element of the strict product $\alpha \otimes \beta$ is either $\perp$ or has the form x/y, for defined $x{:}\alpha$ and $y{:}\beta$. This differs from the Cartesian product $\alpha \times \beta$; if x is defined, then so is $(x, \perp)$ but not $(x/\perp)$.

lifted types

Every element of the lifted type $(\alpha)u$ is either $\perp$, or $UP\ x$, for some $x{:}\alpha$.

the void type

The type *void* contains only the element $\perp$.

A recursive type is the solution of domain equations. The desired ***abstract*** type is defined to be isomorphic to a ***representing*** type involving sums, products, liftings, and *void*. For the strict list type $(\alpha)\mathit{list}$, the representing type is

$$(\alpha)rep \ = \ (void)u \ \oplus \ \alpha \otimes (\alpha)\mathit{list}.$$

By Scott's theory, we can set up isomorphisms between these types:

$$ABS\_LIST(REP\_LIST\ l) \equiv l{:}(\alpha)\mathit{list} \qquad REP\_LIST(ABS\_LIST\ r) \equiv r{:}(\alpha)rep$$

Since $(void)u$ represents NIL and $\alpha \otimes (\alpha)\mathit{list}$ represents $CONS$, the constructors are

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

Monotonicity implies that the isomorphisms are strict and 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

$$(void)u \ \oplus \ (\alpha)u \otimes ((\alpha)\mathit{llist})u,$$

and the lazy version of $CONS$ is

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

8. 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, and is also a concise notation for defining functions [2].

In ***constructive type theory*** [10], structural induction comes from 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 $h\ ll$.

Induction follows from 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 $LLIST\_REC$ combines the parameters $lnil$ and $lcons$, making a function on lazy lists:

$$h \equiv LLIST\_REC(lnil, lcons) \Longleftrightarrow$$
$$h\ \bot \equiv \bot \wedge h\ LNIL \equiv lnil \wedge \forall x\ ll.h(LCONS\ x\ ll) \equiv lcons\ x(h\ ll)$$

A 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

$$h\ \bot \equiv \bot \qquad h\ LNIL \equiv LNIL \qquad h(LCONS\ x\ ll) \equiv LCONS\ x(h\ ll)$$

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 $I\ x \equiv x$. The reachability axiom holds because these functions are equivalent: $FIX\ LLIST\_FUN \equiv I$.

Primitive recursion is similar for strict and mutually recursive types. For an equational type like set, primitive recursion gives a function for parameters ***empty*** and ***include***, provided these satisfy the same equations as the constructors $EMPTY$ and $INCLUDE$ [12].

Burstall and Goguen [4] describe an initial algebra as having two properties: ***no confusion*** (different terms get different values) and ***no junk*** (every element is the value of some term). These correspond to the ***existence*** and ***uniqueness*** of functions defined by primitive recursion. The unique homomorphism from an initial algebra corresponds to a primitive recursive function.

9. Experience in LCF Proofs

Edinburgh LCF users once defined each recursive type by constructing a model, a tedious job of asserting the axioms and deriving the induction rule. Milner [6] automated this for lazy types, using LCF's programming language, ML. Several projects [15] used Milner's program, and many other type definition programs descended from it.

In developing Cambridge LCF from Edinburgh LCF, I have added disjunction and existential quantifiers and used them in a type definition program. It allows strict and lazy constructors, but not mutual recursion. After constructing a strict type, it proves flatness. Cambridge LCF does not provide sums, strict products, or lifted types, since the program can define them. In the verification of the unification algorithm [13], the program developed strict data types for pairs, lists and expressions. I manually developed equational types for finite sets and substitutions, using the methods of section 6.

Acknowledgements: I would like to thank M.J.C. Gordon for daily discussions, R. Burstall for some stimulating conversations about algebras, and also R. Constable, 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] W.H. Burge, *Recursive Programming Techniques*, (Addison-Wesley, 1975).

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

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

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

[6] 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.

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

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

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

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

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

[12] L. Paulson, Structural induction in LCF, Report 44, Computer Laboratory, University of Cambridge (1984).

[13] L. Paulson, Verifying the unification algorithm in LCF, Report 50, Computer Laboratory, University of Cambridge (1984).

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

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

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