A Proof-Stream Semantics for Lazy Narrowing

Paul Hudak and Juan Carlos Guzman

Research Report YALEU/DCS/RR-446
December 1985

Work supported in part by NSF Grant DCR-8403304 and Fundacion Gran Mariscal de Ayacucho (fellowship contract F-EX-84.1044)

A Proof-Stream Semantics for Lazy Narrowing

Paul Hudak and Juan Carlos Guzmán

Research Report YALEU/DCS/RR-446
December 1985

Yale University
Department of Computer Science
Box 2158 Yale Station
New Haven, CT 06520

This research was supported in part by Fundación Gran Mariscal de Ayacucho
(fellowship contract F-EX-84.1044)
and the National Science Foundation (grant DCR-8403304).

A Proof-Stream Semantics for Lazy Narrowing

Paul Hudak Juan Carlos Guzmán

December 1985

Yale University
Department of Computer Science[1]

Abstract

Proof-streams provide a convenient way to express the operational semantics of languages with logical variables and backtracking, and indeed can be viewed as the abstract machine of some existing Prolog interpreters. In this paper we use proof-streams in a denotational setting to express the operational semantics of Prolog and a simple language based on *lazy narrowing*. Narrowing can be viewed as a generalization of *reduction* in the lambda calculus that permits logical variables. Lazy narrowing is simply a non-strict version of narrowing akin to lazy evaluation in functional languages. The non-strict semantics introduces non-trivial complications that are captured in the proof-stream semantics using delayed objects called *conditions*.

1 Introduction

In recent years there have been many efforts at combining functional and logic programming languages in such a way as to capture the best characteristics of both. The most obvious approach is to design a language that allows both functional and logical expressions, and that provides an appropriate interface between them. We refer to such solutions as *hybrid*, and they include the languages LOGLISP[20], QLOG[13], FPL[3], Qute[21], POPLOG[17], FUNLOG[24], and LEAF[2].

Although hybrid languages are practical, and achieve the goal of being able to write programs using either, or both, programming paradigms, they are unsatisfactory for several reasons. First, from the viewpoint of programming style, such languages require the programmer to "shift" between "thinking logically" and "thinking functionally." Surely

[1]This research was supported in part by Fundación Gran Mariscal de Ayacucho (fellowship contract F-EX-84.1044) and the National Science Foundation (grant DCR-8403304).

this is undesirable when writing a program, and is equally so from a standpoint of program readability and maintainability. Furthermore, this dichotomy in the language is usually manifested directly in the semantics, where rather different semantic concepts are required to capture each evaluation strategy. When coupled with the interface specification, the resulting semantics becomes rather unwieldy.

A seemingly better approach is to develop a language with a single coherent evaluation strategy, reflected directly in the semantics, that captures the best characteristics of both classes of languages. We refer to such solutions as *direct*, and they include languages such as EQ-LOG[8], TABLOG[16], and the languages found in [5,9,18]. In this paper we explore the operational semantics of a particularly promising direct approach called *narrowing*. Narrowing was first introduced by Fay[7] in the context of equational theories, and is also the basis of work found in [11,14,22]. More recently it has been studied by Reddy in the context of functional languages, where more efficient implementations are possible than with unrestricted equational languages [18,19]. It has now been used as the operational semantics of at least two language implementations [5,15].

Before describing narrowing further, we might ask just what are the "best characteristics" of functional and logic programming languages? Clearly one wishes to have the power of the *logical variable* [26] — it is noticeably absent from functional languages. One would also like to have the *directionality* afforded by reduction in functional languages — not only does this provide the opportunity for more efficient programs, but it also allows expressing functions (or predicates) in their intended mode (without extraneous annotations). A third useful feature is *lazy evaluation*, permitting computation with unbounded data structures, and found most typically in functional programming languages. Although there are other possible features of interest, these are the ones that we concentrate on in this paper, and indeed are not found collectively in any existing programming language.

2 Narrowing

Following [18], we refer to a language whose operational semantics is narrowing as an *N-language*, and furthermore we refer to an N-language with non-strict, or lazy semantics as an *NL-language*.

Prolog and N-languages are similar in that both support logical variables through unification (this is essentially what differentiates an N-language from a conventional functional language). However, there is an important difference: a Prolog program is represented as a set of Horn clauses which are interpreted as statements in first-order logic, whereas an N-language program is a set of equations interpreted as equivalences between values in some primitive domains. Operationally, even though an N-language supports logical variables, its semantics is primarily one of reduction, whereas Prolog is based on resolution.

An N-language is more "expressive" than a functional language because it supports logical variables. In conventional functional languages equation choice is based on pattern-matching rather than unification. With pattern-matching, the reducing expression passes information to the reduced one in the form of actual parameters, but the reduced one can only pass back the result of its evaluation. An N-language, on the other hand, replaces pattern-matching with unification, allowing the well-known "two-way communication" between logical variables.

An N-language is also more "expressive" than Prolog. Indeed, any Prolog program can be trivially translated to an equivalent N-language program by interpreting the Horn clauses as rewrite rules manipulating values in the Boolean domain. For example, consider the following prolog program:

$$\begin{aligned} append([],C,C). & \\ append([A|B],C,[A|D]) &:- append(B,C,D). \\ reverse([],[]). & \\ reverse([A|B],C) &:- reverse(B,D), append(D,[A],C). \end{aligned}$$

which, when translated into an N-language, becomes:

$$\begin{aligned} append([],C,C) &= true. \\ append(A\hat{}B,C,A\hat{}D) &= append(B,C,D). \\ reverse([],[]) &= true. \\ reverse(A\hat{}B,C) &= and(reverse(B,D),append(D,[A],C)). \\ and(true,B) &= B. \end{aligned}$$

(Note that we use $\hat{}$ to denote list construction in an N-language.) As this example suggests, any Prolog clause can be translated into a boolean-valued equation. But one of the powers of an N-language is that a program may be written in a variety of other ways; in particular, ways that reflect the intended input-output "mode" of the logical variables. For example, the above program can be written as:

$$\begin{aligned} append([],C) &= C. \\ append(A\hat{}B,C) &= A\hat{}append(B,C). \\ reverse([]) &= []. \\ reverse(A\hat{}B) &= append(reverse(B),[A]). \end{aligned}$$

which, in fact, is a typical functional program, but has no direct counterpart in Prolog.

When performing reduction in the lambda calculus, there are several strategies for choosing the next expression to reduce [23]. The most common strategy is *applicative-order reduction*, and is used in languages with "strict," or call-by-value semantics (operationally,

Prolog uses such a strategy). But a more powerful strategy is *normal-order reduction*, which is used in modern functional languages with "lazy," or call-by-need semantics, such as SASL[25], ALFL[10], and FEL[12]. A similar choice in reduction-order arises in narrowing: N-languages use *applicative-order narrowing*, and NL-languages use *normal-order narrowing*. NL-languages thus bring to a language with logical variables the standard benefits of lazy evaluation: one may write programs with "infinite data structures," one gains certain run-time efficiencies because "evaluate only when necessary" often amounts to no evaluation at all, and one is freed from concerns about sequencing that typically clutter up conventional programs [6].

3 Semantics

There are several suitable frameworks for expressing the operational semantics that we require. We have chosen a method based on *proof-streams*[4], in which the set of all "proofs" of a goal is represented as a "lazy list," or *stream*, of environments, each environment containing the bindings necessary for one proof. Demand-driven evaluation of the list provides a convenient tool for expressing backtracking, where exploration of a particular branch in the search tree directly corresponds to evaluation of one of the elements of the proof-stream.[2] Another advantage is that a proof-stream semantics lends itself well to the construction of an interpreter. Indeed, we have built interpreters that exactly mimic both of the semantics presented in this paper, and proof-streams were first used (to our knowledge) to derive a Prolog interpreter in an introductory text on programming [1].

Notationally, to express proof-stream semantics, we adopt the standard conventions of denotational semantics: an abstract syntax is defined, semantic domains are established, and semantic functions are given mapping syntax to elements of appropriate semantic domains. This provides a clean specificational tool that frees us from the intricacies of a particular programming language (such as Lisp as used in [4]). Streams are constructed using $\hat{}$ and $\hat{}\hat{}$, infix operators for (lazy) *cons* and *append*, respectively. We have also found the following two mapping functions to be quite useful with proof-streams. The first is the traditional map, defined by:

$$\begin{aligned} map\ f\ \langle\rangle &= \langle\rangle \\ map\ f\ a\hat{}seq &= (f\ a)\hat{}(map\ f\ seq) \end{aligned}$$

The other is different only in that the elements returned from the function application are themselves lists, and are thus appended together:

$$\begin{aligned} app\_map\ f\ \langle\rangle &= \langle\rangle \\ app\_map\ f\ a\hat{}seq &= (f\ a)\hat{}\hat{}(app\_map\ f\ seq) \end{aligned}$$

[2] A more primitive encoding might use *continuations* to explicitly represent the search; however we find streams simpler and more intuitive.

3.1 Proof-Stream Semantics for Prolog

To introduce the reader to the notion of proof-stream semantics, we first use it to describe the operational semantics of a simple version of Prolog. This has the added advantage of being able to make meaningful comparisons between resolution-based and narrowing-based semantics.

Abstract Syntax

$$\begin{aligned} Pair &::= cons\ Expr\ Expr \\ Expr &::= Log\_Var \mid Const \mid Pair \\ goal \in Pred &::= Const\ Sexpr^* \\ c \in Clause &::= Pred\ Pred^* \\ P \in Program &::= Clause^* \end{aligned}$$

Note that we do not allow *cut* in this syntax, nor are we concerned with the semantics of pre-defined predicates. However both of these could be added without much difficulty; see, for example, [4].

Semantic Domains and Auxiliary Functions

$$\begin{aligned} \kappa \in Context &= Naturals \\ e \in Dexpr &= Expr \times Context \\ goal \in Dpred &= Pred \times Context \\ \theta \in Env &= Expr \rightarrow (Expr + \{unbound\}) \\ \pi \in Prf\_Strm &= \{\langle\rangle\} + (Env \times Prf\_Strm) \end{aligned}$$

Note that syntactic, or static expressions ($Expr$) and predicates ($Pred$) are "coerced" into run-time, or dynamic expressions ($Dexpr$) and predicates ($Dpred$) by attaching a "context,"[3] which we represent as a natural number. Env is the domain of environments, and Prf_Strm is the domain of proof-streams. Environments map logical variables to expressions, but for convenience they also behave as the identity function when applied to something other than a logical variable. More specifically, application of an environment θ to an expression e is defined by:

$\theta\ e$ = *unbound*, if e is a logical variable but has no binding in θ

$\theta\ e$ = *exp*, if e is a logical variable whose value in θ is *exp*

$\theta\ e$ = e, otherwise

Note that although θ indirectly stores the values of all structures, in order to completely dereference a list recursive calls to θ are needed for all its components.

[3] This is a more-or-less standard implementation technique.

void is a special environment that denotes "failure." Its definition is not important as long as it is a unique, distinguishable environment. It may be considered as $\top_{Env}$ (the top element in the domain of environments). Furthermore, *initenv* denotes the initial environment where all variables are unbound (consider this as $\bot_{Env}$). More formally:

$$\begin{aligned} initenv\ e &= unbound, \text{ if } logvar?(e) \\ initenv\ e &= e, \text{ otherwise} \end{aligned}$$

The functions *hd* and *tl* have type $(Pair \times Context) \rightarrow Dexpr$ and are defined by:

$$\begin{aligned} hd\ \langle cons\ s_1\ s_2, \kappa \rangle &= \langle s_1, \kappa \rangle \\ tl\ \langle cons\ s_1\ s_2, \kappa \rangle &= \langle s_2, \kappa \rangle \end{aligned}$$

Finally, the function *to_pair* is defined to coerce dynamic predicates (*Dpred*) into dynamic pairs (in *Dexpr*). (The details are omitted.)

Semantic Functions

$$\begin{aligned} \mathcal{U}: &\quad Expr \rightarrow Expr \rightarrow Env \rightarrow Env \\ \mathcal{S}: &\quad Pred^* \rightarrow Program \rightarrow Env \rightarrow Prf\_Strm \\ \mathcal{R}: &\quad Pred \rightarrow Clauses^* \rightarrow Program \rightarrow Env \rightarrow Prf\_Strm \\ \mathcal{P}: &\quad Pred^* \rightarrow Program \rightarrow Prf\_Strm \end{aligned}$$

$\mathcal{U}\ e_1\ e_2\ \theta$ returns a new environment resulting from the unification of e_1 and e_2 in environment θ. $\mathcal{S}\ goals\ P\ \theta$ returns a proof-stream representing solutions to the conjunctive goal *goals* in program P with environment θ. Similarly, $\mathcal{R}\ goal\ c^*\ P\ \theta$ returns a proof-stream for the resolution of *goal* with respect to clauses c^* in program P. Finally, $\mathcal{P}$ is a top-level function for evaluating a set of goals with respect to an entire program.

Semantic Equations

$$\begin{aligned} \mathcal{U}\ e_1\ e_2\ \theta &= \theta[e_2/e_1], && \text{if } \theta e_1 = unbound \\ \mathcal{U}\ e_1\ e_2\ \theta &= \theta[e_1/e_2], && \text{if } \theta e_2 = unbound \\ \mathcal{U}\ e_1\ e_2\ \theta &= \mathcal{U}\ (hd\ (\theta e_1))\ (hd\ (\theta e_2))\ (\mathcal{U}\ (tl\ (\theta e_1))\ (tl\ (\theta e_2))\ \theta) \\ &&& \text{if } pair?(\theta e_1) \text{ and } pair?(\theta e_2) \\ \mathcal{U}\ e_1\ e_2\ \theta &= \theta, && \text{if } const?(\theta e_1) \text{ and } \theta e_1 = \theta e_2 \\ \mathcal{U}\ e_1\ e_2\ \theta &= void, && \text{otherwise} \end{aligned}$$

$$\begin{aligned} \mathcal{S}\ goals\ P\ void &= \langle\rangle \\ \mathcal{S}\ []\ P\ \theta &= \langle\theta\rangle \\ \mathcal{S}\ goal\hat{}rest\ P\ \theta &= app\_map\ (\mathcal{S}\ rest\ P)\ (\mathcal{R}\ goal\ P\ P\ \theta) \end{aligned}$$

$$
\begin{aligned}
\mathcal{R}\ goal\ []\ P\ \theta &= \langle\rangle \\
\mathcal{R}\ goal\ c\hat{}rest\ P\ \theta &= \text{let } \kappa = newcontext() \\
&\quad\ head\hat{}body = c \\
&\quad \text{in } (\mathcal{S}\ \langle body, \kappa\rangle\ P\ (\mathcal{U}\ to\_pair(goal)\ to\_pair(\langle head, \kappa\rangle)\ \theta))\hat{}\hat{} \\
&\quad\ (\mathcal{R}\ goal\ rest\ P\ \theta)
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{P}\ goals\ P &= \text{let } \kappa = newcontext() \\
&\quad \text{in } \mathcal{S}\ \langle goals, \kappa\rangle\ P\ initenv
\end{aligned}
$$

To one familiar with the semantics of Prolog, the above equations should be fairly self-explanatory. One subtle point is that *newcontext*() is intended to return a new context unique from any used thus far.[4] Note the use of the higher-order function $\mathcal{S}$ *rest* P with *app_map* to give a concise formulation for trying all clauses in a program. If one takes the definition of $\mathcal{U}$ as "given," then the above equations for $\mathcal{S}$, $\mathcal{R}$, and $\mathcal{P}$ yield a rather concise semantics for Prolog.

3.2 Proof-Stream Semantics for an NL-Language

Simply stated, to narrow an expression is to make a minimal substitution such that the resulting expression is reducible, and then to reduce it. In an N-language, narrowing comes about when a function is about to be applied (i.e., the formal parameters are narrowed with the actual parameters), and the reductions are specified by the equations. Reddy [18] provides such a semantics, and it can be cast easily into the proof-stream framework as we did for Prolog. Rather than do that directly, however, we choose to proceed to the semantics of an NL-language, in which the introduction of lazy evaluation induces non-trivial complications.

In order to accomplish lazy narrowing we must first abandon Prolog's depth-first proving strategy, so that we can guarantee that *no single branch of an expression is infinitely evaluated without trying other branches.* We accomplish this by "interleaving" the evaluations of all of the subexpressions of a given expression. This, in turn, requires a way to "suspend" the evaluation of an expression, which we accomplish by introducing the notion of "environment conditions," or just *conditions.* Conditions carry all the delayed evaluations of expressions that are needed in order to generate the ultimate environment that is the correct (intended) interpretation of the query. Conditions are "resumed" in the future with a new environment when subsequent evolution of the solution demands it.

[4] The purely functional among us will have to forgive our sloppiness here!

Abstract Syntax

$s \in Sexpr$	::=	$Const \mid LogVar \mid Pair \mid Applic$	Syntactic Expressions
$Pair$	::=	$cons\ Expr\ Expr$	Pairs
$Applic$	::=	$Expr^+$	Applications
$Equation$	::=	$LHS = RHS$	Equations
$lhs \in LHS$	::=	$Id\ Pattern^*$	Left Hand Sides
$Pattern$	::=	$Const \mid LogVar \mid cons\ Pattern\ Pattern$	LHS patterns
$rhs \in RHS$	::=	$Sexpr$	Right Hand Sides
$P \in Program$	::=	$Equation^*$	NL Programs

Semantic Domains and Auxiliary Functions

$\kappa \in Context$	=	$Naturals$	Contexts
$e \in Dexpr$	=	$Expr \times Context$	Expressions
$\theta \in Env$	=	$Dexpr \rightarrow (Dexpr + \{unbound\})$	Environments
$\sigma \in Cond$	=	$CondEnv \rightarrow CPrfStrm$	Conditions
$\Sigma \in CondSet$	=	$Cond^*$	Condition Sets
$CondEnv$	=	$Env \times CondSet$	Conditioned Environments
$PrfStrm$	=	$\{\langle\rangle\} + (Env \times Prf\_Strm)$	P.S. w/o Conditions
$\pi \in CPrfStrm$:	=	$\{\langle\rangle\} + (CondEnv \times CPrfStrm)$	P.S. with Conditions

Because of the special action needed when evaluating applications in an NL-language, we allow an environment to contain updateable bindings for applications as well as logical variables. One can think of each application node in the parse tree as having a special "tag" whose value is looked up in the environment, but we leave out such detail here. We simply extend the definition of an environment given earlier to:

$\theta\ e$ = *unbound*, if e is a logical variable but has no binding in θ

$\theta\ e$ = *exp*, if e is a logical variable *or application* whose value in θ is *exp*

$\theta\ e$ = e, otherwise

initenv is exactly as defined in the previous section.

Note that there are two kinds of proof-streams: ones with conditions ($CPrfStrm$) and ones without ($PrfStrm$). An element $\langle\theta, \Sigma\rangle$ of a proof-stream $\pi \in CPrfStrm$ essentially means that θ is a correct proof contingent upon the complete evaluation of the conditions in Σ.

Semantic Functions

$\mathcal{N}$:	$Dexpr \rightarrow Dexpr \rightarrow Program \rightarrow CondEnv \rightarrow CPrfStrm$	Narrowing
$\mathcal{R}$:	$Dexpr \rightarrow Program \rightarrow CondEnv \rightarrow CPrfStrm$	Reduction
$\mathcal{E}$:	$Dexpr \rightarrow Program \rightarrow CondEnv \rightarrow CPrfStrm$	Evaluation
$\mathcal{E}_c$:	$CPrfStrm \rightarrow PrfStrm$	Evaluate Conditions
$\mathcal{P}$:	$Expr \rightarrow Program \rightarrow PrfStrm$	Evaluate Program

Semantic Equations

$$
\begin{array}{lcll}
\mathcal{N}\ e_1\ e_2\ P\ \langle\theta,\Sigma\rangle & = & \langle\langle\theta[e_2/e_1],\Sigma\rangle\rangle & \text{if } \theta e_1 = \mathit{unbound} \\
\mathcal{N}\ e_1\ e_2\ P\ \langle\theta,\Sigma\rangle & = & \langle\langle\theta[e_1/e_2],\Sigma\rangle\rangle & \text{if } \theta e_2 = \mathit{unbound} \\
\mathcal{N}\ e_1\ e_2\ P\ \langle\theta,\Sigma\rangle & = & \mathit{app\_map}\ (\mathcal{N}\ (\mathit{tl}\ (\theta e_1))\ (\mathit{tl}\ (\theta e_2))\ P) & \text{if } \mathit{pair?}(\theta e_1) \\
 & & \qquad (\mathcal{N}\ (\mathit{hd}\ (\theta e_1))\ (\mathit{hd}\ (\theta e_2))\ P\ \langle\theta,\Sigma\rangle) & \text{and } \mathit{pair?}(\theta e_2) \\
\mathcal{N}\ e_1\ e_2\ P\ \langle\theta,\Sigma\rangle & = & \mathit{app\_map}\ (\mathcal{N}\ e_1\ e_2\ P)\ (\mathcal{R}\ e_1\ P\ \langle\theta,\Sigma\rangle) & \text{if } \mathit{applic?}(\theta e_1) \\
\mathcal{N}\ e_1\ e_2\ P\ \langle\theta,\Sigma\rangle & = & \mathit{app\_map}\ (\mathcal{N}\ e_1\ e_2\ P)\ (\mathcal{R}\ e_2\ P\ \langle\theta,\Sigma\rangle) & \text{if } \mathit{applic?}(\theta e_2) \\
\mathcal{N}\ e_1\ e_2\ P\ \langle\theta,\Sigma\rangle & = & \langle\langle\theta,\Sigma\rangle\rangle & \text{if } \mathit{const?}(\theta e_1) \\
 & & & \text{and } \theta e_1 = \theta e_2 \\
\mathcal{N}\ e_1\ e_2\ P\ \langle\theta,\Sigma\rangle & = & \langle\rangle & \text{otherwise}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{R}\ e\ P\ \langle\theta,\Sigma\rangle & = & \text{let } \kappa = \mathit{newcontext}() \\
 & & \quad f = \lambda(\mathit{lhs} = \mathit{rhs}). \\
 & & \qquad \mathcal{N}\ \langle \mathit{lhs},\kappa\rangle\ e\ P\ \langle\theta[\langle \mathit{rhs},\kappa\rangle/e],\Sigma\rangle \\
 & & \text{in } (\mathit{app\_map}\ f\ P)
\end{array}
$$

$$
\begin{array}{lcll}
\mathcal{E}\ e\ P\ \langle\theta,\Sigma\rangle & = & \langle\langle\theta,\Sigma\rangle\rangle & \text{if } \mathit{const?}(\theta e) \\
 & & & \text{or } \theta e = \mathit{unbound} \\
\mathcal{E}\ e\ P\ \langle\theta,\Sigma\rangle & = & \langle\langle\theta,\Sigma\hat{}\hat{}\langle(\mathcal{E}\ (\mathit{hd}\ (\theta e))\ P),(\mathcal{E}\ (\mathit{tl}\ (\theta e))\ P)\rangle\rangle\rangle & \text{if } \mathit{pair?}(\theta e) \\
\mathcal{E}\ e\ P\ \langle\theta,\Sigma\rangle & = & \text{let } f = \lambda\langle\theta,\Sigma\rangle.\langle\theta,\Sigma\hat{}\hat{}\langle\mathcal{E}\ e\ P\rangle\rangle & \\
 & & \text{in } (\mathit{map}\ f\ (\mathcal{R}\ e\ P\ \langle\theta,\Sigma\rangle)) & \text{if } \mathit{applic?}(\theta e)
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{E}_c\ \langle\rangle & = & \langle\rangle \\
\mathcal{E}_c\ \langle\theta,\langle\rangle\rangle\hat{}\pi & = & \theta\hat{}(\mathcal{E}_c\ \pi) \\
\mathcal{E}_c\ \langle\theta,\sigma\hat{}\Sigma\rangle\hat{}\pi & = & (\mathcal{E}_c\ (\sigma\ \langle\theta,\Sigma\rangle))\hat{}\hat{}(\mathcal{E}_c\ \pi)
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{P}\ s\ P & = & \text{let } \kappa = \mathit{newcontext}() \\
 & & \text{in } \mathcal{E}_c\ (\mathcal{E}\ \langle s,\kappa\rangle\ P\ \langle \mathit{initenv},\langle\rangle\rangle)
\end{array}
$$

Explanation of Equations

The predicates *pair*?, *applic*? and *const*? return true if their argument (in *Expr*) is derived from the syntactic domains *Pair*, *Applic* and *Const*, respectively.

Narrowing ($\mathcal{N}$). Narrowing an unbound logical variable v with another expression e simply returns an environment in which $v = e$. Lists are narrowed by recursively narrowing their heads and tails — note how the use of *app_map* simplifies this task. Narrowing an application with an expression amounts to first reducing the application to the right-hand-side of some (matching) equation, and then continuing the narrowing process on the resulting proof-stream (again by using *app_map*).

Reduction ($\mathcal{R}$). The reduction of an expression e (which must be an application) is accomplished by finding an equation ϵ whose left-hand-side successfully narrows with e, and

returning an environment where e is then bound (with appropriate substitutions) to the right-hand-side of ϵ.[5] Of course, this is all done at the proof stream level, since ultimately several solutions may be found. It is important to note that reduction performs only one step (namely, the binding of formal parameters to actual parameters in an application) – this is crucial to maintaining the lazy evaluation semantics.

Evaluation ($\mathcal{E}$). The evaluation of a constant or unbound variable immediately succeeds. The evaluation of a list, on the other hand, is immediately delayed (by adding conditions for the evaluation of the head and tail to the existing list of conditions). The evaluation of an application is somewhere in between – it results in a proof-stream representing the evaluation of all possible (one-step) reductions.

Evaluation of conditions ($\mathcal{E}_c$). $\mathcal{E}_c$ simply "coerces" a proof stream with conditions to one without; it is only used at the top-level of program evaluation (i.e., with $\mathcal{P}$). When a condition is resumed it is done so not only in the current environment (instead of the one in effect when the expression was suspended), but also under the contingency of the current conditions. This is why a condition is a function from environment/condition pairs to proof-streams. The resumption of a condition, of course, yields another proof stream, since several proofs may be found. Thus the coercion involves appending these proof streams together.

Program evaluation ($\mathcal{P}$). A program is run by evaluating it in the empty environment *initenv* with no conditions, and then using $\mathcal{E}_c$ to force the evaluation of any conditions that result.

3.3 Comparison of Prolog and NL-language Semantics

A rough analogy between the proof-stream semantics for an NL-language and that given for Prolog is that $\mathcal{P}$, $\mathcal{E}_c$, and $\mathcal{E}$ in the former are collectively like $\mathcal{P}$ plus $\mathcal{S}$ in the latter, $\mathcal{R}$ (reduce) is like $\mathcal{R}$ (resolve), and $\mathcal{N}$ is like $\mathcal{U}$. With this analogy one should note that in Prolog resolution depends on unification, but unification depends only on itself. Also, by the nature of unification, only one most general unifier is found. With an N-language, in contrast, narrowing and reduction are mutually dependent and narrowing can return more than one result.

[5] Technically, we need to coerce expressions and left-hand-sides to the same type of objects — hopefully the context makes this clear.

References

[1] H. Abelson, G.J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press and McGraw-Hill, 1985.

[2] R. Barbutti, M. Bellia, G. Levi, and M. Martelli. *LEAF: A language which integrates logic, equations, and functions*, page . Prentice-Hall, 1985.

[3] M. Bellia, P. Degano, and G. Levi. *Call by name semantics of a clause language with functions*, pages 189–198. Academic Press, 1982.

[4] M. Carlsson. On implementing prolog in functional programming. In *Int'l Sym. on Logic Prog.*, pages 154–159, IEEE, February 1984.

[5] J. Darlington, A.J. Field, and H. Pull. *The Unification of Functional and Logic Languages.* Technical Report, Department of Computing, Imperial College of Science and Technology, February 1985.

[6] J. Darlington, P. Henderson, and D.A. Turner, editors. *Functional Programming and its Applications.* Cambridge University Press, Cambridge, England, 1982.

[7] M. Fay. First-order unification in an equational theory. In *Fourth Workshop on Automated Deduction*, pages 161–167, 1979.

[8] J.A. Goguen and J. Meseguer. Equality, types,, modules and generics for logic programming. In *Proc. 2nd Int'l Logic Prog. Conf.*, pages 115–125, 1984.

[9] A. Hansson, S. Haridi, and S-A. Tarnlund. *Properties of a logic programming language*, pages 267–280. Academic Press, 1982.

[10] P. Hudak. *ALFL Reference Manual and Programmer's Guide.* Research Report YALEU/DCS/RR-322, Second Edition, Yale University, October 1984.

[11] J-M. Hullot. Canonical forms and unification. In *Conf. Automated Deduction*, pages 318–334, 1980.

[12] R.M. Keller. *FEL programmer's guide.* AMPS TR 7, University of Utah, March 1982.

[13] H.J. Komorowski. *QLOG — The programming environment PROLOG in LISP*, pages 315–323. Academic Press, 1982.

[14] D.S. Lankford. *Canonical inference.* Technical Report ATP-32, Univ. of Texas at Austin, 1975.

[15] G. Lindstrom. Functional programming and the logical variable. In *Proc. 11th Sym. on Prin. of Prog. Lang.*, pages 266–280, ACM, January 1985.

[16] Y. Malachi, Z. Manna, and R. Waldinger. Tablog: the deductive-tableau programming language. In *Proc. Sym. LISP and Functional Prog.*, pages 323–330, ACM, August 1984.

[17] C. Mellish and S. Hardy. *Integrating PROLOG in the POPLOG environment*, pages 147–162. Ellis Horwood, 1984.

[18] U.S. Reddy. Narrowing as the operational semantics of functional languages. In *Submitted to Int'l Sym. on Logic Prog.*, IEEE, 1985.

[19] U.S. Reddy. *On the relationship between logic and functional languages*, page . Prentice-Hall, 1985.

[20] J.A. Robinson and E.E. Sibert. *LOGLISP: motivation, design and implementation*, pages 299–314. Academic Press, 1982.

[21] M. Sato and T. Sakurai. Qute: a prolog/lisp type language for logic programming. In *Eigth Int'l Joint Conf. Artificial Intelligence*, 1983.

[22] J.R. Slagle. Automated theorem-proving for theories with simplifiers. *JACM*, 21(4):622–642, 1974.

[23] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, Cambridge, Mass., 1977.

[24] P.A. Subrahmanyam and J-H. You. Funlog = functions + logic: a computational model integrating functional and logic programming. In *Int'l Sym. Logic Prog.*, pages 144–153, IEEE, 1984.

[25] D.A. Turner. *SASL language manual.* Technical Report, University of St. Andrews, 1976.

[26] D. Warren, L.M. Pereira, and F. Pereira. Prolog — the language and its implementation compared with lisp. *SIGPLAN Notices*, 12(8):, 1977.