Completeness of an Operational Semantics for First-Order Lazy Narrowing

Juan C. Guzmán and Paul Hudak

Research Report YALEU/DCS/RR-525
July 1987

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

Completeness of an Operational Semantics for First-Order Lazy Narrowing

Juan C. Guzmán and Paul Hudak

July 1987

Yale University
Department of Computer Science[1]

Abstract

We have studied several evaluation strategies suitable for *lazy narrowing* — a generalization of normal-order reduction in the lambda calculus that permits logical variables. We propose an operational semantics based on *proof-streams* to model this computational paradigm. We show that, in general, languages based on narrowing are not confluent — this is a non-intuitive result derived from the fact that narrowing has the "side-effect" of refining environments. However, we further show that if an outermost rewrite strategy is used, resulting in lazy narrowing, the language becomes confluent. Finally, our operational semantics is *complete* in that any solution obtainable by lazy narrowing via a finite number of of reductions and unifications is also obtainable (in finite time) by our semantics, and no inconsistent answers are found.

1 Introduction

The debate over which model of computation is better, logic or functional programming, has been around long enough without either of these paradigms being a clear winner. Each has its advantages and drawbacks compared to the other one, so that several groups have proposed new models of computation that gather the best features of logic and functional programming and avoid their drawbacks. LOGLISP [8], FUNLOG [11], TABLOG [4] and FRESH [9] are just a few of many such attempts. A particularly attractive approach is languages based on *narrowing*, studied most extensively by Reddy [5,6]. We have thus

[1]This research was supported in part by Fundación Gran Mariscal de Ayacucho (fellowship contract F-EX-84.1044) and the National Science Foundation (grants DCR-8403304 and DCR-8302018).

adopted narrowing as the basis of our research in combining logic and functional programming [3], and have concentrated in particular on the operational semantics of *lazy narrowing*, corresponding to normal-order reduction in the lambda calculus.

Narrowing will be introduced more formally in the next section. Informally, it arises in the context of functional languages when the "communication" between formal and actual parameters that occurs at function application is modelled by ***unification***, rather than by pattern-matching. If, in addition, the goal of lazy evaluation is maintained, non-trivial complications arise. Among these is the fact that, unlike in the case of functional languages, there are evaluation strategies that fail to yield *proper results* for some programs. We shall show examples of this shortly.

Our accomplishment can be summarized as follows: First we note that different evaluation strategies applicable to narrowing essentially provide different interpretations to programs, and that the most general evaluation strategy is ambiguous.

Secondly, we provide a formal operational semantics (based on *proof-streams*) for lazy narrowing (one such evaluation strategy). Using proof-streams not only aids reasoning about the semantics, for example in the way described in the next paragraph, but also makes the construction of an interpreter fairly easy, reducing essentially to a direct realization of the proof-stream semantics.

Thirdly we prove that lazy narrowing is a confluent evaluation strategy, i.e., when several reducible expressions may be reduced, any choice will lead to the same set of solutions.

Finally, we prove the said semantics to be *correct* and *complete* with respect to lazy narrowing. We prove that any solution to a program obtainable by lazy narrowing via a finite number of reductions and unifications is also obtainable by the proposed semantics. Conversely, we prove that any solution computable by our semantics can be cast in the lazy narrowing paradigm.

From a global perspective, our research reveals the non-trivial subtleties that arise with lazy narrowing. The operational semantics is considerably more complex than the corresponding semantics of strict narrowing, and our completeness proof is the first that we know of demonstrating the correctness of the algorithm.

In the next section we formally introduce narrowing, lazy narrowing, and provide examples of the behavior of different evaluation strategies on some programs. In Section 3 we propose a semantics for lazy narrowing. We prove in Section 4 that the semantics presented correctly describes this model of computation. Finally, in Section 5, we give some concluding remarks and possible directions for future work.

2 Preliminary Discussion

Narrowing is the semantics that results from extending a typical functional language semantics in such a way that when reducing function applications, the communication established between formal and actual parameters is by unification rather than by pattern-matching. This results in the callee's ability to communicate to its environment by affecting its actual parameters via logical variables. Of course, functions still reduce to values — the only thing that has changed is that these languages allow the introduction of "unknown values" represented by *logical variables.* We will refer to languages based on narrowing as *N-languages*, and to those based on lazy narrowing as *NL-languages.*

For the purpose of the following examples, $[val_1/X_1, ..., val_n/X_n]$ is an environment where X_i is bound to val_i. $\langle\langle s_1, \theta_1\rangle, ..., \langle s_m, \theta_m\rangle\rangle$ represents alternate solutions s_i, each of them associated with environment θ_i, and thus, its own bindings. We refer to this as a *proof-stream* when it represents all possible solutions to a program and goal. Consider the following program:

$$\begin{aligned} append(nil, C) &= C. \\ append(cons(A, B), C) &= cons(A, append(B, C)). \\ reverse(nil) &= nil. \\ reverse(cons(A, B)) &= append(reverse(B), cons(A, nil)). \end{aligned}$$

For the goal

$$append(cons(1, nil), cons(2, nil))$$

a proof-stream that captures the correct behavior of the program is

$$\langle\langle cons(1, cons(2, nil)), [cons(2, nil)/C_2, cons(2, nil)/C_1, 1/A_1, nil/B_1]\rangle\rangle$$

where A_i's, B_i's and C_i's are logical variables introduced by the interpretation of the goal.

2.1 Confluence of Lazy Narrowing

In N- and NL-languages, the reduction of a function application will result not only in the value of the application but also in bindings to logical variables. The use of lexical environments insures that referential transparency of logical variables is retained. However, unlike functional languages, when both applicative-order and normal-order narrowing exist for a given program and goal, the results might differ in that the environments generated by applicative-order narrowing may be more refined than their corresponding normal-order ones. Furthermore, expressions in either of these languages may have more than one possible evaluation, and some of them may be non-terminating.

To illustrate this fact, suppose there is an outermost reduction strategy S_o (only function applications that are not inside any other function application are chosen for reduction). Suppose there is another strategy S_i that is innermost (selects only function applications that do not have other applications as subexpressions). Finally, imagine a strategy S_g which is a more general reduction strategy — the application to be reduced next is selected (possibly non-deterministically) regardless of its relative position in the goal expression.

It is often the case that, when any of these strategies S is repeatedly applied to an expression e, it will evolve into another one $\bar{e}$ that cannot be further reduced. We call $\bar{e}$ an S-normal-form. In the λ-calculus when normal forms are found under the three strategies, they are the same normal form. This is a direct consequence of the Church-Rosser theorem for the λ-calculus [10].

The perspective for narrowing is somewhat different. In contrast to the λ-calculus, in narrowing, an application can be reduced in different ways (by different equations). Thus, there is no notion of unique normal form, but rather, a goal can be reduced to several different normal forms, each of which modifies the environment through instantiation of logical variables in a distinctive way. Still, there is a notion of *confluence*: if there are two function applications to be reduced, the choice of a particular one over another should not affect the solutions found. It is our belief that confluence is a desirable property of any evaluation strategy.

for a certain evaluation strategy to be intuitive, it must be confluent.

Example 1. Consider the following program:

$$\begin{aligned} f(nil) &= nil. \\ f(cons(X,Y)) &= cons(h(g(X,Y),X),f(Y)) \\ g(X,X) &= X. \\ h(X,Y) &= Y. \end{aligned}$$

Evaluation of the goal $f(cons(V_1,cons(V_2,nil)))$ using S_o produces its argument $((cons(V_1,cons(V_2,nil))))$. The goal evaluates to $cons(cons(nil,nil),cons(nil,nil))$ using the S_i strategy. If the strategy used is S_g, the result produced might be either of the previous two, or possibly one of two others, depending on the exact order of reductions chosen. As this example suggests, S_g is not a very "good" evaluation strategy, because its interpretation of a program might be ambiguous; i.e., the strategy is not confluent.

Confluence can be achieved, however, if the reduction strategy is fixed. We advocate S_o, which in λ-calculus is also called *normal order reduction*, for the same reasons it is advocated by the functional programming community. For example, it is a well-known result that, in λ-calculus, S_i is not a *safe* evaluation strategy — there are expressions for which there is a normal form that cannot be obtained via innermost reduction. Since λ-calculus is subsumed by our paradigm, this property is inherited. Also, solutions found

by S_o subsumes solutions found using other strategies because any reduction performed by S_o must also be performed by any other strategy.

Note that S_o, S_i and S_g do not represent fixed evaluation strategies. They represent families of strategies that satisfy certain property, as in the case of S_o "only outermost function application are reduced". Also note that every strategy in family S_o is also in family S_g. In the next sections, we will exhibit a very representative strategy from the family S_o in the sense that this strategy captures the "best" behavior of all S_o strategies. By abuse of language, we may refer to S_o as a single strategy meaning some strategy in this family.

2.2 Lazy Narrowing

The next example highlights the behavior of lazy evaluation in the presence of multiple narrowings of a single expression.

Example 2. Consider the NL-language program:

$$\begin{aligned} f(1,Y) &= 1. \\ f(0,Y) &= Y. \\ g(1) &= g(1). \\ g(0) &= 0. \\ h(0) &= 0. \end{aligned}$$

and the goal $f(g(X),h(X))$. Since h is only defined for 0, any solution to the above goal must narrow X and, therefore, $g(X)$ to 0, and thus the whole expression will be reduced to 0. This is the only correct solution, but how does an evaluator find it?

Suppose the evaluator initially has no binding for X and it chooses to reduce $g(X)$ according to the first equation for g. If the evaluator continues evaluating this subexpression depth-first, it will loop forever. On the other hand, a breadth-first search of solutions (i.e., an interleaving strategy) would also explore the second equation for g, thus discovering the solution mentioned above, and binding X to 0. It will soon discover that there is no other solution. The resulting (and correct) proof-stream can be written: $\langle\langle 0, [0/X]\rangle\rangle$.

Example 3. We use for this example the same program as above, but with the goal $f(g(1),h(1))$. There is clearly no substitution of variables that will satisfy this goal — the proper proof-stream is $\langle\rangle$. This answer will indeed be found if a breadth-first strategy is used in evaluating the subexpressions in the goal, because the evaluation of $g(1)$ would be interleaved with $h(1)$. However, a depth-first strategy will try reducing $g(1)$ and will not terminate, thus never concluding that there are no solutions — the (incorrect) proof-stream returned by such a strategy is effectively $\perp$.

These three examples demonstrate the need for different evaluation mechanisms for a proper (meaning lazy yet complete) evaluation strategy for NL-languages:

1. *Lazy Reduction*: Expressions should only be reduced "on demand." This makes possible the definition of non-strict functions, as in Example 1, and can be achieved by using a normal-order reduction strategy, or more precisely, by always choosing an outermost reducible expression.

2. Completeness is ensured by evaluating the "computational tree" in a *breadth-first* fashion:

   (a) Since evaluation of an expression may occur in environments that will eventually be discarded, and yet the current evaluation of the current environment may be non-terminating, all subexpressions that need to be evaluated should be given a chance to refine the environment, as in Example 3. This can be achieved by evaluating the *goal subexpressions* breadth-first. This technique is similar to AND-parallelism in Prolog.

   (b) In order to be able to enumerate all possible normal forms that an expression may have (as in Example 2) it is necessary to evaluate all possible reductions of any expression breadth-first. This technique is similar to OR-parallelism in Prolog.

   Note the difference between these two items, the former is similar to assert in a functional language that several subexpressions will have their evaluation interleaved. It is also less expensive to implement than the latter; this essentially establishes that the "abstract machine" that evaluates a goal is actually "forked" to evaluate all possible normal forms of the goal. This behavior does not have counterpart in evaluation of λ-calculus expressions.

In the following section a formal semantics will be given that captures these behaviors.

3 Operational Semantics

There are several suitable frameworks for expressing the semantics that we require; we have chosen a method based on *proof-streams* [1]. Intuitively, a proof-stream for a program and a goal is the set of all *proofs* of the goal represented as a "lazy list" (or *stream*) of environments, each containing the bindings necessary for one proof and, for N- and NL-languages, the value of the goal being evaluated. 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

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

construction of an interpreter. Indeed, we have built an interpreter that exactly mimics the proof-stream semantics for an NL-language.

Our semantics, although operational, is expressed in a *denotational* style — the meaning of an expression is given as a composition of the meanings of the expression's constituent components. Thus we adopt the standard notational conventions of denotational semantics (as outlined in [2]): an abstract syntax is defined, semantic domains are established, and semantic functions are given mapping syntax to elements of appropriate semantic domains. Although the utility of denotational semantics in defining the semantics of a wide range of programming languages is well known, its use in defining the semantics of logic programming languages has been limited. Despite this, we feel that its use is justified. The framework provides us with a formal yet clear specification tool that, even in the case of a purely operational semantics, frees us from the intricacies of a particular programming language (such as Lisp as used in [1]) or the ambiguities of algorithms expressed in English (such as is often found in operational descriptions of Prolog). It is especially useful in defining the semantics of languages where reduction and resolution coexist, such as the class of N- and NL-languages that we consider in this paper.

Our notation can be summarized as follows: Double brackets are used to surround syntactic objects, as in "$\mathcal{N}\ [\![e_1]\!]\ [\![e_2]\!]$." A finite sequence of zero or more syntactic elements c is written c^*; the empty sequence is represented by ϵ. Thus $c' = c^*$ is equivalent to $c' = \epsilon \mid c \mid c\ c'$. "Pattern-matching" is often used to destructure the abstract syntax; for example "$\mathcal{P}[\![c\hat{}rest]\!] = body$" destructures an element of c^*.

We write "$d \in D = Exp$" to define the domain (or set) D with "canonical" element d. A sequence of semantic values is called a *stream*, and is defined by a recursive domain equation such as $D = \{\langle\rangle\} + (Bas \times D)$, where $\langle\rangle$ represents the empty stream. A specific stream may be written $\langle d_1, ..., d_n\rangle$, or may be constructed using $\hat{}$ and $\hat{}\hat{}$, infix operators for (lazy) *cons* and *append*, respectively. Pattern-matching is also used to destructure streams, as in "$map\ f\ a\hat{}rest = body$." When necessary, domains are assumed to be chain-complete partial orders; the bottom element in a domain D is denoted $\perp_D$. For simplicity we omit most domain/sub-domain coercions, since the context usually makes the meaning clear. Finally, the traditional *map* function is quite useful with streams, and is defined by:

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

3.1 Abstract Syntax

k	$\in$	$Konst$	Constants
c	$\in$	$Const$	Constructors
f	$\in$	$Funct$	Function names
v	$\in$	$LogVar$	Logical Variables

We assume that each element of *Const* and *Funct* has a fixed arity, and that elements having different arity are never equal. From these primitive syntactic domains the following composite domains are defined:

$$\begin{array}{rcll} e, x, y, r & \in & Expr & ::= \; k \mid v \mid c(e_1, ..., e_n) \mid f(e_1, ..., e_n) \\ \bar{x}, \bar{y} & \in & Expr^* & ::= \; e_1, ..., e_n \\ h & \in & Head & ::= \; f(\bar{x}) \\ eq & \in & Equation & ::= \; h = e \\ P & \in & Program & ::= \; eq^* \end{array}$$

We also require that the left-hand-sides of equations be *pairwise non-narrowable*. This condition ensures *confluence* of the language — once the condition is met, all function applications that have already been reduced and whose parameters narrow must have been reduced by the same equation. Henceforth we assume that all programs are syntactically correct.

3.2 Semantic Domains and Auxiliary Functions

$$\begin{array}{rcll} Value & = & Expr \cup \{\nu\} & \text{Values} \\ \theta \in Env & = & ((LogVar \times LogVar)^* \times (LogVar \to Value)) & \\ & & & \text{Variable Environments} \\ \varphi \in FEnv & = & Funct \to Expr \to Cond \to Env \to (Ctxt \times RPrfStrm) & \\ & & & \text{Function Environments} \\ \kappa \in Ctxt & = & Naturals & \text{Contexts} \\ \sigma \in Cond & = & Ctxt \to State \to PrfStrm & \text{Conditions} \\ \Sigma \in CondQ & = & \{\langle\rangle\} + (Cond \times CondQ) & \text{Condition Queues} \\ State & = & Env \times CondQ & \text{States} \\ \pi \in PrfStrm & = & \{\langle\rangle\} + (State \times PrfStrm) & \text{Proof-Streams} \\ \pi \in RPrfStrm & = & \{\langle\rangle\} + (Expr \times State \times RPrfStrm) & \text{P.S. with Answers} \end{array}$$

Note that the domain *Value* contains ν. This element denotes the value of unbound (logical) variables.

Ctxt is introduced only for variable renaming. Its presence enables us to express α-*conversion* functionally (and thus denotationally).

Env is the domain of *environments*, which associate logical variables with values. Each environment is a pair $\langle R, \mu \rangle$ where R is an equivalence relation among variables (when two variables have been bound to one another they are in the same equivalence class), and μ maps variables to values. Environments are built up from the initial environment θ_{init}:

$$\begin{array}{rcl} \theta_{init} & = & \text{let } R_{init} = \{\langle v, v \rangle | v \in LogVar\} \\ & & \quad \mu_{init} = \lambda\, v.\, \nu \\ & & \text{in } \langle R_{init}, \mu_{init} \rangle \end{array}$$

which is then incrementally updated according to the following definition of environment update:

$$\begin{aligned}
\langle R,\mu\rangle[v_2/v_1] &= \langle R \cup \{\langle \dot{v}_1,\dot{v}_2\rangle | \dot{v}_1 \in [v_1]_R, \dot{v}_2 \in [v_2]_R\}, \mu[\mu v_2/v_{11},...,\mu v_2/v_{1n}]\rangle, \qquad \forall v_{1i} \in [v_1]_R \\
\langle R,\mu\rangle[\hat{x}/v_1] &= \langle R,\mu[\hat{x}/v_{11},...,\hat{x}/v_{1n}]\rangle, \qquad \forall v_{1i} \in [v_1]_R
\end{aligned}$$

$\theta[y/x]$ denotes the environment resulting from updating θ so that x is bound to y. x must be a non-bound variable in θ, y may be any expression. As a result of this update, all variables bound to x in θ will have as its value the value of the expression y in the updated environment. $[v]_R$ denotes the equivalence class of v with respect to R. Note that there is no need for variable chaining with this approach.

Accessing values in an environment is defined via the function *val*:

$$val : Env \rightarrow Expr \rightarrow Value$$

$$val\ \langle R,\mu\rangle\ e = \text{if } e \in LogVar \text{ then } \mu e \text{ else } e$$

This function retrieves the value of a variable in an environment, but for convenience it also behaves as the identity function when applied to something other than a logical variable. Note that $(val\ ((\theta[[\![y]\!]/[\![x]\!]])[1/[\![y]\!]])\ [\![x]\!])$ returns 1, not $[\![y]\!]$.

The breadth-first evaluation strategy (whether for AND- or OR-parallelism) requires not only a representation for delayed evaluations, but also a queue to hold those delayed objects. The former is accomplished by a *condition* ($\in Cond$) that is similar to a *continuation* in conventional denotational semantics. A queue of conditions is represented as a sequence using the domain $CondQ$. Elements of $CondQ$ are used in different places, depending on the type of evaluation being delayed. The *state* of the computation at any time consists of the current environment and current condition queue, and is captured by the domain $State$. The proper way to view a state $\langle\theta,\Sigma\rangle$ is that θ represents a correct solution contingent upon the success of the conditions in Σ.

Note the two kinds of proof-streams: ones with results ($RPrfStrm$) and ones without ($PrfStrm$). Elements of the former type are used to express the outcome of reduction. Elements of $PrfStrm$ express the outcome of narrowing. This agrees with the intuition that the reduction of an expression yields a value and possibly refines the environment, whereas narrowing two expressions only refines the environment. Both types of proof-streams may have conditions but the proof-stream that holds the final answer will have an empty condition queue. The function that coerces proof-streams of states to proof-stream of results and states is $PStoRPS$:

$$PStoRPS : Expr \rightarrow PrfStrm \rightarrow RPrfStrm$$

$$PStoRPS\ [\![e]\!]\ \pi = map\ (\lambda\ \langle\theta,\Sigma\rangle.\ \langle [\![e]\!],\theta,\Sigma\rangle)\ \pi$$

We use the function *app_map* that expects a function, a context and a list, returning an updated context and list. The new list is the result of applying the function to each element of the original list, updating the context each time.

$$\begin{aligned} app\_map\ f\ \kappa\ \langle\rangle &= \langle\rangle \\ app\_map\ f\ \kappa\ a\hat{}rest &= \text{let } \langle \kappa_1, \pi_1 \rangle = f\ \kappa\ a \\ &\qquad \langle \kappa_2, \pi_2 \rangle = app\_map\ f\ \kappa_1\ rest \\ &\quad \text{in } \langle \kappa_2, \pi_1 \hat{}\hat{} \pi_2 \rangle \end{aligned}$$

Finally, define the function $alpha : Ctxt \rightarrow Expr^* \rightarrow (Ctxt \times Expr^*)$ which performs an "α-conversion" of its argument. That is, $(alpha\ \kappa\ e)$ returns an expression identical to e except that all variables have been renamed with unique identifiers not used so far. This is used to prevent name clashes when evaluating the same expression in different contexts.[3]

3.3 Semantic Functions

$\mathcal{N}$:	$Expr \rightarrow Expr \rightarrow FEnv \rightarrow Ctxt \rightarrow State \rightarrow (Ctxt \times PrfStrm)$	Narrowing
$\mathcal{E}_c$:	$Ctxt \rightarrow CPrfStrm \rightarrow (Ctxt \times PrfStrm)$	Evaluate Conditions
$\mathcal{R}$:	$Funct \rightarrow Expr \rightarrow Ctxt \rightarrow State \rightarrow (Ctxt \times RPrfStrm)$	Reduction
$\mathcal{P}$:	$Program \rightarrow Expr \rightarrow Expr \rightarrow PrfStrm$	Evaluate Program

There are two primary semantic functions: $\mathcal{N}$ (narrow) and $\mathcal{R}$ (reduce). $\mathcal{R}$ defines the reduction that takes place when a function application occurs. It uses $\mathcal{N}$ (instead of pattern-matching) to communicate the environments of the caller and callee. $\mathcal{N}$ is similar to unification, except that reduction of function application is permitted. Figure 1 contains the definition of these functions.

A condition is created by the partial application of one of these semantic functions — for example, $\sigma = \mathcal{N}\ [\![e_1]\!]\ [\![e_2]\!]\ \varphi$. Such a condition is later resumed by passing to it the remaining arguments; namely, the context and the state that are current at the time of resumption — for example, $(\sigma\ \kappa\ \langle\theta, \Sigma\rangle)$. This provides a general framework for delaying the actual refinement of the environment.

At first glance the semantics in Figure 1 may appear quite complex, but in fact it is relatively straightforward. Most of the complexity is due to two factors:

1. We have insisted that *contexts* be manipulated *explicitly*. Most other accounts of operational semantics for logic-based languages leave such mechanisms implicit, and

[3] An often-used implementation strategy is to use an increasing sequence of integers for the purpose of generating new variables, although other techniques may be used for this purpose.

$$
\begin{array}{rl}
\mathcal{N}\ [\![e_1]\!]\ [\![e_2]\!]\ \varphi\ \kappa\ \langle\theta,\Sigma\rangle = & \text{case } val\ \theta\ [\![e_1]\!], val\ \theta\ [\![e_2]\!] \text{ of} \\
[\![k]\!], [\![k]\!]: & \langle\kappa, \langle\langle\theta, \Sigma\rangle\rangle\rangle \\
\nu, z: & \langle\kappa, \langle\langle\theta[[\![e_2]\!]/[\![e_1]\!]], \Sigma\rangle\rangle\rangle \\
z, \nu: & \langle\kappa, \langle\langle\theta[[\![e_1]\!]/[\![e_2]\!]], \Sigma\rangle\rangle\rangle \\
[\![f(\bar{y})]\!], [\![x]\!];\ [\![x]\!], [\![f(\bar{y})]\!]: & \text{let } \langle\kappa_1, \pi\rangle = (\varphi\ [\![f]\!]\ [\![\bar{y}]\!]\ \kappa\ \langle\theta, \Sigma\rangle) \\
& \text{in } \langle\kappa_1, map\ (\lambda\ \langle[\![r]\!], \theta, \Sigma\rangle.\ \langle\theta, \Sigma\hat{}\hat{}\langle\mathcal{N}\ [\![x]\!]\ [\![r]\!]\ \varphi\rangle\rangle)\ \pi\rangle \\
[\![c(e_{11}, \ldots, e_{1n})]\!], [\![c(e_{21}, \ldots, e_{2n})]\!]: & \\
& \text{let } \sigma_i = (\mathcal{N}\ [\![e_{1i}]\!]\ [\![e_{2i}]\!]\ \varphi), \qquad i = 1, .., n \\
& \text{in } \langle\kappa, \langle\langle\theta, \Sigma\hat{}\hat{}\langle\sigma_1, ..., \sigma_n\rangle\rangle\rangle\rangle \\
x_1, x_2: & \langle\rangle \qquad\qquad \text{otherwise}
\end{array}
$$

$$
\begin{array}{rl}
\mathcal{E}_c\ \kappa\ \langle\rangle = & \langle\kappa, \langle\rangle\rangle \\
\mathcal{E}_c\ \kappa\ \langle\theta, \langle\rangle\rangle\hat{}\pi = & \text{let } \langle\kappa_1, \pi_1\rangle = (\mathcal{E}_c\ \kappa\ \pi) \\
& \text{in } \langle\kappa_1, \langle\theta, \langle\rangle\rangle\hat{}\pi_1\rangle \\
\mathcal{E}_c\ \kappa\ \langle\theta, \sigma\hat{}\Sigma\rangle\hat{}\pi = & \text{let } \langle\kappa_1, \pi_1\rangle = (\sigma\ \kappa\ \langle\theta, \Sigma\rangle) \\
& \text{in } \mathcal{E}_c\ \kappa_1\ \pi\hat{}\hat{}\pi_1
\end{array}
$$

$$
\begin{array}{rl}
\mathcal{P}\ [\![& f^1(\bar{x}^{11}) = e^{11}, \\
& \ldots, \\
& f^1(\bar{x}^{1t_1}) = e^{1t_1}, \\
& \ldots, \\
& f^n(\bar{x}^{nt_n}) = e^{nt_n}]\!] = \quad \text{let } \varphi = [(\mathcal{R}\ [\![f_1]\!])/f_1, ..., (\mathcal{R}\ [\![f_n]\!])/f_n] \\
& \qquad \mathcal{R}\ [\![f^i]\!] = \lambda\ [\![\bar{y}]\!]\ \kappa\ \langle\theta, \Sigma\rangle. \\
& \qquad\qquad \text{let } g\ \kappa\ \langle[\![\bar{x}]\!], [\![e]\!]\rangle = \\
& \qquad\qquad\qquad \text{let } \langle\kappa_1, \langle[\![\bar{x}']\!], [\![e']\!]\rangle\rangle = alpha\ \kappa\ \langle[\![\bar{x}']\!], [\![e']\!]\rangle \\
& \qquad\qquad\qquad\quad \langle\kappa_2, \pi_2\rangle = (\mathcal{N}\ [\![\bar{x}']\!]\ [\![\bar{y}]\!]\ \varphi\ \kappa_1\ \langle\theta, \Sigma\rangle) \\
& \qquad\qquad\qquad \text{in } \langle\kappa_2, (PStoRPS\ [\![e']\!]\ \pi_2)\rangle \\
& \qquad\qquad \text{in } app\_map\ g\ \kappa\ \langle\langle[\![\bar{x}^{i1}]\!], [\![e^{i1}]\!]\rangle, ..., \langle[\![\bar{x}^{it_i}]\!], [\![e^{it_i}]\!]\rangle\rangle \\
& \text{in } \lambda\ [\![e_1]\!]\ [\![e_2]\!]. \\
& \qquad \text{let } \langle\kappa, \pi\rangle = \mathcal{N}\ [\![e_1]\!]\ [\![e_2]\!]\ \varphi\ \kappa_{init}\ \langle\theta_{init}, \langle\rangle\rangle \\
& \qquad \text{in } \mathcal{E}_c\ \kappa\ \pi
\end{array}
$$

Figure 1: Semantic Functions

if we were to do the same our equations would appear less cluttered. However, since our goal from the outset has been a rigorous proof of completeness, we did not wish to leave any behavior implicit.

2. The mechanism to handle breadth-first evaluation, namely the condition queue, is cumbersome, although the principle upon which it lies, namely to perform incremental evaluation of several paths in "parallel", is quite simple.

Hopefully the following paragraphs will help clarify the meaning of the semantic equations.

As an overview, $\mathcal{N}$ and $\mathcal{R}$ express the semantics of lazy narrowing and reduction respectively. The technique used in both of them is that only a small amount of work is done in each equation before returning a *conditioned proof-stream* that expresses how to finish the work initiated there. No constraints are made on how the conditions in the resulting proof-stream must be executed; this task is left to $\mathcal{E}_c$. $\mathcal{E}_c$ can be thought of as a "scheduler" that interleaves the execution of conditions, not only within one proof (environment) but also interleaving the refinement of several environments. $\mathcal{P}$ gives meaning to a program — it tailors the function environment that associates meaning to function names from the definition of the equations that represent the program. It is in this function environment that expressions are narrowed. In the following paragraphs the semantic functions are described in detail.

Narrowing ($\mathcal{N}$). Narrowing an unbound logical variable (not demanded yet) v with another expression e simply returns an environment in which $v = e$. Narrowing of two objects created by the same constructor (such as a list) causes delayed evaluation of the arguments; this is done by adding conditions for the narrowing of the corresponding arguments to the existing condition queue. Narrowing an application with an expression (including an already demanded variable) amounts to reducing the application to the right-hand-side of some matching equation (via a look-up in φ), and then continuing the narrowing process on the resulting proof-stream. Narrowing of a demanded variable v to either another demanded variable or a constructor e also amounts to returning an environment in which $v = e$. Narrowing of two constants succeeds only if both constants are the same. Otherwise, the narrowing of the expressions fails (returning an empty stream).

Reduction ($\mathcal{R}$). The reduction of an application e is accomplished by finding an equation eq 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 eq. 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 of conditions ($\mathcal{E}_c$). $\mathcal{E}_c$ simply "coerces" a proof stream with conditions to one without. A condition is resumed 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 ($\mathcal{P}$). We have defined as program evaluation a function that takes a program and two expressions and returns the proof-stream of all possible (lazy) narrowings of the two expressions starting with the initial variable environment (θ_{init}) and a function environment that encapsulates the meaning of the program.

4 Correctness of the Semantics

In the previous section we presented an operational semantics for lazy narrowing, but still, we have not formally answered the question *is lazy narrowing a confluent paradigm of computation?* We showed that unrestricted narrowing (following S_g evaluation strategy) is not confluent. Although we presented some examples where lazy narrowing was confluent, we have not proved it in all cases. Furthermore, the presentation of a semantics for such a paradigm raises questions as to how faithfully the semantics represents the paradigm.

In this section we not only address the question of confluence, but we also answer two other questions: (1) *Is the semantics correct, or does it generates solutions that are not computable by lazy narrowing?* (2) *Is the semantics complete, or are there lazy narrowing solutions that cannot be generated by the semantics?* We first introduce some basic definitions, and then attempt to answer these questions.

4.1 Basic Definitions

Suppose we would like to narrow two expressions e_1 and e_2. The natural way to do it is by structurally narrowing subexpressions of e_1 to corresponding subexpressions of e_2. If we view expressions as trees, we would test their roots for *local* equality and, if they are equal, we would proceed to compare respective subexpressions. Our notion of local equality is provided by the following definition of *structural agreement* of two expressions:

Definition 1 *Structural Agreement.* Two expressions *structurally agree* iff either (1) both expressions are the same variable or constant, or (2) both expressions are constructors of the same type with the same number of arguments. Two expressions *structurally disagree* iff they do not structurally agree.

Note that a function application structurally disagrees with all expressions.

References

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

[2] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF (LNCS 78).* Springer-Verlag, New York, 1979.

[3] P. Hudak and J. Guzmán. *A proof-stream semantics for lazy narrowing.* Research Report YALEU/DCS/RR-446, Yale University, Department of Computer Science, December 1985.

[4] 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.

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

[6] U.S. Reddy. On the relationship between logic and functional languages. In *Functional and Logic Programming*, pages 3–36, Prentice-Hall, 1985.

[7] A. Robinson. A machine-oriented logic based on the resolution principle. *JACM*, 12:23–41, 1965.

[8] J.A. Robinson and E.E. Sibert. Loglisp: motivation, design and implementation. In *Logic Programming*, pages 299–314, Academic Press, 1982.

[9] G. Smolka. Fresh: a higher-order language based on unification. In *Functional and Logic Programming*, pages 469–524, Prentice-Hall, 1985.

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

[11] 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.