Functions + Logic in theory and practice

Richard B. Kieburtz
Oregon Graduate Center
19600 N.W. von Neumann Dr.
Beaverton, Oregon 97006 U.S.A.

Abstract

Although many models have been proposed for a union of functional and logic programming styles, none yet has fulfilled the goals of being easy to understand, easy to use and straightforward to implement efficiently. This paper presents a union of these styles in a language called F+L and discusses some aspects of the semantics and implementation of this language. Some interesting programming issues are revealed through examples.

F+L combines functional programming with Horn-clause logic and can be understood as a logical language with interpreted equality. Its semantics provide both a function space and relations over the non-functional semantic values. We outline a strategy for an efficient implementation by compiled graph reduction.

The research reported in this paper has been partially supported by the National Science Foundation under Grant No. DCR-8513572.

Functions + Logic in theory and practice

Richard B. Kieburtz
Oregon Graduate Center
19600 N.W. von Neumann Dr.
Beaverton, Oregon 97006 U.S.A.

Abstract

Although many models have been proposed for a union of functional and logic programming styles, none yet has fulfilled the goals of being easy to understand, easy to use and straightforward to implement efficiently. This paper presents a union of these styles in a language called F+L and discusses some aspects of the semantics and implementation of this language. Some interesting programming issues are revealed through examples.

F+L combines functional programming with Horn-clause logic and can be understood as a logical language with interpreted equality. Its semantics provide both a function space and relations over the non-functional semantic values. We outline a strategy for an efficient implementation by compiled graph reduction.

1. Introduction

There have been many proposals to merge functional and logic programming styles into the framework of a common language. In these proposals, two basic ideas come to light: (I) Add equations to a Horn-clause logic. This will impose a congruence upon the Herbrand universe of terms. (II) Add logical variables and unification to an equational logic. This can alter the interpretations of equations customarily given for a functional programming language. Most of the combined logics languages involve both of these ideas.

When a Horn clause programming logic is enriched with equations, interpretation of the combined logic no longer constructs a free Herbrand model. In order to obtain a complete resolution proof strategy, unification must also take account of the congruences of terms. A substitution S unifies a pair of terms t_1, t_2 relative to an equational theory $\mathbf{E}$ if $S(t_1) =_{\mathbf{E}} S(t_2)$. The ordinary unification algorithm is generalized to **E**-unification if it computes most general unifiers relative to an equational theory. Unfortunately, convergent algorithms for **E**-unification are known only for fairly simple theories. For a theory as rich as elementary arithmetic, equivalence of terms is undecidable and hence there can be no convergent algorithm for **E**-unification.

Consequently, a programming language based upon Horn clause logic enriched with equations may employ an incomplete resolution proof strategy[1]. Resolution offers

The research reported in this paper has been partially supported by the National Science Foundation under Grant No. DCR-8513572.

[1] Of course Prolog, even without equations, does not employ a complete proof strategy because it uses depth-first search in its attempt to find a proof quickly. This incomplete proof strategy is efficient, but is often troublesome to logic programmers.

no advantage over reduction to perform computations in a type that can readily be axiomatized by an equational theory. Such computations can be programmed functionally. Computation by resolution is useful to produce values in a non-algebraic type for which an equational axiomatization is not natural. (Of course a non-algebraic type may use algebraic values in its representation.)

When logical variables are allowed in the terms of an equation, we must be precise about the interpretations to be given to the equations. In a purely functional programming language the equations have no free variables. Every variable occurring in an equation is universally quantified (we also say lambda-bound). When logical variables are introduced, they may have free occurrences in an equation. Some designers of merged languages have chosen to interpret free variables as singly bound [Lin85, Red86] and some as multiply bound [GoM86, Smo86] or set-valued [DFP86].

When equations as well as Horn clauses are resolved by unification, rather subtle logical problems can arise. In equational logic, equality has the substitution property; a right-side expression can be substituted for a left-side expression (or vice-versa) without changing the meaning of a context. However, when the meaning of an expression is resolved by unification with the left side of an equation, a variable free with respect to the equation may become bound. Such a binding becomes a side effect of having used the equation. It is easy to demonstrate an incongruity by a foolish example:

```
def foo nil = 3;
let i = foo x in
5.x
```

The **let**-defined constant `i` is never referred to in the principal expression, yet if the **let** clause were evaluated anyway, the free variable `x` would be bound by unification. Does the expression have a value?

Another incongruity can occur if logical variables are interpreted as multiply bound and equations are resolved by unification. In each instantiation of an equation, a logical variable might take a different binding. Evaluating the same applicative expression in subsequent iterations of a segment of program can result in different values, because different bindings obtain for some variable that occurs free in an equation that is resolved. The meaning of an equation must then be explained as a relation entailing the values it equates and the values of its free variables [DFP86].

Although there are explanations for these phenomena, we have reservations about programming languages that manifest apparent incongruities of meaning. F+L is a rather conservatively designed language in which the meanings of programs will hopefully be obvious. Eqlog [GoM86] is an equational language designed to respect the integrity of the underlying logic. The designers of Eqlog have been somewhat more adventuresome with respect to implementation strategies, however. An F+L implementation can be nearly as efficient as that of a purely functional programming language.

2. Programming logics

A pure, functional programming language is an equational logic, usually subjected to syntactic restrictions that make it easy to give a weakly constructive interpretation[2] to the equations. The equations that define functions are oriented from left to right so

[2] We say weakly constructive because there is no guarantee that the evaluation of an expression will terminate.

that evaluation can proceed by reduction, or term-rewriting. Equations are commonly constrained to be left-linear, that is, no variable has a repeated occurrence in the left-hand side of any equation[3]. In order to make it easier to do efficient pattern-matching and to impose a notion of consistency on a system of equations, the set of constant symbols can be partitioned into defined functors and free constructors. Under the free constructor discipline, a left-hand side term may contain nested constructor applications but is allowed to contain at most one occurrence of a defined functor, which must be in the function position in the outermost application.

Equational languages of this kind have a well defined semantics, or model theory. The standard models are algebraic, weakly constructive, and the semantic domain contains its function space. Functions of any order can be defined in an equational language. Least fixpoints of recursive equations are computable whenever they exist. Completeness of an interpretation is assured by following a deterministic computation rule, the rule of normal order reduction.

A non-equational logic is usually given semantics by a Herbrand model. Without further restrictions, this tells very little about the properties of a model. For a programming logic, a weakly constructive interpretation is required. Accordingly, the syntax must be restricted so that quantifiers are not interleaved, conclusions are unequivocal, and a positive conclusion cannot be inferred from a negation. Horn clause logics meet these requirements, and we shall henceforth restrict our attention to these as our non-equational progrmming logics.

Even with the stringent restriction to Horn clauses, we are not guaranteed of algebraic properties of a model, nor given conditions for a deterministic evaluation strategy that will ensure completeness, short of basing evaluation on a complete proof theory[4]. Nevertheless, logic programming has some significant advantages. The semantics of predicate symbols are relations over the domain of ground values, and the relations need not be functional. Sets are expressed very naturally by characteristic predicates. Arbitrary projections of an undirected relation can be taken, whereas functional relations are directed; a function can only be applied. Logic programmers also advertise the advantages of demand-driven scheduling of evaluation, but this advantage is also inherent in functional programs with lazy evaluation [Joh86].

The advantages of a language that provides equational logic, Horn clause logic, free constructors and embeds semantic algebras for arithmetic and boolean domains are that:

a) Equational logic is an expressive notation in which to describe functions on an algebraic domain;

b) Horn clause logic is quite natural for expressing sets and non-functional relations. These are useful even for ~~algebraic~~ domains with algebraic constraints, but for computing over non-algebraic freely constructed domains they are essential data structures.

c) Data structures built with free constructors allow ~~both algebraic and non-algebraic~~ ~~freely constructed and~~ non-free values to be embedded; It is unreasonable to handicap programming with a language

[3] The language OBJ2 and its derivatives [OBJ85] do not require left linearity, but at the cost of less efficient pattern-matching.

[4] This is not generally considered feasible because it requires breadth-first search, which can be costly beyond any complexity bound. Some Prolog implementations have experimented with depth-bounded search, which simulates breadth-first search only if depth-first search to some previously determined depth bound fails.

that is natural for only one type of domain.

The challenge in combining these logics in a single programming language is how to give a coherent semantics to the combined logics, while respecting the integrity of the logic. Once the semantics is understood, implementations of the semantics will be relatively straightforward to design.

2.1. Equality interpretations

It is essential to understand the interpretations of equality that are possible in the combined logics. There is potential for confusion because the interpretations made of equality depend upon the syntactic context of an occurrence of the equality symbol. Equality is expressed as a relation on terms, but may be interpreted either on terms themselves (intensional equality) or on the meanings of terms (semantic, or extensional equality).

Boolean-valued equality. A Boolean-valued expression may select an arm of a conditional expression. This forces a very strong interpretation on the equality predicate. This equality interpretation is a (partial) decision procedure for the semantic equality of expressions. It requires construction of an explicit demonstration that the two arguments of an equality proposition have the same value or that their values differ. The arguments must generally evaluate to ground terms, for if they were to contain free variables, then the equality would have to be demonstrated for all possible valuations of the variables.

It is only possible to have a constructive interpretation for Boolean-valued equality if its domain is restricted to a type of finite elements. By this we mean a type all of whose values are finite elements of a semantic domain. Thus *int* and *list*(*int*) are admissible types for Boolean-valued equality, while *int*→*int* and *stream*(*int*) are not.

Definitional equality. The equality of definitions is given a more liberal interpretation. Definitional equality is extensional, but since no decision procedure is required, the type of this equality is unrestricted. Terms containing variables may be defined as equal, with the understanding that the equality holds for all possible valuations of the variables over their types. The property of referential transparency of expressions is a consequence of the fact that definitional equality is given this interpretation. It is not the same interpretation as is given to the assignment operator ':=' in imperative languages.

Predicated equality. A Horn clause logic can have an interpreted equality predicate. Logic terms in a Horn clause are tested for satisfiability; a query asks for the set of bindings for logic variables for which a clause is satisfiable. Satisfiability of equality is constructive if there is an effective decision procedure for the equality of non-ground terms and weakly constructive if there is a semi-decision procedure. In a free Herbrand model, unification gives an effective decision procedure for equality of first-order object terms.

In conjunction with an equational logic, a set of Horn clauses may admit non-free models. A partial semi-decision procedure for equality can be gotten by the two-step process of (1) reducing the arguments of a predicated equality to normal forms according to the equational theory, followed by (2) unification of the reduced terms. This is the process of narrowing [Fay79,Hul80,Red86]. Unification alone can only demonstrate the intensional equality of reduced terms that belong to types of finite elements. It can provide only a partial decision procedure for semantic equality, even if the reduction of terms were known to terminate.

It has been proposed to strengthen the decision procedure by using semantic or E-unification [GoM86, SuY86]. This would provide an extensional equality interpretation for predicated equality in the presence of an equational theory. Unfortunately, unification relative to a theory is not known to be complete except for rather simple theories. In particular, it is necessarily incomplete for the theory of elementary arithmetic, whose equality is undecidable.

3. F+L -- a language with functions plus logic

F+L is a programming language that integrates equational definitions, algebraic expressions, and positive definite Horn clause logic. It is evaluated by reducing applications of defined functors to canonical reduced terms and by narrowing to bind logic variables in order to satisfy constraining propositions.

F+L has a type system similar to the Milner type system for a purely functional ML [Mil78]. Not much will be said about typed terms in this paper. Types are assumed to be gotten by type inference rather than by explicit declaration. The reader should keep in mind that we are discussing a typed language.

An abbreviated F+L syntax is presented in Table 1 to assist in reading the examples. The alternative forms for each syntactic category are listed in order of precedence; the first listed form is more tightly bound that those that follow. *Patterns* (abbreviated as *pat* in Table 1) is a subclass of expressions with no operators other than free constructors. The identifier in the applicative position in an application pattern must be a free constructor declared by its occurrence in a type definition

A new type can be specified as a disjoint union of injections from constituent types. The type definition declares names, arities, and types for the constructors of a disjoint union in a pattern-structured syntax. This scheme of notation has been used in several functional programming languages.

Table 2: Syntax of type definitions

type_defn ::=	`type` *id* {(*type_var*[, *type_var*])} `is` *disjoint_union;*
disjoint_union ::=	*id* {(*type_expr* [, *type_expr*])} [+ *disjoint_union*]
type_expr ::=	*id* {(*type_expr* [, *type_expr*]}

The constructors declared in a type definition are distinguished from functor symbols declared in equations. Constructors are free, defined functors are not free. The literal constant symbols of an algebraic type, such as `true` and `false`, or numerals for the natural numbers or integers also have the status of free constructors.

We impose a requirement that constructors are free and functors are non-free through a convention called the *free constructor discipline.* If a system of equations **E** entails the congruence of expressions $C_1(x)$ and $C_2(y)$, where C_1 and C_2 are different constructors[5], then **E** is *inconsistent* with the free constructor discipline. Conversely, if there is a well-typed functor application $f\ e_1, \ldots, e_n$ whose **E**-congruence class contains no canonical constructor expression $C_i(x)$, then **E** is *incomplete* with the free constructor discipline. If an **E**-congruence class contains an irreducible functor application, but no canonical constructor expression, then the system **E** is *finitely incomplete.*

[5] It is understood that the constructor applications could also be nullary.

Table 1: Abbreviated syntax of F+L

pat ::=	*id*	variable, constant
	number	
	character	
	string	
	nil	empty list
	id (*pat* [, *pat*])	constructor application
	pat . *pat*	constructed list
	pat : *pat*	stream
	pat , *pat* [, *pat*]	tuple
	(*pat*)	
expr ::=	*pat*	
	expr expr	application
	'{' *pat* | *prop* [, *prop*] '}'	set expression
	expr operator expr	algebraic expression
	operator expr	algebraic expression
	expr . *expr*	list construction
	expr : *expr*	stream construction
	expr , *expr* [, *expr*]	tuple construction
	pat . *expr*	Abstraction
	if *expr* then *expr* else *expr*	conditional
	expr where var *id* [, *id*]	
	let *eq_defs* in *expr*	equational definition
	let *prop* [, *prop*] {*var_decl*} in *expr*	constrained expression
	def *logic_defs* in *expr*	predicate definition
	case *expr* in *case_inst* [| | *case_inst*] end	case expression
	(*expr*)	
prop ::=	*id* (*expr* [, *expr*])	
logic_defs ::=	*clauses* [and *logic_defs*]	
clauses ::=	*id* (*pat*) {:- prop [, prop]} [| | clauses]	
eq_defs ::=	*equations* [and *equations*]	
equations ::=	*id* {*pat* [*pat*]} = *expr* [| | *equations*]	
case_inst ::=	*pat* : *expr*	

Braces {,} are meta-symbols indicating zero or one occurrence of the enclosed text. Brackets [,] indicate zero or more occurrences.

In a semantically meaningful F+L program, an equational theory must neither be inconsistent nor finitely incomplete with respect to the free constructor discipline. Although these conditions are undecidable in general, it is well known that for equational theories alone (that is, without Horn clause logic) that if the left-hand sides of equations have non-overlapping patterns (no critical pairs) they are consistent. If the set of left side patterns covers all constructor patterns, the equations are not finitely incomplete. Neither of these restrictions is necessary, however, and they are not enforced for F+L programs.

The left side terms of both equational definitions and Horn clauses are syntactically restricted by the free constructor discipline, since a pattern cannot contain an occurrence of a defined functor.

Observation 1: A pattern more complex than a single variable cannot unify with the left side of a defining equation.

The free constructor discipline limits the expressiveness of the Horn clause logic. If an equational theory is consistent and finitely complete with respect to the free constructor discipline, then properties of equationally defined functions cannot be predicated by the restricted Horn clause logic. Although equality can be predicated in the logic, finite completeness assures that no irreducible functor terms will be equated by unification in resolving a predicated equality, since reduction of terms precedes unification. Conversely, resolution of a predicated equality fails on an attempt to unify dissimilar constructor terms.

Observation 2: If an equational theory is consistent and finitely complete, it remains so with the addition of Horn clauses respecting the (syntactic) free constructor discipline.

Under the free constructor discipline, **E**-unification is also simplified with respect to the general case. If a well-typed propositional expression p contains as an argument a term $t = f\ e_1, \ldots, e_n$ but t is not an instance of one of the left sides of the equations defining f, then t must be unifiable with the left sides of one or more of these equations. After choosing an equation and unifying, the substituted term t' reduces. Thus from p is obtained a finite set of narrowed propositions to be resolved. The narrowing steps may be recursively repeated until the set of propositional terms contains no occurrence of an unreduced application of f.

Observation 3: Under the free constructor discipline, and for an equational theory **E** that is consistent and finitely complete, narrowing is relatively complete as **E**-resolution for Horn clause logic. If the equational theory is also noetherian, then narrowing relative to **E** gives a complete proof theory for positive definite Horn clause logic.

The second assertion of Observation 3 can be proved by a recursive-path ordering.

The free constructor discipline ensures a comfortable marriage of equations and Horn clause logic.

3.1. Logical variables and constrained expressions

The syntax and semantics of equational definitions in F+L are largely inherited from LML [Joh83]. In F+L, however, logic variables [Lin85] may occur in expressions along with universally quantified variables. The fact that reductions produce canonical reduced terms, not necessarily values, is a consequence of the introduction of logical variables. In another departure from its parent language, applications of functions and constructors (other than the stream constructor, ':') are strict. Streams are introduced as distinct types, not to be confused with list types.

A canonical reduced term (henceforth called a *cart*) is either a basic value (i.e. an integer, character or boolean value), an unbound logical variable, a canonical construction of carts or the abstraction of a cart. Thus, the terms we are accustomed to regarding as canonical value terms of a functional programming language are carts, but a cart may also contain occurrences of unbound logical variables. A non-canonical expression, such as 3+u, where u is a logical variable, is not a cart. In fact, an attempt to evaluate a non-ground, non-canonical expression such as `3+u` will be detected as an error, since the algebraic operator '+' expects its arguments to be values. The notion of cart is derived from [Lin86] although the present author is to blame for introducing the name.

A **def** clause consists of a sequence of Horn clauses defining a set of predicate symbols. Every variable occurring in a Horn clause is considered to be bound and cannot be confused with a free variable. A Horn clause

$$P(x_1,...,x_n) :- Q_1(...), \ . \ . \ . \ , Q_m(...)$$

is implicitly quantified as

$$\forall \ x_1,...,x_n . P(x_1,...,x_n) :- \exists \ y_1,...,y_k . Q_1(...), \ . \ . \ . \ , Q_m(...)$$

where the $y_1,...,y_k$ includes all variables different from the $x_1,...,x_n$ that occur in the terms on the right side of the clause.

A logic variable is introduced in a **let** clause[6], for instance

```
let var x in expr
```

An expression that contains occurrences of a logic variable has the same operational semantics as any other expression in which free variables occur; a cart is computed by reduction in an environment, ρ, that maps variables to carts. An environment defining carts for the lambda-bound, or universally quantified variables of a function definition is gotten at each application of the function by matching arguments to variables by position. An environment that defines carts for logic variables is computed by resolving the definitions of predicates, in order to satisfy a propositional constraint.

A logic variable may be bound by narrowing. This occurs as a consequence of satisfying a constraint. A constrained expression has a **let** clause that consists of a constraining proposition, rather than an equational definition. For instance, the constrained expression

```
let P(x) var x in expr
```

has a cart relative to an environment ρ such that `P(`ρ`(x))` is satisfied, otherwise it is undefined. The proposition `P(x)` constrains the interpretations allowed for `expr`.

A constraining propositional term need not contain an explicit occurrence of a logic variable in order to narrow its interpretation. Since a logic variable can occur as an argument in an applicative expression, it may be constrained by the definition of the function that is applied.

3.2. An example: binding definitions

The following example is an adaptation of one that has appeared in the logic programming literature [AdF86,Red86]. (Unlike Prolog, there is no syntactic significance to the use of capitalized identifiers in F+L. For readability, the syntax allows the abbreviation `[a;b;c]` for an explicit list construction `a.b.c.nil` in patterned expressions.)

[6] A **where** clause would do as well. The reason that logic variables are introduced explicitly rather than implicitly as in [Lin85] is to avoid the accident of having a misspelled identifier introduce a new logic variable.

Example 1: binding definitions

```
type Term is DEF(char) + USE(char);
def Member (a,(a.S))
||  Member (a,(b.S)) :- Member (a,S)
in
letrec
    defseq (USE(x).seq) n T = let Member((x,i),T) var i in
                              i:defseq seq n T
||  defseq (DEF(x).seq) n T = let Member((x,n),T) in
                              defseq seq (n+1) T
||  defseq nil n T = 0:s where var s in
    -- 0 is a sentinel value marking the end of a stream of output
let input =
    [USE('B');DEF('B');USE('B');USE('A');USE('C');DEF('C');DEF('A')] in
defseq input 1 Table  where var Table
```

In this example, `defseq` is a function that maps its first argument, a list of type `Term`, to a list of natural numbers. The output stream[7] is the image of the sequence of `USE` terms in the input list. The value associated with each `USE(id)` is the index of a corresponding `DEF(id)` in the sequence of `DEF` terms in the input list. The second argument to `defseq` is the index to be given to the next `DEF` term encountered. The third argument represents a table which associates identifier characters with the index of a `DEF` term that contained the identifier.

The predicate `Member` represents the relation that an identifier-index pair exists in the table. The result of evaluating this program is the sequence `[1;1;3;2]`.

In the first of the defining equations for `defseq`, the expression on the right side is constrained to have a cart in an environment in which the proposition `Member((x,i),T)` is satisfied. This constraint may narrow either the interpretation of the identifier `i` or that of `T` depending on whether or not the table contains an association pair for `x` when the constraint is enforced. Note that `Table` is introduced as an unbound logical variable in the applicative expression in the last line of the program.

The second defining equation for `defseq` is constrained by the proposition `Member((x,n),T)`, which may narrow the interpretation for `T`, and may also possibly narrow the interpretation of instances of the variable `i` that were introduced by reductions of `defseq` applications that matched the first equation.

It is inconsequential to this example whether or not the language definition specifies lazy or strict evaluation rules for function application or list construction. Logic variables are inherently lazy.

This example could also be programmed entirely in Horn clause logic. To do so, the functor `defseq` of the F+L program is replaced by a four-place relation symbol,

[7] Output is a stream in order to obtain incremental evaluation, since the list constructor is strict in F+L. The program would still work if the output were presented in a list, but evaluation would then not be incremental.

```
    r_defseq((USE(x).seq),n,T,i.outseq) :-
                    Member((x,i),T), r_defseq(seq,n,T,outseq)
 || r_defseq((DEF(x).seq),n,T,outseq) :-
                    Member((x,n),T), r_defseq(seq, (n+1),T,outseq)
 || r_defseq(nil,n,T,outseq)
```

and the expression applying the functor to an input list becomes a propositional query:

```
?outseq in r_defseq(input,1,T,outseq)
```

As a program, this version is harder to read because the essentially functional nature of the relation expressed by defseq is not manifest. Mode declarations would help, but the program is certainly less clear without the use of equational logic to express functional dependency.

The same example can also be programmed in a functional language with lazy evaluation rules. Here is a program in LML:

```
import type Term = DEF(char) + USE(char);
letrec
   lookup x ((y,n).s) = if x=y then n
                        else lookup x s
and
   enter a nil = [a]
|| enter a (b.s) = b. enter a.s
and
   defseq (USE(x).seq) n T =
             let (outseq,T) = defseq seq n T in
             lookup x T. outseq, T
|| defseq (DEF(x).seq) n T =
             defseq seq (n+1) (enter (x,n) T)
|| defseq nil n T = nil, T
in
fst (defseq input 1 nil)
```

In this version of the program, we need to express the relation of the association list to the input sequence functionally. This has two obvious consequences. It is necessary to replace the predicate Member with the definitions of two functions, lookup and enter. It is further necessary that the function defseq must return a pair of values, the desired output list and an auxiliary list representing the association list. The need to separately define functions for table entry and lookup is not a serious flaw because it does not make the program harder to understand, but introducing a spurious value as a component of the result of a defseq application is distracting.

There is a more disturbing consequence which is not so obvious. Even though lazy evaluation is the computation rule (evaluation fails under a call-by-value rule), the output sequence ~~cannot be~~ is not produced incrementally! When the program is executed, the first element of the output sequence ~~cannot be~~ is not printed until the entire input sequence has been read, even though the definition of the first USE occurrence has been read earlier. This circumstance arises because the association list cannot be constructed until the `nil` ~~list~~ value marking the end of the input has been seen. Even though entries are inserted only at the tail end of the association list, there is no way to express in the syntax of the functional language that an initial sequence of the association list is never changed by inserting a new value.

The two versions of the program that make use of Horn clause logic express the construction of the association list with the use of logical variables as placeholders, rather than by suspending construction of the list itself. Logical variables provide a finer-grained unit of delayed evaluation.

It is important to allow incremental evaluation when the programmer intends it. A problem that functional programming languages have not solved is to produce multiple output lists without synchronous coupling. If the output lists are produced totally independently, then expression as a pair of lists is satisfactory, but if there is a dependence between them, it is difficult to avoid the problem illustrated by the `defseq` example. The introduction of logical variables appears to provide a way to accomplish this. A further consideration, not elaborated on here, is the fact that synchronously produced result sequences may be asynchronously consumed. This can lead to the storage management problem known as "space leaks" [Wad86].

3.3. Referential transparency

As mentioned before, the property of referential transparency is a consequence of equational logic. It cannot be compromised by introducing logical variables if we are to respect the integrity of the logic.

equation on one line, reversed.

Observation 4: Let ρ be an environment such that *expr* has a cart relative to ρ. If e has an occurrence at i in *expr*, and if $Reduce_{\mathbf{E}} \| e \| \ \rho = v$, then $Reduce_{\mathbf{E}} \| expr[i \leftarrow v] \| \ \rho = Reduce_{\mathbf{E}} \| expr \| \ \rho$.

Here, $Reduce_{\mathbf{E}}$ is the operation of reduction modulo a term rewriting system **E**.

The computation of an environment in order to satisfy a constraining proposition is not a side effect, since the proposition must be satisfied in order that the cart of a constrained expression is defined. Furthermore, a subsequent evaluation of the expression in the same environment cannot produce altered bindings because the constraining proposition will already have been satisfied. This principle answers the question as to whether logical variables can have multiple bindings in a language defined as is F+L.

3.4. Set-typed expressions

Logic clauses can be used to specify a set in terms of a characteristic predicate. Generalizing this mode of specification slightly, a set is specified by an expression for its members, constrained by a conjunction of propositional terms. Variables that occur in the expression for members are quantified over the set expression. For instance, the set expression

```
{e(x,y) | P(x,3), Q(x,y)}
```

denotes a set of carts obtained by substituting in `e` the ordered pairs of carts that simultaneously satisfy the constraining propositions `P(x,3)` and `Q(x,y)`, and reducing the resulting expression instances. The variables are, of course, implicitly universally quantified in a set expression. of infite extent.

The representation of a set is elaborated as a stream (a lazy list), with possible repetitions of elements. Stream access operations are defined on set-typed expressions. To compute an element of the stream representing a set, first compute an environment in which the constraining proposition is satisfied. The bindings given by this environment are then used in reduction of the members expression[8] to a cart. The tail of the stream contains the carts that result from additional bindings for the variables.

There ~~are two~~ is a special case~~s~~ of set expressions that ~~are~~ is worth mentioning. ~~When the members expression is an individual variable or a tuple of variables, the set value is a projection of a relation defined by the predicate expression.~~ When the predicate expression is empty, then the set value consists of the values of the members expression under all instantiations of the logical variables over their respective types.

When the evaluation of the members expression produces a cart that is not a ground value, subsequent evaluation of the program may narrow its interpretation to a single value. A non-ground cart cannot be presumed to represent more than a single value. In order that the stream elaboration of a set expression can be complete, it must produce repetitions of carts in which the sets of occurring logical variables are disjoint. (A) → [Upon repeated selection of its head, the head of its tail, etc., the set expression evaluates to a stream of fresh instances of non-ground carts.] This mechanism is used in Example 2 to generate independent axioms from a list of axiom schemes.

This example is a resolution theorem prover for minimal combinatory logic. It was originally programmed in Standard ML as an example in lecture notes by G. Huet [Hue86]. The first two axiom schemes are formulae for the types of **K** and **S**; the third is the rule for application of an arrow type. A proof of a formula is derived by a series of arrow applications of the axioms for **K** and **S**, resulting in the formula to be proved. Derivations are constructed in reverse order by this program because it proceeds backward from the goal. The function `all_proofs` produces a stream of traces of the derivations found in constructing proofs of a list of desired goal formulae.

In the ML version, a unification algorithm had to be programmed explicitly. A term constructor to represent variables was necessary, and new instances of axioms had to be constructed explicitly by generating new variable terms. In the version shown above, the F+L implementation resolves logic clauses, and new instances of carts defined in the stream representation of a set are created each time the stream is separated into head and tail. The function `new` performs this separation over a list of set typed elements. The programmer must appreciate the distinction between an axiom and an axiom scheme, ~~which~~ the axiom scheme is represented as a set expression~~,~~ in order to provide for independent instances of the axioms.

4. An evaluation strategy

Lazy evaluation of functional language programs requires the evaluator to distinguish the representation of a suspended application from a canonical value. With lazy evaluation, it is no longer possible for a compiler to embed in the compiled code an accurate schedule for evaluating applicative expressions, as it can for a call-by-value rule. The abstract architecture of a lazy evaluator depends on having tagged data objects.

[8] We use the term "members expression" to denote the part of a set expression preceding the stroke.

Example 2: A prover for minimal combinatory logic

```
type term is CONST(int) + ARROW(term,term);
let KS = ({(ARROW(x,ARROW(y,x)), nil) | var x,y}).
         ({(ARROW(ARROW(x,ARROW(y,z)),ARROW(ARROW(x,y),ARROW(x,z))), nil)
                   | var x,y,z}).
         ({(y, [ARROW(x,y); x]) | var x,y}).nil;
    -- Axiom schemes for miminal combinatory calculus.
letrec Resolve (goal.subgoals, (goal,hypoths).R, hypoths, R, 1, true)
    || Resolve (goals, a.R, h, S, i+1, success) :-
                Resolve (goals, R, h, S, i, success)
    -- i counts the number of steps to a successful resolution
and new nil = nil,nil
 || new (h.t) = let (nt,tt) = new t in
                (shd h).nt, (stl h).tt
    -- forms a list of new instances from a list of axiom schemes.
    -- shd and stl are the head and tail operators on streams.
in
letrec
   all_proofs nil $ = nil -- goals exhausted
|| all_proofs (($,$,$,nil).Rest) Th = all_proofs Rest Th -- a proof path fails
|| all_proofs ((trace, goal.subgs, j, Axs).Rest) Th =
      let Resolve ((goal.subgs), Axs, hyps, residue, i, success)
          var hyps, residue, i, success)
      and (insts,Th) = new Th in
      if success then
         case subgs @ hyps in
            nil : (i+j.trace):
                  all_proofs (Rest @
                              [trace, goal.subgs, i+j, residue]) Th
         || goals : all_proofs (Rest @ [i+j.trace, goals, 0, insts] @
                                [trace, goal.subgs, i+j, residue]) Th
         end
      else
         all_proofs Rest Th
and (KS_axioms,KS_Theory) = new KS
in all_proofs [nil, [ARROW(CONST(1),CONST(1))], 0, KS_axioms] KS_theory
```

Unbound logic variables must also have a tagged representation in the abstract architecture of an F+L evaluator. A compiler can schedule the bindings of functional variables. Binding occurs when a function is applied or a data structure is built to represent a suspended application. Bindings of logical variables cannot be statically scheduled, thus the occurrences of variables must be representable in a similar manner as are values.

This similarity suggests that a tagged-memory architecture for evaluating purely functional programs by graph reduction might be equally as suitable for evaluating

F+L programs. Gary Lindstrom has observed that a reduction mechanism can be used to evaluate logic programs [Lin86]. The capability for lazy evaluation of applicative expressions is required because F+L has stream-typed objects, thus the abstract machine's data types are:

(CT)	canonical term
(App)	suspended application
(Ubv)	unbound variable

The action that can be taken on a data object depends upon the demand of the operator that is applied. Algebraic operators, such as '+', '&', '<' demand evaluated arguments. The EVAL combinator that is compiled when lazy evaluation is the rule demands only a suspension. A free constructor, viewed as an operator, demands only a cart. The following table gives the operator application rules for the machine data types.

Demand	**Tag**	**Action**
value	CT	apply the operator directly
	App	fail with an error[9]
	Ubv	error notification
suspension	CT	apply the operator directly
	App	evaluate first, then apply operator
	Ubv	error notification
cart	CT	apply the operator directly
	App	apply the operator directly
	Ubv	apply the operator directly

[9] If a compiler emits correct code, this case should never occur.

4.1. Evaluating propositions

Although syntactic resolution provides a straightforward means for evaluating propositions relative to a set of Horn clauses, it is rather expensive computationally. Modern Prolog compilers eschew the unification of terms whenever possible, to secure better performance. In our evaluation strategy, full unification is used only to evaluate predicated equality. Resolution of every other propositional form is accomplished with a unification compiled specifically for the term that occurs in the head of a particular clause. Thus specialized, term unification is little more than pattern matching. In particular, the occurs check is unnecessary except when testing predicated equality.

At the outset, Horn clauses are rewritten to make predicated equality explicit. A clause whose head contains repeated occurrences of a variable is rewritten, introducing new variables to make all occurrences distinct. For each new variable introduced, the proposition of its equality to the old variable is explicitly added to the conjunct of the right side. For instance,

$$\texttt{P(x,x.S) :- Q(x,S)} \Longrightarrow P(x,y.S) \text{ :- } y{=}x,\ Q(x,S)$$

This produces a logically equivalent clause, but operationally it allows the unification of repeated occurrences to be deferred until after a goal term is successfully unified with the head of a clause.

A proposition expresses a constraint on the values that might be assumed by unbound logical variables. Evaluating a propositional expression can produce bindings for previously unbound variables. We can explain in a semi-formal way exactly how this occurs. This amounts to an operational semantics for propositions involving predicate symbols defined by Horn clause logic[9].

We shall make use of the following types:

Semantic Types		
Env	$Var \rightarrow Cart$	Environments
Trans	$Env \rightarrow Env$	Environment transformations
Cont	$Trans \rightarrow Trans$	Continuations

The "value" of a proposition in an environment is a new environment. Thus the meaning of a proposition is an environment transformation. This is computed dynamically by attempting to satisfy a set of Horn clauses. In case of failure, alternative clauses must be tried. This backtracking can usefully be represented as a continuation.

A predicate symbol P is defined by a set of Horn clauses having P-propositions at their heads,

$$\begin{array}{l} P_1 :- Qs_1 \\ P_2 :- Qs_2 \\ \quad\vdots \\ P_n :- Qs_n \end{array}$$

For each predicate symbol P, defined by a set of Horn clauses, there is a function $\text{Resolve}_P : Term \rightarrow Cont \rightarrow Cont$ that expresses the resolution of a proposition G against the P-clauses:

```
Resolve_P $G\ \omega\ \kappa\ \rho =$
    case unify $G$ with
        $P_1 \Rightarrow \rho_1$ : let $\omega' = \text{Resolve}_{P/P_1}\ G\ \omega$ in
                    Satisfy $Qs_1\ \omega'\ \kappa\ (\rho_1 \cup \rho)$
    ‖   $P_2 \Rightarrow \rho_2$ : let $\omega' = \text{Resolve}_{P/P_1,P_2}\ G\ \omega$ in
                    Satisfy $Qs_2\ \omega'\ \kappa\ (\rho_2 \cup \rho)$
    ‖   ...
    ‖   $P_n \Rightarrow \rho_n$ : Satisfy $Qs_n\ \omega\ \kappa\ (\rho_n \cup \rho)$
    ‖   $ : $\omega\ \kappa\ \rho$
    end
```

The symbol '$' is a universally matching pattern that designates a default case instance. The notations $\text{Resolve}_{P/P-sequence}$ indicate residual resolutions with the P-clauses less the clauses headed by members of $P{-}sequence$.

In this notation (which is not F+L), the i^{th} case instance is selected by a successful unification with the head of the i^{th} clause, producing a unifying substitution (an environment) ρ_i. In that case, a continuation expressing the resolution of the term with all the remaining P-clauses is passed as an argument to a function that expresses the result of satisfying the conjunction of propositions on the right side of the i^{th} clause. The continuation argument is invoked only in case of failure. When a continuation

[9] This operational semantics uses a leftmost, depth-first search strategy on a list of goal propositions and hence, does not constitute a logically complete inference system for the logic. This can be remedied by adopting a modified search strategy, such as incremented, depth-bounded searches.

argument is invoked, all the bindings made on the way to a failure are ignored.

A function that expresses the computation of an environment satisfying a conjunction of propositions is

$$\begin{aligned} & \text{Satisfy } \textit{nil}\ \omega\ \kappa\ \rho = \kappa\ \rho \\ \| \ & \text{Satisfy } (Q.Qs)\ \omega\ \kappa\ \rho = \\ & \quad \textbf{let } P(\mathrm{A}) = Q \textbf{ in} \\ & \quad \text{Resolve}_P\ (\textit{Reduce}\ \ Q\ \ \rho)\ \omega\,(\text{Satisfy } Qs\ \ \omega\ \kappa)\ \rho \end{aligned}$$

Of course the selection of a particular Resolve function is made by case selection on the predicate symbol appearing in the goal term.

The meaning of a constrained expression,

```
let P(A) in expr
```

relative to an environment ρ, is

$$\textit{Reduce}_{\mathrm{E}}\ \ \texttt{expr}\ \ (\text{Satisfy } \texttt{[P(A)]}\ \ \textit{Fail}\ \ \text{Id}_{\textit{Trans}}\ \rho)$$

where *Fail* is a system-defined failure continuation.

In denotational semantics, continuations were introduced to explain discontinuous control transfers, or go-to's. In the compiled implementation of an F+L program, the invocation of a continuation argument, which is seen to be a tail-call in the description of Resolve, is made by a direct control transfer, or go-to.

4.2. Compiled narrowing

In explaining the semantics, the resolution of a goal term $P(\mathrm{A})$ with the heads of P-clauses was described as a unification. A compiled implementation need not make use of a general algorithm for every one of these unifications, however. Since pattern terms are the heads of Horn clauses that occur in the program text, a compiler can analyze the set of P-clauses to produce a unification algorithm custom-tailored for each predicate symbol P. Recall that resolution is restricted by (1) having required the free constructor discipline in the heads of clauses, and (2) having transformed each clause into a left-linear form with predicated equality in its hypotheses. Resolution thus restricted can be compiled into efficient code by adaptating an algorithm that compiles matching of case-instance patterns in a functional programming language [Aug85]. We shall illustrate with an example.

Suppose a predicate P is defined by a pair of Horn clauses,

```
   P(C1(C0))  :- H1
|| P(C1(x))  :- H2
```

where `H1` and `H2` are lists of propositions, possibly empty. Then the Resolve function produced by a compiler for these P-clauses would correspond to:

```
Resolve$_P$ $G\ \omega\ \kappa\ \rho =$
    let $P$(A) = G in
    def var $x$ in
    case A in
        Ubv($v_0$) : let $\omega' = \lambda\kappa.\lambda\rho.$Satisfy H2 $\omega\ \kappa$ (($v_0$,C1($x$)).$\rho$) in
                    Satisfy H1 $\omega'\ \kappa$ (($v_0$,C1(C0)).$\rho$)
    ||  CT(C1(C0)) : let $\omega'$ = Satisfy H2 $\omega$ in
                    Satisfy H1 $\omega'\ \kappa$ (($x$,C0).$\rho$)
    ||  CT(C1(y)) : Satisfy H2 $\omega\ \kappa$ (($x$,y).$\rho$)
    ||  $ : $\omega\ \kappa\ \rho$
    end
```

Here, we have represented the bindings of logical variables as identifier-cart pairs, and an environment as a list of such pairs.

In a compiled implementation, a logical variable is represented by a node in an expression graph marked with a Ubv tag. There is no explicit representation for an environment; it is captured in the state of memory representing the current expression graph. The next version of the implementation is described as an imperative procedure, much closer in structure to the code that would be produced by a compiler.

```
declare resolve$_P$ procedure (G, $\omega$);
    allocate Ubv($x$);
    let $P$(A) = G in
    case A in
        Ubv($v_0$) : label $\omega'$: unbind [$v_0$];
                                bind ($v_0$,C1($x$));
                                satisfy (H2, $\omega$);
                                end in
                    bind ($v_0$,C1(C0));
                    satisfy (H1, $\omega'$);
    ||  CT(C1(C0)) : label $\omega'$: unbind [$x$];
                                satisfy (H2, $\omega$);
                                end in
                    bind ($x$,C0);
                    satisfy (H1, $\omega'$);
    ||  CT(C1(y)) : bind ($x$,y);
                    satisfy (H2, $\omega$);
    ||  $ : goto $\omega$;
    end;
```

Since an environment is represented by the state of graph memory, a failure continuation in the procedural implementation must "unbind" the logical variables bound en route to a failed unification.

The procedure that does the work of Satisfy is:

```
declare satisfy procedure (Gs, $\omega$);
    case Gs in
        nil : return;
    ||  (Q.Qs) : let $P$(A) = Q in
                    resolve$_P$ (A, $\omega$);
                    satisfy (Qs, $\omega$);
end;
```

5. Abstract interpretations

Compilation of F+L can be assisted by several abstract interpretations of a program.

Type inference is an abstract intepretation computing the (polymorphic) type signature of each expression. It is straightforward to extend the interpretation to predicate symbols, computing the type signature of the relation defined by each predicate. A predicate P is badly typed if the most general types inferred from the clauses defining P fail to have a least upper bound in the type system.

Mode analysis tries to determine for each occurrence of a logic variable in the literal terms of a Horn clause whether the variable will definitely be bound to a canonical value when the literal is to be satisfied (input mode), will definitely not be bound but may acquire a value if the literal is satisfied (output mode), or might fall under either of these cases (indefinite mode). Mode analysis has been proposed as an aid in compiling control for logic programs [Nai85].

When applied to the implementation scheme we have outlined, knowledge that the i^{th} argument of the head of a clause has a definite input mode would mean that in the case analysis of this term as a pattern, the test for a Ubv in the i^{th} position of the goal proposition could be omitted. Conversely, if the i^{th} argument has a definite output mode, then a matching proposition must not have a canonical term in the corresponding position and the test for a CT tag can be omitted. When the mode of an argument is indefinite, no such optimization is applicable.

V-strictness is the property that the argument of a function must be a canonical term value, not an unbound logical variable. The property is defined by analysis of a set of defining equations.

Definition 1: We say of an equation

$$E\colon\ f\ p_1\ \cdots\ p_n = e$$

that E is V-strict at the i^{th} place, $1 \le i \le n$, if either p_i is not a simple variable, or $p_i = x$ and x is strict in e.

Definition 2: A functor f defined by a set of equations $\mathrm{E} = \{E_1, \ldots, E_m\}$ is V-strict in the i^{th} position if $\exists\ j$, $1 \le j \le m$, such that E_j is V-strict at the i^{th} place.

Definition 3: We say that a variable x is V-strict in an expression according to the syntactic cases given in Table 3: Notice in particular that an occurrence of a variable x in an arbitrary context is not a V-strict occurrence. In an arbitrary context, one not requiring that x be evaluated, a logical variable will do as well.

When a variable occurs as an argument of an algebraic operator or as the discrimination expression of a case or a conditional expression, its value is required to reduce the expression. It is an error if an unbound logical variable is encountered in such a

Table 3: V-strictness in an expression

x is V-strict in:	if x is V-strict in
x+e e+x (also for -,*,/, &, |, =,~=,<,<=,>,>=)	
-x ~x	
`if` x `then` e_1 `else` e_2	
`if` e `then` e_1 `else` e_2	e or e_1 or e_2
`case` x `in` $\cdots$	

`case` e `in`
$p_1 : e_1$
$\vdots$
$p_n : e_n$
`end`
	e or any of e_i, $1 \leq i \leq n$

`C(`e`)`	e, where `C` is a free constructor
e_1,e_2	e_1 or e_2
e_1.e_2	e_1 or e_2
e_1:e_2	e_1 or e_2
$\lambda p.e \quad p \neq x$	e
$e_1\ e_2$	e_1 or e_2
x is V-strict in:	if:
$f\ e_1 \ldots e_{i-1}\ x\ e_{i+1} \ldots e_m$	f is V-strict in the ith position
`let` $P(e_1,\ldots,e_m)$ `in` e	x is V-strict in e and it is not the case that: $\exists\, i$, $1 \leq i \leq m$ and $e_i = x$ and the ith argument of P has definite output mode, or x is V-strict in any of the e_i, or $\exists\, i$, $1 \leq i \leq m$ and $e_i = x$ and the ith argument of P has definite input mode

position at the time the expression is to be evaluated. Accordingly, a compiler must either verify that all such occurrences of variables have previously been bound at the time that an expression is evaluated, or must insert a runtime check of the tag on the data representation, to confirm that it is not Ubv. The use made of V-strictness is to help a compiler to avoid inserting these runtime tests whenever possible.

Accordingly, if an equation $f\ p_1 \ldots x_i \ldots p_n = e$ is V-strict in the ith place because the variable x_i is V-strict in e, we should like the compiled code to check the tag of x_i at most once in reducing an application of f. Using the information that f is V-strict in the ith position, the compiler can insert the code to check the UBV tag of the ith argument expression of an application $f\ e_1 \cdots e_n$ only if that argument cannot be confirmed to have a value at the point of the function call. This is directly analogous to the use made of strictness analysis to permit call-by-value argument passing in a lazy functional language.

Acknowledgements

Conversations over a period of time with Boris Agapiev, Gary Lindstrom, Jose Meseguer, Harry Porter, Uday Reddy, Bart Schaefer and Cliff Walinsky have helped the author to crystallize the ideas presented in this paper.

References

[AdF86] Adams, D. D. and Faget, J., "Logic programming viewed functionally," *Proceedings of the 1986 Graph Reduction Workshop*, Santa Fe, New Mexico, 1986.

[Aug85] Augustsson, L., "Compiling pattern matching," in *Functional Programming Languages and Computer Architecture*, vol. 201, J. Jouannaud (ed.), Springer-Verlag, 1985, pp. 368-381.

[DFP86] Darlington, J., Field, A. J. and Pull, H., "The unification of functional and logic languages," in *Logic Programming: Functions, Relations and Equations*, D. DeGroot and G. Lindstrom (ed.), Prentice-Hall, 1986, pp. 37-70.

[Fay79] Fay, M., *First-order unification in an equational theory,* Proc. 4th Workshop on Automated Deduction, 1979. pp. 161-167

[OBJ85] Futatsugi, K., Goguen, J., Jouannaud, J. and Meseguer, J., "Principles of OBJ2," *Proc. 12th ACM Symp. on Principles of Programming Languages*, New Orleans, LA., Jan. 1985, pp. 52-66.

[GoM86] Goguen, J. A. and Meseguer, J., "EQLOG: equality, types, and generic modules for logic programming," in *Logic Programming: Functions, Relations and Equations*, D. DeGroot and G. Lindstrom (ed.), Prentice-Hall, 1986, pp. 295-363.

[Hue86] Huet, G., *Formal Structures for Computation and Deduction*, INRIA, Rocquencourt, 1986.

[Hul80] Hullot, J., "Canonical forms and unification," in *Proc. 5th Conf. on Automated Deduction*, vol. LNCS 87, W. Bibel and R. Kowalski (ed.), Springer-Verlag, 1980, pp. 318-334.

[Joh83] Johnsson, T., *The G-machine -- an abstract architecture for graph-reduction*, Dept. of Computer Sciences, Chalmers Univ. of Technology, Gothenburg, 1983.

[Joh86] Johnsson, T., *Attribute grammars and functional programming*, Dept. of Computer Sciences, Chalmers Technical University, Gothenburg, 1986.

[Lin85] Lindstrom, G., "Functional programming and the logical variable," *Proc. Twelfth ACM Sympos. on Principles of Programming Languages*, New Orleans, 1985, pp. 266-280.

[Lin86] Lindstrom, G., "Implementing logical variables via graph reduction," *Proceedings of the 1986 Graph Reduction Workshop*, Santa Fe, New Mexico, 1986.

[Mil78] Milner, R., "A theory of type polymorphism in programming," *J. Computer and System Sciences*, vol. 17(1978), pp. 348-375.

[Nai85] Naish, L., "Automating control for logic programs," *J. Logic Programming*, vol. 2, 3 (1985), pp. 167-183.

[Red86] Reddy, U. S., "On the relationship between functional and logic programming," in *Logic Programming: Functions, Relations and Equations*, D. DeGroot and G. Lindstrom (ed.), Prentice-Hall, 1986, pp. 3-36.

[Smo86] Smolka, G., "FRESH: a higher order language with unification and multiple results," in *Logic Programming: Functions, Relations and Equations*, D. DeGroot and G. Lindstrom (ed.), Prentice-Hall, 1986, pp. 469-523.

[SuY86] Subrahmanyam, P. A. and You, J., "FUNLOG: a computational model integrating logic programming and functional programming," in *Logic Programming: Functions, Relations and Equations*, D. DeGroot and G. Lindstrom (ed.), Prentice-Hall, 1986, pp. 157-198.

[Wad86] Wadler, P., "Fixing a space leak with a garbage collector," *Proceedings of the 1986 Graph Reduction Workshop*, Santa Fe, New Mexico, 1986.