Functional Logic Languages
Part I
(Preliminary Report)

Uday S. Reddy
University of Illinois at Urbana-Champaign

Abstract

Functional logic languages are extensions of functional languages with principles derived from logic programming. While syntactically they look similar to conventional functional languages, their operational semantics is based on narrowing, an evaluation mechanism that uses unification for parameter passing. We present here a small formal language based on lambda calculus with existential quantification and set abstraction, and define its denotational and narrowing semantics.

1 Introduction

Logic programming languages, typified by Prolog, have come into widespread use over the recent years [CKPR73,Kow79,CM81]. Even though programs written in logic languages exhibit a disarming similarity to functional programs, a deeper analysis shows that logic languages have more expressive power than functional languages [Kow83,Red86a]. The additional expressive power arises from the use of *unification* for parameter passing in logic languages. Subsequently, a number of efforts were made to incorporate this expressive power in functional languages by using unification in their parameter passing mechanisms as well [Lin85,SS84,DFP86,Smo86]. We have shown in [Red85,Red86b] that the result of extending the operational semantics of functional languages with unification amounts to using a mechanism called *narrowing* originally introduced for theorem proving in classical logic with equality [Sla74,Fay79,Hul80]. The languages obtained by such an extension are termed *functional logic languages.*

Let us introduce the concepts of functional logic languages through a simple conventional example: a function for appending lists.

$$\begin{aligned} \mathit{append}\ \mathit{Nil}\ y &\equiv y \\ \mathit{append}\ a.x\ y &\equiv a.(\mathit{append}\ x\ y) \end{aligned}$$

A definition such as this can be written in a number of functional languages such as HOPE [BMS80], Standard ML [Mil84], and Miranda [Tur85]. The use of such a definition in these languages is restricted to the evaluation of ground (variable-free) expressions involving *append*, e.g.,

$$\mathit{append}\ 1.\mathit{Nil}\ 2.\mathit{Nil} \Longrightarrow 1.2.\mathit{Nil}$$

The evaluation method can easily be extended to allow free variables in the input expression in some contexts, e.g.,

$$\mathit{append}\ a.\mathit{Nil}\ 2.\mathit{Nil} \Longrightarrow a.2.\mathit{Nil}$$

But, when we want to evaluate an expression such as

$$\mathit{append}\ t\ 2.\mathit{Nil}$$

no evaluation is possible. The expression is in normal form. However, we would be reluctant to treat the expression as a "literal value", since it has a function application.

When evaluating the expression ($append$ $1.Nil$ $2.Nil$) a conventional functional language *matches* the arguments against the formal parameter patterns. This yields a substitution for the formal parameter variables (e.g. $a = 1$, $x = Nil$, $y = 2.Nil$). A functional logic language, in contrast *unifies* the arguments with functional parameter patterns. This yields a substitution for both the formal parameter variables and the argument variables. The expression ($append\ t\ 2.Nil$) can be unified with the left hand sides of both the equations for *append*, yielding the unifier substitutions:

(1). $t = Nil,\ y = 2.Nil$
(2). $t = a.x,\ y = 2.Nil$

The unifiers can be divided into two parts: an argument substitution σ and a formal parameter substitution θ. The latter applied to the right hand side of the equation becomes the result of the evaluation step. The former is a condition under which the original expression is reducible. So, there are two possible evaluation paths for the expression ($append\ t\ 2.Nil$)

$$\begin{array}{lll} (1)\ append\ t\ 2.Nil & \Longrightarrow & t = Nil \rightarrow 2.Nil \\ (2)\ append\ t\ 2.Nil & \Longrightarrow & t = a.x \rightarrow a.(append\ x\ 2.Nil) \end{array}$$

Such evaluation steps are called *narrowing steps.* The substitutions to the left of $\rightarrow$ in the result expressions are called *narrowing substitutions.* Whenever

$$e \Longrightarrow \sigma \rightarrow e'$$

we say e narrows to e' via the narrowing substitution σ. As is evident, narrowing is a nondeterministic evaluation process. The *append* expression cited above has an infinite number of nondeterministic narrowings.

Semantically, a nonground expression such as ($append\ t\ 2.Nil$) denotes a *parametric value* in the domain $Env \rightarrow Value$

$$[\![append\ t\ 2.Nil]\!]\eta = \begin{cases} 2.Nil & \text{if } \eta t = Nil \\ v_1.2.Nil & \text{if } \eta t = v_1.Nil \\ v_1.v_2.2.Nil & \text{if } \eta t = v_1.v_2.Nil \\ \vdots & \\ \bot & \text{otherwise} \end{cases}$$

Conditional terms of the kind $t = Nil \rightarrow 2.Nil$, and $t = a.x \rightarrow x$ similarly denote parametric values.

$$[\![t = Nil \rightarrow 2.Nil]\!]\eta = \begin{cases} 2.Nil & \text{if } \eta t = Nil \\ \bot & \text{otherwise} \end{cases}$$

$$[\![t = a.x \rightarrow x]\!]\eta = \begin{cases} v_2 & \text{if } \eta t = v_1.v_2 \\ \bot & \text{otherwise} \end{cases}$$

The narrowing process produces a compact (not necessarily finite) representation of the parametric value of a nonground expression in terms of such conditional terms. If the set of all nondeterministic narrowings of an expression e is $\{e \Longrightarrow t_i\}$ then

$$[\![e]\!] = \bigsqcup [\![t_i]\!]$$

Thus the nondeterminism involved in narrowing is quite meaningful and does not imply any kind of referential opacity.

Let us demonstrate the usefulness of narrowing through a less trivial example. Consider the boolean-valued expression

$$(\mathit{append}\ t\ 2.\mathit{Nil}) = 1.2.\mathit{Nil}$$

The narrowing evaluation should produce a solution for t. We show below two nondeterministic narrowing sequences of the expression:

1. $(\mathit{append}\ t\ 2.\mathit{Nil}) = 1.2.\mathit{Nil}$
   $\Longrightarrow (t = \mathit{Nil} \rightarrow 2.\mathit{Nil}) = 1.2.\mathit{Nil}$
   $\Longrightarrow t = \mathit{Nil} \rightarrow (2.\mathit{Nil} = 1.2.\mathit{Nil})$
   failure in evaluating $2.\mathit{Nil} = 1.2.\mathit{Nil}$

2. $(\mathit{append}\ t\ 2.\mathit{Nil}) = 1.2.\mathit{Nil}$
   $\Longrightarrow (t = a.x \rightarrow a.(\mathit{append}\ x\ 2.\mathit{Nil})) = 1.2.\mathit{Nil}$
   $\Longrightarrow t = a.x \rightarrow a.(\mathit{append}\ x\ 2.\mathit{Nil}) = 1.2.\mathit{Nil}$
   $\Longrightarrow t = a.x \rightarrow (a = 1) \wedge (\mathit{append}\ x\ 2.\mathit{Nil}) = 2.\mathit{Nil}$
   $\Longrightarrow t = a.x \rightarrow (a = 1 \rightarrow \mathit{true}) \wedge (x = \mathit{Nil} \rightarrow 2.\mathit{Nil} = 2.\mathit{Nil})$
   $\Longrightarrow t = a.x \rightarrow a = 1 \wedge x = \mathit{Nil} \rightarrow \mathit{true}$
   $\Longrightarrow t = 1.\mathit{Nil} \rightarrow \mathit{true}$

The remaining narrowing sequences all lead to failure. In general, narrowing gives the programming language a capability to *solve* arbitrary constraints for instantiations of the variables.

In this paper, we shall present a formal framework for functional logic languages using an abstract language based on λ-calculus. This is a natural sequel to [Red85]. Many aspects of functional logic languages that were left tacit in [Red85] are made explicit and formal. In addition, adequate attention is paid to higher order functions and set abstraction.

2 Framework

We first define a small abstract language based on λ-calculus. Its main difference from λ-calculus in that we use terms as formal parameters in abstractions. Syntactically, the language is very similar to HOPE-like functional languages. But, its operational semantics uses narrowing.

2.1 Syntax

Syntactic domains:

Variable ranged over by x and y
Atom ranged over by a and b
Term ranged over by t
Expression ranged over by e, d and f.

To distinguish atoms from variables, we use names beginning with capital letters for atoms. A distinguished atom *true* is assumed to be present in *Atom*. The abstract syntax of the language is

$$\begin{array}{lll} t & ::= & x \mid a \\ & & (t_1, t_2) \quad \text{- pairs} \end{array}$$

$$
\begin{array}{lll}
e ::= & x \mid a & \\
& \mid (e_1, e_2) & \text{– pair} \\
& \mid (\lambda t.\, e) & \text{– abstraction} \\
& \quad \text{where } t \text{ is linear} & \\
& \mid (e_1; e_2) & \text{– union or lub} \\
& \mid (e_1\ e_2) & \text{– application} \\
& \mid (e_1 \rightarrow e_2) & \text{– conditional} \\
& \mid (e_1 = e_2) & \text{– equality} \\
& \mid (e_1 \wedge e_2) & \text{– conjunction}
\end{array}
$$

The restriction that t should be *linear* in abstraction means that no variable should appear twice in t. This restriction is necessary to ensure that the functions denoted by abstraction expressions are continuous. To avoid the proliferation of parentheses, we use the following precedence rule:

$$\lambda \leq ; \leq \rightarrow \leq \wedge \leq =$$

In addition, we assume that ",", λ, $\rightarrow$ and $\wedge$ associate to the right, and ";" and application associate to the left.

As an example of the syntax, the *append* function of section 1 is written in the abstract language as follows:

$$
\begin{array}{l}
(fix\ (\lambda append.\ (\lambda Nil.\ \lambda y.\ y); \\
\qquad\qquad\qquad (\lambda (a, x).\ \lambda y.\ (a,\ (append\ x\ y)))))
\end{array}
$$

The functions defined by the various equations are combined using the ";" operator. Conventionally, equational definitions such as that of *append* are translated into the standard λ calculus using conditional, e.g.

$$
\begin{array}{l}
(fix\ (\lambda append.\ \lambda l.\ \lambda y. \\
\quad (l = Nil \rightarrow y); \\
\quad ((pair?\ l) \rightarrow \textbf{let}\ a \equiv (fst\ l),\ x \equiv (snd\ l)\ \ \textbf{in}\ (a,\ (append\ x\ y))) \\
))
\end{array}
$$

Using terms directly as parameters in abstraction expressions provides a convenient notation, and makes it narrowing to define narrowing as we will see later.

2.2 Denotational semantics

The denotational semantics uses complete lattices rather than cpo's. There is a top element $\top$ to denote inconsistent values that may be produced by the ";" operator. Let B be the *set of basic values*.

$$B = A + B \times B$$

Let $B_{\perp\top}$ be the flat lattice obtained by augmenting B with $\perp$ and $\top$. The semantic domain D is the complete lattice defined by

$$D = B_{\perp\top} + [D \rightarrow D]$$

$$
\begin{array}{lcl}
[\![-]\!] & : & Env \rightarrow D \\
\\
[\![x]\!]\eta & = & \eta x \\
[\![a]\!]\eta & = & a
\end{array}
$$

$$
\begin{aligned}
[\![(e_1, e_2)]\!]\eta &= \begin{cases} ([\![e_1]\!]\eta,\ [\![e_2]\!]\eta) & \text{if } [\![e_1]\!]\eta \in B, [\![e_2]\!]\eta \in B \\ \top & \text{if } [\![e_1]\!]\eta = \top \text{ or } [\![e_2]\!]\eta = \top \\ \bot & \text{otherwise} \end{cases} \\
[\![(\lambda t.\ e)]\!]\eta &= \lambda v.\ \text{if } \exists \rho \in FV(t) \to D \text{ such that } [\![t]\!]\rho = v \text{ then } [\![e]\!](\eta;\rho) \text{ else } \bot \\
[\![(e_1; e_2)]\!]\eta &= [\![e_1]\!]\eta \sqcup [\![e_2]\!]\eta \\
[\![e_1 \to e_2]\!]\eta &= \begin{cases} [\![e_2]\!]\eta & \text{if } [\![e_1]\!]\eta = \mathit{true} \\ \top & \text{if } [\![e_1]\!]\eta = \top \\ \bot & \text{otherwise} \end{cases} \\
[\![e_1 = e_2]\!]\eta &= \begin{cases} \mathit{true} & \text{if } [\![e_1]\!]\eta = [\![e_2]\!]\eta \in B \\ \top & \text{if } [\![e_1]\!]\eta = \top \text{ or } [\![e_2]\!]\eta = \top \\ \bot & \text{otherwise} \end{cases} \\
[\![e_1 \wedge e_2]\!]\eta &= \begin{cases} \mathit{true} & \text{if } [\![e_1]\!]\eta = \mathit{true} \text{ and } [\![e_2]\!]\eta = \mathit{true} \\ \top & \text{if } [\![e_1]\!]\eta = \top \text{ or } [\![e_2]\!]\eta = \top \\ \bot & \text{otherwise} \end{cases}
\end{aligned}
$$

The environment $(\eta;\rho)$ in the semantics of abstraction expression means the environment obtained by updating η with the bindings in ρ. $FV(t)$ is the set of free variables of t.

Note that the pairs constructed by "," are *strict*. It will be seen later that strict data structures are necessary to make good use of the capabilities offered by narrowing. Nonstrict data structures are still available through the use of functions, just as in conventional λ calculus.

$$
\begin{aligned}
\mathit{pair} &= \lambda x.\ \lambda y.\ ((\lambda 1.\ x);\ (\lambda 2.\ y)) \\
\mathit{fst} &= \lambda p.\ (p\ 1) \\
\mathit{snd} &= \lambda p.\ (p\ 2)
\end{aligned}
$$

2.3 Reduction semantics

For comparison, we first present the conventional reduction semantics for the language.

$$
\begin{array}{lrcl}
\alpha: & (\lambda t.\ e) & \Longrightarrow & \lambda t'.\theta[e] \\
 & & & \text{where } t' = \theta[t] \text{ is a renaming of } t \\
\text{APP:} & ((\lambda t.\ d)\ e) & \Longrightarrow & \theta[d] \\
 & & & \text{if there are no common free variables in } t \text{ and } e \\
 & & & \text{and } \theta \text{ is the minimal substitution such that } \theta[t] = e \\
\text{CHOICE1:} & (e_1; e_2) & \Longrightarrow & e_1 \\
\text{CHOICE2:} & (e_1; e_2) & \Longrightarrow & e_2 \\
\text{COND:} & (\mathit{true} \to e) & \Longrightarrow & e \\
\text{EQ1:} & (a = a) & \Longrightarrow & \mathit{true} \\
\text{EQ2:} & ((e_1, e_2) = (e_3, e_4)) & \Longrightarrow & (e_1 = e_3) \wedge (e_2 = e_4) \\
\text{CONJ:} & (\mathit{true} \wedge \mathit{true}) & \Longrightarrow & \mathit{true}
\end{array}
$$

The complex rule for function application can be seen to be just a short hand for the following simpler rules:

$$
\begin{array}{lrcl}
\text{APP1:} & ((\lambda x.\ d)\ e) & \Longrightarrow & [e/x][d] \\
\text{APP2:} & ((\lambda a.\ d)\ a) & \Longrightarrow & d \\
\text{APP3:} & ((\lambda (t_1, t_2).\ d)\ (e_1, e_2)) & \Longrightarrow & ((\lambda t_1.\ \lambda t_2.\ d)\ e_1\ e_2)
\end{array}
$$

Evaluation using these rules does not necessarily preserve the semantics of expressions because of the nondeterminism involved in the CHOICE rules. However, if $(e \Longrightarrow e')$, then $[\![e]\!] \sqsupseteq [\![e']\!]$. We call this *soundness* of the reduction relation. Furthermore, for any expression e, there is a set of normalizing reductions $\{e \Longrightarrow e_i\}_i$ such that $[\![e]\!] = \bigsqcup_i [\![e_i]\!]$. We call this *completeness* of the reduction relation.

2.4 Narrowing semantics

We can now extend the operational semantics of the language to narrowing. A syntactic (meta-level) substitution is written as

$$[t_1/x_1, \ldots, t_n/x_n]$$

The variables $x_1, \ldots, x_n$ are called the *domain variables (DV)* of the substitution. For narrowing, we need to represent substitutions as parts of expressions. We use the following subclass of expressions for this purpose.

$$\begin{array}{lll} \sigma & \in & \textit{Substitution-expression} \\ \sigma & ::= & (x = t) \quad \text{where } x \text{ is not free in } t \\ & & \mid (\sigma_1 \wedge \sigma_2) \quad \text{where } DV(\sigma_1) \cap FV(\sigma_2) = \emptyset, \;\; DV(\sigma_2) \cap FV(\sigma_1) = \emptyset \end{array}$$

A syntactic substitution $[t/x]$ is represented by the equality $(x = t)$, and compound substitutions with several bindings are represented by conjunctions. Since the correspondence is straightforward, we shall not distinguish between substitutions and their expression representations.

First of all, the rules for application are extended to handle variable arguments:

$$\begin{array}{lrcl} \text{APP2}': & ((\lambda a.\, d)\; x) & \Longrightarrow & x = a \rightarrow [a/x][d] \\ \text{APP3}': & ((\lambda(t_1,t_2)\; d)\; x) & \Longrightarrow & x = (x_1,x_2) \rightarrow ((\lambda t_1.\, \lambda t_2.\, [(x_1,x_2)/x][d])\; x_1\; x_2)) \\ & & & \text{where } x_1 \text{ and } x_2 \text{ are new variables, and} \\ & & & x \text{ is not free in } t_1 \text{ or } t_2 \end{array}$$

All these rules for application can be combined into a single complex rule that uses unification:

$$\begin{array}{lrcl} \text{APP}: & ((\lambda t.\, d)\; e) & \Longrightarrow & (\sigma \rightarrow \theta[\sigma[d]]) \\ & & & \text{where } t \text{ and } e \text{ have no common free variables} \\ & & & \text{and } \sigma \wedge \theta \text{ is a most general unifier such that } \sigma[e] = \theta[t]. \end{array}$$

The other forms of expressions such as conditional are disguised function applications. So, they too need to be extended for solving for variables.

$$\begin{array}{lrcl} \text{COND}': & (x \rightarrow e) & \Longrightarrow & (x = \mathit{true} \rightarrow e) \\ \text{EQ1}': & (x = a) & \Longrightarrow & (x = a) \rightarrow \mathit{true} \\ \text{EQ2}': & (x = (t_1,t_2)) & \Longrightarrow & (x = (x_1,x_2) \rightarrow (x_1 = t_1 \wedge x_2 = t_2)) \\ & & & \text{where } x_1 \text{ and } x_2 \text{ are new variables, and} \\ & & & x \text{ is not free in } (t_1,t_2) \\ \text{EQ3}': & (t = x) & \Longrightarrow & (x = t) \\ \text{CONJ1}': & (x \wedge e) & \Longrightarrow & (x = \mathit{true} \rightarrow [\mathit{true}/x][e]) \\ \text{CONJ2}': & (e \wedge x) & \Longrightarrow & (x = \mathit{true} \rightarrow [\mathit{true}/x][e]) \end{array}$$

By using narrowing, we now have a new kind of irreducible normal forms: conditional terms of the form $(\sigma \rightarrow t)$. The substitutions σ in such conditional terms (and conditional expressions) need to be propagated out from all possible reduction positions. This is handled by the following rules:

$$\begin{array}{lrcl} \text{PROP-PAIR1}: & ((\sigma \rightarrow e_1), e_2) & \Longrightarrow & (\sigma \rightarrow (e_1, \sigma[e_2])) \\ \text{PROP-PAIR2}: & (e_1, (\sigma \rightarrow e_2)) & \Longrightarrow & (\sigma \rightarrow (\sigma[e_1], e_2)) \\ \text{PROP-APP1}: & ((\sigma \rightarrow e_1)\; e_2) & \Longrightarrow & (\sigma \rightarrow (e_1\; \sigma[e_2])) \\ \text{PROP-APP2}: & ((\lambda t.\, e_1); (\sigma \rightarrow e_2)) & \Longrightarrow & (\sigma \rightarrow (\sigma[\lambda t.\, e_1]\; e_2)) \\ & & & \text{if } t \text{ is not a variable} \\ \text{PROP-COND}: & ((\sigma \rightarrow e_1) \rightarrow e_2) & \Longrightarrow & (\sigma \rightarrow (e_1 \rightarrow \sigma[e_2])) \\ \text{PROP-EQ1}: & ((\sigma \rightarrow e_1) = e_2) & \Longrightarrow & (\sigma \rightarrow (e_1 = \sigma[e_2])) \\ \text{PROP-EQ2}: & (e_1 = (\sigma \rightarrow e_2)) & \Longrightarrow & (\sigma \rightarrow (\sigma[e_1] = e_2)) \\ \text{PROP-CONJ1}: & ((\sigma \rightarrow e_1) \wedge e_2) & \Longrightarrow & (\sigma \rightarrow (e_1 \wedge \sigma[e_2])) \\ \text{PROP-CONJ2}: & (e_1 \wedge (\sigma \rightarrow e_2)) & \Longrightarrow & (\sigma \rightarrow (\sigma[e_1] \wedge e_2)) \end{array}$$

Note that there is only one propagation rule for conditionals. Because we reduce the condition of a conditional before reducing its body, we do not need to propagate a substitution out from the body. But, there is a special case. If the condition of the conditional is itself a substitution, then we need to take the composition of the two substitutions.

$$\text{COMP:}\ \ (\sigma_1 \rightarrow (\sigma_2 \rightarrow e)) \ \ \Longrightarrow\ \ (\sigma_1 \circ \sigma_2 \rightarrow e)$$

The propagation of substitutions shows why we need strict pairs. Consider, for example,

$$(x = A \rightarrow 0,\ x)$$

Its first component is defined only if $x = A$ in the current environment, but its second component is whatever is the binding of x in the environment. So, in an environment in which x is bound to 0, it would denote $(\perp, 0)$. But, typically, we would like the substitutions that come up in the evaluation of one part of a data structure to be used for instantiating its other parts. In other words, we would be interested in those environments in which all the components of the data structure are defined. The propagation rules PROP-PAIR1 and PROP-PAIR2 which achieve this are valid only for strict pairs. Through these rules, the above expression reduces to

$$x = A \rightarrow (0, A)$$

which simply denotes $\perp$ in an environment in which x is bound to 0.

Similar propagation does not happen for nonstrict pairs constructed using the *pair* function. The expression

$$(pair\ (\sigma \rightarrow e_1)\ e_2)$$

does not reduce to

$$\sigma \rightarrow (pair\ e_1\ e_2)$$

because the rule PROP-APP2 does not apply when the formal parameter of the function is a variable.

The narrowing relation given by these rules is sound. It is also complete except for a minor technical problem, *viz.*, that narrowing can introduce new variables which are not present in the original expression. Consider, for example,

$$[\![((\lambda(a,x).\ a)\ z)]\!]\eta = \begin{cases} v_1 & \text{if } \eta z = (v_1, v_2) \\ \perp & \text{otherwise} \end{cases}$$

But, narrowing evaluates the expression to

$$[\![z = (a,x) \rightarrow a]\!]\eta = \begin{cases} v_1 & \text{if } \eta z = (v_1, v_2),\ \eta a = v_1,\ \eta x = v_2 \\ \perp & \text{otherwise} \end{cases}$$

While the original expression is defined in environments in which only z is bound, the result expression is defined in environments in which a and x are also bound appropriately. In the next section, we shall introduce a construct for existential quantification using which this problem can be avoided. But, for now, observe that narrowing is complete provided we allow for the environments to be extended for new variables in the semantics of the result expressions.

In addition to soundness and completeness, narrowing also satisfies the property of *solution-completeness*. Let *basic environments* be

$$Benv = Variable \rightarrow B_{\perp\top}$$

Then the *basic parametric value* of a nonground expression is

$$\mathcal{B}[\![e]\!]\eta = \begin{cases} [\![e]\!]\eta & \text{if } \eta \in \mathit{Benv} \\ \bot & \text{otherwise} \end{cases}$$

For every nonground expression e, there is a set of terminal narrowings $\{e \Longrightarrow \sigma_i \rightarrow t_i\}_i$ where t_i is a term or an abstraction, such that

$$\mathcal{B}[\![e]\!] = \bigsqcup_i [\![\sigma_i \rightarrow t_i]\!].$$

We call this *basic solution-completeness* of narrowing. It means that narrowing can solve for basic-valued instantiations of variables in nonground expressions. It cannot solve for function-valued instantiations because the APP rules do not allow variables as functions.

3 Existential quantification

Now that we have a basic narrowing functional language, we can extend it in a number of ways to add more expressive power. Note that the programs that can be written in the language so far are independent of whether reduction or narrowing is used as the operational semantics. Though narrowing gives us the capability to evaluate nonground expressions with free variables, this capability is exhibited only in the *use* of the programs, not in the programs themselves. So, the first extension we would like is to make this capability available in writing programs.

To introduce new "free" variables in expressions, we use a new binding construct in the syntax of expressions, *viz.*, existential quantification.

$$\begin{array}{rcl} e & ::= & (\exists x.\, e) \\ [\![(\exists x.\, e)]\!]\eta & = & \bigsqcup_{v \in Dcon} [\![e]\!](\eta; [v/x]) \end{array}$$

where *Dcon* is the set $D - \{\top\}$. The construct $\exists x. e$ may be read as "for some value of x, the value of e". Intuitively, it means that every atomic piece information obtainable from $\exists x. e$ is obtainable from e with some instantiation of x from *Dcon*. If e has no free occurrences of x then $(\exists x.\, e)$ is trivially equivalent to e. If e has free occurrences of x, then it may take different values for different instantiations of x. If all these values are consistent, then the information obtainable from $(\exists x. e)$ is the sum total of the information obtainable from the individual values. Otherwise, the expression denotes $\top$, and, operationally, it may evaluate to one or more of these values. Usually, we use the existential quantification in such a way that e denotes a non-$\bot$ value for only one instantiation of x.

In logic programming parlance, variables introduced using the existential quantification are called *logical variables.* They provide a powerful programming paradigm discussed later in the section. Recall the semantic problems caused by the fact that the basic narrowing mechanism of section 2.4 introduces new free variables that are not present in the original expression. Using existential quantification, such new variables can be quantified so that they are not left free in the final expression. So, we modify the rules APP3′, EQ2′, and all the substitution propagation rules, including COMP, of section 2.4 as follows:

$$\begin{array}{lrcl}
\text{APP3}': & ((\lambda(t_1,t_2)\, d)\; x) & \Longrightarrow & \exists x_1.\, \exists x_2.\, x = (x_1, x_2) \rightarrow ((\lambda t_1.\, \lambda t_2.\, d)\; x_1\; x_2)) \\
 & & & \text{where } x_1 \text{ and } x_2 \text{ are new variables} \\
\text{EQ2}': & (x = (t_1, t_2)) & \Longrightarrow & \exists x_1. \exists x_2.\, x = (x_1, x_2) \rightarrow (x_1 = t_1 \wedge x_2 = t_2) \\
 & & & \text{where } x_1 \text{ and } x_2 \text{ are new variables} \\
\text{PROP-PAIR1}: & ((\exists \overline{x}.\, \sigma \rightarrow e_1), e_2) & \Longrightarrow & (\exists \overline{x}.\, \sigma \rightarrow (e_1, \sigma[e_2])) \\
 & & & \text{if the variables } \overline{x} \text{ are not free in } e_2, \\
 & & \vdots & \\
\text{COMP}: & (\sigma_1 \rightarrow (\exists \overline{x}.\, \sigma_2 \rightarrow e)) & \Longrightarrow & (\exists \overline{x}.\, \sigma_1 \circ \sigma_2 \rightarrow e) \\
 & & & \text{if the variables } \overline{x} \text{ are not free in } \sigma_1
\end{array}$$

The notation $\exists\overline{x}.$ stands for a series of *zero or more* variable bindings $\exists x_1 .\dots \exists x_n$. Note that, in general, the terminal forms for successful narrowings are now of the form

$$\exists\overline{x}.\, \sigma \rightarrow t$$

and the propagation rules propagate the quantifiers as well as the substitutions outwards.

For the example mentioned in section 2.4, the new rules now produce

$$((\lambda(a, x).\, a)\ z) \Longrightarrow \exists a.\, \exists x.\, z = (a, x) \rightarrow a$$

The semantics of the result expression is

$$\begin{aligned}
&[\![\exists a.\, \exists x.\, z = (a, x) \rightarrow a]\!]\eta \\
&= \bigsqcup_{v_1, v_2 \in Dcon} [\![z = (a, x) \rightarrow a]\!](\eta; [v_1/a, v_2/b]) \\
&= \bigsqcup_{v_1 . v_2 \in Dcon} \left\{ \begin{array}{ll} v_1 & \text{if } \eta z = (v_1, v_2) \\ \bot & \text{otherwise} \end{array} \right\} \\
&= \left\{ \begin{array}{ll} v_1 & \text{if } \eta z = (v_1, v_2) \\ \bot & \text{otherwise} \end{array} \right.
\end{aligned}$$

which is the same as the semantics of the original expression.

The $\exists$ quantification provides more justification for strict data structures. Consider

$$\exists x.\, (x = A \rightarrow 0,\ x)$$

If pairs were not strict, its denotation would be

$$\begin{aligned}
&\lambda\eta.\, \bigsqcup_{v \in Dcon} (\left\{ \begin{array}{ll} 0 & \text{if } v = A \\ \bot & \text{otherwise} \end{array} \right\}, v) \\
&= \lambda\eta.\, (0, \top)
\end{aligned}$$

Even though there is a unique consistent value for the second component whenever the first component is defined, the second component turns out to be $\top$ since no information can be exchanged between the two components. But, if pairs are strict, its denotation is

$$\begin{aligned}
&\lambda\eta.\, \bigsqcup_{v \in Dcon} \left\{ \begin{array}{ll} (0, A) & \text{if } v = A \\ \bot & \text{otherwise} \end{array} \right\} \\
&= \lambda\eta.\, (0, A)
\end{aligned}$$

Thus, while nonstrict data structures allow us to avoid $\bot$, strict data structures allow us to to avoid $\top$ when $\exists$ quantification is used.

The evaluation rules for the existential expressions are

$$\begin{array}{lrcl}
\alpha\text{-EXIST:} & \exists x.\, t & \Longrightarrow & \exists x'.\, [x'/x][t] \\
 & & & \text{if } x' \text{ is not free in } t \\
\text{EXIST1:} & (\exists x.\, t) & \Longrightarrow & t \\
\text{EXIST2:} & (\exists x.\, \exists\overline{y}.\, x = t \rightarrow e) & \Longrightarrow & (\exists\overline{y}.\, e) \\
 & & & \text{if } x \text{ is not free in } t \text{ or } e \\
\text{EXIST3:} & (\exists x.\, \exists\overline{y}.\, x = t \wedge \sigma \rightarrow e) & \Longrightarrow & (\exists\overline{y}.\, \sigma \rightarrow e) \\
 & & & \text{if } x \text{ is not free in } t, \sigma \text{ or } e \\
\text{PROP-EXIST:} & (\exists x.\, \exists\overline{y}.\, \sigma \rightarrow e) & \Longrightarrow & (\exists\overline{y}.\, \sigma \rightarrow (\exists x.\, e)) \\
 & & & \text{if } x \text{ is not free in } \sigma
\end{array}$$

We now give a number of examples illustrating the paradigmatic use of logical variables.

Example 1 (Member function) We can define a predicate for testing membership in a list by

$member(a, b.x) \equiv (a = b \rightarrow true;\ member(a, x))$

Given ground arguments, it works the same way as a *member* function in a conventional functional language. For example,

$member(A, A.B.Nil)$
$\Longrightarrow A = A \rightarrow true;\ member(A, B.Nil)$
$\Longrightarrow true$

using the rule CHOICE1 to resolve the choice. If we use CHOICE2, we get

$member(A, B.nil)$
$\Longrightarrow A = B \rightarrow true;\ member(A, Nil)$

and both the choices fail (denote $\perp$).

We can also invoke *member* with nonground arguments, just as we did with *append* in section 1. If we use a variable for the first argument as, for example,

$member(e, A.B.Nil)$

then we should expect to get all instantiations of e for which $member(e, A.B.Nil)$ holds. This is indeed what is obtained by evaluating this expression. It evaluates nondeterministically to two terminal forms:

$member(e, A.B.Nil)$	
$\Longrightarrow e = A \rightarrow true;\ member(e, B.Nil)$	APP
$\Longrightarrow e = A \rightarrow true$ (1)	CHOICE1
$\Longrightarrow member(e, B.Nil)$	CHOICE2
$\Longrightarrow e = B \rightarrow true;\ member(e, Nil)$	APP
$\Longrightarrow e = B \rightarrow true$ (2)	CHOICE1
$\Longrightarrow member(e, Nil)$	CHOICE2
fails	

The use of a variable as the second argument to *member* is much more interesting, but it is also much more complex. The evaluation of

$member(A, l)$

should produce as an instantiation of l, a list that contains A. Since there are an infinite number of such lists, we get an infinite number of narrowings:

$member(A, l)$	
$\Longrightarrow \exists b.\ \exists x.\ l = b.x$	APP
$\rightarrow (A = b \rightarrow true;\ member(A, x))$	
$\Longrightarrow \exists b.\ \exists x. l = b.x \rightarrow b = A \rightarrow true$	CHOICE1, EQ3$'$
$\Longrightarrow \exists b.\ \exists x. l = A.x \wedge b = A \rightarrow true$	COMP
$\Longrightarrow \exists x. l = A.x \rightarrow true$	EXIST3

Using CHOICE2 again produces an infinite number of further narrowings:

$\exists b.\ \exists x'. l = b.A.x' \rightarrow true$
$\exists b.\ \exists b'.\ \exists x''. l = b.b'.A.x'' \rightarrow true$
$\exists b.\ \exists b'.\ \exists b''.\ \exists x'''. l = b.b'.b''.A.x''' \rightarrow true$

Expressions with such infinite solutions are problematic since in most applications only one solution would do. In Prolog, the infinite solutions are inhibited using the nonlogical "cut" primitive. But, since we have no such primitive, they need to be inhibited through careful use of the $\exists$ quantification.

We can assert any number of values to be members of l. For example

$$member(A,l) \wedge member(B,l) \wedge member(C,l) \Longrightarrow \exists x.\, l = A.B.C.x \rightarrow true$$

However, there is a problem. If we evaluate

$$member(A,l) \wedge member(A,l)$$

we get not only the straightforward first narrowing

$$\exists x.\, l = A.x \rightarrow true$$

but also the one in which l is bound to a list with two occurrences of A.

$$\exists x.\, l = A.A.x \rightarrow true$$

Avoiding such solutions requires the use of negation, discussed in section 4.3

Example 2 (Retrieving attributes from association lists) Let an association list be a list of pairs, whose first components are keys and second components are attributes, so that no key appears more then once in the association list. We can define a function to retrieve the attribute of a key as

$$retrieve(key, alist) \equiv \exists attr.\ member((key, attr), alist) \rightarrow attr$$

Operationally, its evaluation is done by first evaluating the body of the existential quantifier with *attr* as a free variable. When it narrows to a conditional term of the form

$$attr = A \rightarrow A$$

the condition is dropped by the rule EXIST2 and A is produced as the value of the quantified expression. Since, by assumption, *alist* contains at most one pair for any key, the body of the quantifier has at most one successful narrowing. Hence, the definition of *retrieve* is consistent.

Like *member*, *retrieve* can also be invoked with a variable for either the first or the second argument. If a variable is used as the second argument, it gets instantiated to an association list that contains a pair for the given key. For example,

$$retrieve(A,l) \Longrightarrow \exists a.\, \exists x.\, l = (A,a).x \rightarrow a$$

But, if we try to retrieve the same key twice, it is possible to get different attributes. For example,

$$(retrieve(A,l),\ retrieve(A,l) \Longrightarrow \exists a.\, \exists a'.\, \exists x.\, l = (A,a).(A,a').x \rightarrow (a,a)$$

Even though we have assumed that an association list does not have multiple pairs for any value of the key, the definition of *retrieve* does not explicitly use this assumption. We can strengthen *retrieve* so that this assumption is built into it by using negation. We discuss this in section 4.3.

Note that, in a conventional functional language without logical variables, we would have to give an independent recursive definition for *retrieve* which compares only the keys in the association list with the given key. Logical variables allow us to reuse code in ways not possible in conventional functional languages.

Example 3 (Address translation) We want to define a function that translates a sequence of definitions and uses of symbolic addresses to one of numeric addresses. Suppose the input is a list containing two kinds of terms: (Def, s) and (Use, s), where s is a symbol. For each symbol s that appears in a (Use, s) term, there is a unique term (Def, s) in the list. The translation should translate each (Def, s) to a term $(Assign, s, n)$ and each (Use, s) to a term (Use, n), such that

1. if (Def, s) precedes (Def, s') in the input sequence, then their translations $(Assign, s, n)$ and $(Assign, s', n')$ satisfy $n \leq n'$,
2. if (Def, s) is translated to $(Assign, s, n)$ then every (Use, s) is translated to (Use, n), and
3. the numbers assigned to symbols are successive integers starting from 1.

What is interesting about the problem is that the use of a symbol can appear in the input list before its definition itself appears. So, two passes over the list are required in a conventional functional language: one to construct a table mapping symbols to numeric addresses, and a second to do the translation.

In the following program, we use a logical variable as a table. It gets constructed (bound) on the fly, as the translation is done.

```
translate seq ≡
    letrec tran(Nil, table, n) ≡ Nil
           tran((Def, s).seq, table, n) ≡  retrieve(s, table) = n
                                            → (Assign, s, n).tran(seq, table, n + 1)
           tran((Use, s).seq, table, n) ≡ (Use, retrieve(s, table)).tran(seq, table, n)
    in ∃table. tran(seq, table, 1)
```

This program is based on the fact that when *retrieve* is invoked with a variable as the second argument, it instantiates it to an association list containing a pair for the given key. Let us look at an example evaluation

$$\begin{array}{l} translate\ (Use, A).(Def, A).Nil \\ \Longrightarrow \exists table.\ tran((Use, A).(Def, A).Nil), table, 1) \\ \Longrightarrow \exists table.\ \begin{array}[t]{l}(Use, retrieve(A, table))\ . \\ (retrieve(A, table) = 1 \rightarrow (Assign, A, 1))\ . \\ Nil\end{array} \end{array}$$

The subexpression $retrieve(A, table)$ has an infinite number of nondeterministic narrowings:

$$\begin{array}{l} \exists a.\ \exists t.\ table = (A, a).t \rightarrow a \\ \exists k.\ \exists a.\ \exists a'.\ \exists t'.\ table = (k, a).(A, a').t' \rightarrow a \\ \vdots \end{array}$$

Suppose we use the first narrowing. Then the original expression narrows to

$$\exists table.\ \exists a.\ \exists t.\ table = (A, a).t \rightarrow \begin{array}[t]{l}(Use, a)\ . \\ (retrieve(A, (A, a).t) = 1 \rightarrow (Assign, A, 1))\ . \\ Nil\end{array}$$

Again choosing the first narrowing for the remaining *retrieve* application, we obtain

$$\begin{array}{ll} \exists table.\ \exists a.\ \exists t.\ table = (A, a).t \rightarrow \begin{array}[t]{l}(Use, a)\ . \\ (a = 1 \rightarrow (Assign, A, 1))\ . \\ Nil\end{array} & \\ \Longrightarrow \exists table.\ \exists a.\ \exists t.\ table = (A, 1).t \wedge a = 1 \rightarrow (Use, 1).(Assign, A, 1).Nil & \\ \Longrightarrow \exists t.\ (Use, 1).(Assign, A, 1).Nil & \text{by EXIST2, EXIST3} \\ \Longrightarrow (Use, 1).(Assign, A, 1).Nil & \text{by EXIST1} \end{array}$$

Thus, even though the attribute of A is not known at the time (Use, A) is translated, it is assumed to be a variable a and is bound whenever (Def, A) is translated.

But, there are other concerns about this program. Since each *retrieve* application has infinite narrowings, the body of the quantifier $tran(seq, table, 1)$ has infinite narrowings too. Is the definition of *translate* still consistent? Secondly, do we get infinite narrowings operationally or just one? To answer the first question, recall that each symbol has a (Def, s) term in the input sequence. So, every variable temporarily assumed as the attribute of a symbol eventually gets bound when the (Def, s) term is processed. So, the final term is variable-free. The various narrowings can bind table to contain other irrelevant pairs, e.g.,

$$\exists table.\, \exists k.\, \exists a.\, \exists t'.\, table = (k, a).(A, 1).t \rightarrow (Use, 1).(Assign, A, 1).Nil$$

But, the final result is the same, since the result term is variable-free. If the two retrieves produce different attributes for A as discussed in example 2, then we can get a result term with variables, such as

$$\exists table.\, \exists a'.\, \exists t'.\, table = (A, 1).(A, a').t \rightarrow (Use, a').(Assign, A, 1).Nil$$

But, we shall give a stronger definition of *retrieve* in section 4.3 by which such narrowings are eliminated. So, the definition of *translate* is consistent. But, operationally, we do not yet have a framework by which the different nondeterministic branches can be collected together. So, we get an infinite number of narrowings all of which produce the same result. We will present a solution to this problem in section 4.2.

This example also illustrates our need for strict pairs. If pairs were not strict then the original *translate* application would be equivalent to

$$\begin{array}{l} (Use,\ \exists table.\, retrieve(A, table))\ . \\ (\exists table.\, retrieve(A, table) = 1 \rightarrow (Assign, A, 1))\ . \\ Nil \end{array}$$

Different components of the output sequence can use different instantiations of *table*, and it becomes impossible to enforce that we want a single *table* to be used for the entire translation.

Example 4 (Permutation Inversion) We are given a vector of n elements each of which is an integer between 1 and n, so that no integer appears more than once. So, the vector represents a permutation of the integers $1 \ldots n$. We want to define a function that constructs a vector representing the inverse of the permutation.

Let us assume a primitive predicate $vector(v, n)$ which means "v is a vector of n elements". If v is a variable, it binds it to a vector of n existentially quantified variables. If n is a variable, it binds it to the size of v. If both of them are variables, then the result is undefined.

Now, the function *inverse* can be defined as

$$\begin{array}{lll} inverse\ p & \equiv & \exists n.\, \exists v.\ (vector(p, n) \wedge \\ & & \quad vector(v, n) \wedge \\ & & \quad forall(ints\text{-}between(1, n),\ \lambda i.\, v(p(i)) = i)) \\ & & \quad \rightarrow v \end{array}$$

$ints\text{-}between(m, n)$ generates a list of integers between m and n. $forall(l, p)$ checks the predicate p for all elements of the list l.

The body of the quantifier is evaluated with the logical variables n and v. $vector(p, n)$ binds n to the size of p. $vector(v, n)$ binds v to a vector of n variables. The *forall* expression binds every $p(i)$'th element of v to i. At the end of this, there is a unique ground binding of v, since each $j \in 1 \ldots n$ appears exactly once in p.

Note that this it is quite hard to solve this problem efficiently in a conventional functional language without logical variables. The result vector v would have to be defined by a function that maps an index j to the corresponding element of v.

$$\begin{aligned} \textit{inverse } p \;\equiv\; & \textit{make-vector}(\; \textit{size } p, \\ & \qquad \lambda j.\, \textit{such-that}(\textit{ints-between}(1,n),\ \lambda i.\, p(i) = j)) \end{aligned}$$

$\textit{make-vector}(n, f)$ generates a vector of size n, whose j'th element is $(f\ j)$. $\textit{such-that}(l, p)$ yields the first element of the list l satisfying the predicate p. Since the element $v(j)$ can only be known by searching for j in the input vector p, this solution takes $O(n^2)$ time. In contrast, the solution with logical variables takes only $O(n)$ time.

In conventional functional languages, a variable is always introduced together with its binding. Therefore, the variables do not play any useful role and can be eliminated. This is indeed demonstrated by the combinatory calculus. In a functional logic language, on the other hand, a variable can be introduced without a binding, and can be bound whenever information becomes available. For example, in the permutation inversion program, we introduce a vector of variables and then bind them independently in an arbitrary order. It is this decoupling of the introduction and the binding of variables, that allows us to solve the problem in $O(n)$ time. Similarly, in example 2, we introduce a variable for the attribute of a key, and bind the variable using *member*. This flexibility is reminiscent of imperative languages where we can allocate a location and use a pointer to it, before the contents of the location itself is defined. This is indeed how logical variables are implemented. But, they have several advantages over imperative variables. Firstly, the pointers are transparent. Secondly, variables are bound only once; they are not mutable. Therefore, the same operator can be used both for testing equality and for binding. When performing a unification, we do not need to know whether a variable is already bound or not.

However, there is a qualitative difference between the binding of logical variables and the binding done in function application. Consider the following two expressions that appear similar:

$$\textbf{let } x \equiv e_1 \textbf{ in } e_2$$

$$\exists x.\, x = e_1 \rightarrow e_2$$

The first is actually syntactic sugar for

$$((\lambda x.\, e_2)\ e_1)$$

The binding of x in its evaluation is *strong* binding. It does not require the evaluation of e_1. But, the binding of x in the evaluation of the second expression is *weak* binding. It is done only after e_1 is evaluated to a term. If, for example, e_1 is an infinite list and $e_2 = (\textit{fst}\ x)$, then the first expression terminates under outermost evaluation, whereas the second one does not terminate even under outermost evaluation. This is because the $=$ construct denotes the application of a continuous equality function which cannot yield *true* if its arguments are infinite objects.

4 Failure and nondeterminism

4.1 Failure

By the evaluation rules we have seen so far, there are many terminal forms which simply denote $\perp$, e.g., $A = B$. They are the cases where our rules fail to reduce them to meaningful values. We can introduce an explicit failure expression which captures them all into a single terminal form. We use a special symbol ω for this purpose.

$$\begin{aligned} e &::= \omega \\ [\![\omega]\!]\eta &= \perp \end{aligned}$$

ω is also treated as a term.

The evaluation rules for failure are of two kinds: failure introduction rules and failure propagation rules. The failure introduction rules are

APP-FAIL1:	$(t\ e)$	$\Longrightarrow$	ω	
APP-FAIL2:	$((\lambda a.\ d)\ t)$	$\Longrightarrow$	ω	if $t \neq a$
APP-FAIL3:	$((\lambda(t_1, t_2).\ d)\ a)$	$\Longrightarrow$	ω	
COND-FAIL:	$(t \rightarrow e)$	$\Longrightarrow$	ω	if $t \neq \mathit{true}$
EQ-FAIL1:	$(a = b)$	$\Longrightarrow$	ω	if $a \neq b$
EQ-FAIL2:	$(a = (e_1, e_2))$	$\Longrightarrow$	ω	
EQ-FAIL3:	$((e_1, e_2) = a)$	$\Longrightarrow$	ω	
EQ-FAIL4:	(e_1, e_2)	$\Longrightarrow$	ω	if e_1 or e_2 is an abstraction
CONJ-FAIL:	$(t_1 \wedge t_2)$	$\Longrightarrow$	ω	if $t_1 \neq \mathit{true}$ or $t_2 \neq \mathit{true}$

When ω appears in a strict position in a bigger expression, then the whole expression reduces to ω. Many of these failure propagation cases are covered by the above rules. For example, $(\omega\ e) \Longrightarrow \omega$. The other cases are the following:

FAIL-APP2:	$((\lambda t.\ d)\ \omega)$	$\Longrightarrow$	ω	if t is not a variable
FAIL-COND2:	$(\sigma \rightarrow \omega)$	$\Longrightarrow$	ω	
FAIL-PAIR1:	(ω, e)	$\Longrightarrow$	ω	
FAIL-PAIR2:	(e, ω)	$\Longrightarrow$	ω	
FAIL-EQ1:	$(\omega = (e_1, e_2))$	$\Longrightarrow$	ω	
FAIL-EQ2:	$((e_1, e_2) = \omega)$	$\Longrightarrow$	ω	
FAIL-EXIST:	$(\exists x.\ \omega)$	$\Longrightarrow$	ω	

4.2 Nondeterminism

The rules for the choice operator ";" given in section 2.4 are nondeterministic. The evaluation can choose CHOICE1 or CHOICE2 along different branches. However, the nondeterminism does not commit to any one choice, but requires all the choices to be tried. This can be made explicit by carrying the choice operator until one of the choices evaluates to ω. The basic choice rules are

CHOICE-ELIM1:	$(\omega; e)$	$\Longrightarrow$	e
CHOICE-ELIM2:	$(e; \omega)$	$\Longrightarrow$	e
CHOICE-COM:	$(e_1; e_2)$	$\Longrightarrow$	$(e_2; e_1)$
CHOICE-ASSOC:	$((e_1; e_2); e_3)$	$\Longrightarrow$	$(e_1; (e_2; e_3))$

We also need to propagate the choice operator from every strict evaluation position. This is done by the following rules:

CHOICE-APP1:	$((e_1; e_2)\ e_3)$	$\Longrightarrow$	$(e_1\ e_3)\ ;\ (e_2\ e_3)$
CHOICE-APP2:	$((\lambda t.\ d)\ (e_1; e_2))$	$\Longrightarrow$	$((\lambda t.\ d)\ e_1)\ ;\ ((\lambda t.\ d)\ e_2)$ if t is not a variable
CHOICE-COND1:	$((e_1; e_2) \rightarrow e_3)$	$\Longrightarrow$	$(e_1 \rightarrow e_3\ ;\ e_2 \rightarrow e_3)$
CHOICE-COND2:	$(\sigma \rightarrow (e_1; e_2))$	$\Longrightarrow$	$(\sigma \rightarrow e_1)\ ;\ (\sigma \rightarrow e_2)$
CHOICE-EQ1:	$((e_1; e_2) = e_3)$	$\Longrightarrow$	$(e_1 = e_3)\ ;\ (e_2 = e_3)$
CHOICE-EQ2:	$(e_1 = (e_2; e_3))$	$\Longrightarrow$	$(e_1 = e_2)\ ;\ (e_1 = e_3)$
CHOICE-CONJ1:	$((e_1; e_2) \wedge e_3)$	$\Longrightarrow$	$(e_1 \wedge e_3)\ ;\ (e_2 \wedge e_3)$
CHOICE-CONJ2:	$(e_1 \wedge (e_2; e_3))$	$\Longrightarrow$	$(e_1 \wedge e_2)\ ;\ (e_1 \wedge e_3)$
CHOICE-PAIR1:	$((e_1; e_2), e_3)$	$\Longrightarrow$	$(e_1, e_3)\ ;\ (e_2, e_3)$
CHOICE-PAIR2:	$(e_1, (e_2; e_3))$	$\Longrightarrow$	$(e_1, e_2)\ ;\ (e_1, e_3)$
CHOICE-EXIST:	$(\exists x. (e_1; e_2)$	$\Longrightarrow$	$(\exists x.\ e_1)\ ;\ (\exists x.\ e_2)$

The rules CHOICE-PAIR1 and CHOICE-PAIR2 propagate the choice operator out of pairs, whereas the rule CHOICE-ELIM3 propagates it into pairs. Our intent is to use CHOICE-ELIM rules to eliminate as many choice operators as possible and then to propagate them to the outermost level. The terminal forms of the evaluation are given by

$$\begin{array}{rcl} \tau & ::= & \omega \mid \alpha\,\{;\alpha\}^* \\ \alpha & ::= & \exists\overline{x}.\,\sigma \rightarrow t \mid t \mid \exists\overline{x}.\,\sigma \rightarrow (\lambda t.e) \mid \lambda t.e \qquad \text{where } \overline{x} \not\in DV(\sigma) \\ t & ::= & a \mid (t_1, t_2) \end{array}$$

In practice, the terminal form of an expression is produced "lazily" as a series of approximations, each of the form

$$\alpha_1; \cdots; \alpha_n; e$$

where e is a continuation yet to be evaluated. The user is given the control to evaluate the continuation or to abort.

Note that, with our new choice rules, the evaluation is semantics-preserving. The only case where the earlier evaluation rules lost information was in CHOICE1 and CHOICE2. Since we have replaced them by semantics-preserving rules which carry the choice operators till the end, we avoid the loss of information. Though noble, this is not pragmatically desirable. Recall that in example 3 we have the situation of obtaining an infinite number of α's all of which are identical. At the cost of losing information only when the input expression is inconsistent, we can introduce a rule by which the infinite answers can be eliminated.

$$\text{CHOICE-ELIM3:}\quad t; e \implies t \qquad \text{if } t \text{ does not contain } \omega$$

The rationale is that since t is completely defined, evaluating the continuation e can at best produce an inconsistency. If the expression is consistent, e would certainly evaluate to t or $\bot$. So, if we do not care about finding inconsistencies we can throw e away.

4.3 Alternative formulations

It may have appeared strange that we had only one truth value in our domain, and falsity was denoted by $\bot$. This makes it, in particular, impossible to use a negation operation because it would be nonmonotonic. Negation is, in fact, one of the big unsolved problems in logic programming. However, we have some preliminary ideas on how to treat it in functional logic languages. Our solution is to make the failure constant ω as a distinct value in the semantic domain.

$$D = \{\omega\}_{\bot\top} + B_{\bot\top} + [D \rightarrow D]$$

The choice operator is interpreted as the semantic operation $\vee$ defined by

$$\begin{array}{lcll} \omega \vee v & = & v & \\ v \vee \omega & = & v & \\ v_1 \vee v_2 & = & v_1 \sqcup v_2 & \text{if } v_1, v_2 \in B_{\bot\top} \\ (v_1 \vee v_2)x & = & v_1 x \vee v_2 x & \text{if } v_1 \text{ and } v_2 \text{ are functions} \end{array}$$

We can now define negation, using ω for falsity.

$$e \ ::= \ (\neg\, e)$$

$$[\![\neg\, e]\!]\eta = \begin{cases} \omega & \text{if } [\![e]\!]\eta = \mathit{true} \\ \mathit{true} & \text{if } [\![e]\!]\eta = \omega \\ \top & \text{if } [\![e]\!]\eta = \top \\ \bot & \text{otherwise} \end{cases}$$

Once we have negation, we can also introduce the classical if-then-else construct by syntactic sugaring:

$$(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3) \stackrel{\text{def}}{=} \textbf{let } p \equiv e_1 \textbf{ in } (p \rightarrow e_2; \neg\, p \rightarrow e_3)$$

Another possibility is to interpret ";" as a "sequential" choice operation, i.e.,

$$\begin{aligned} \omega \vee v &= v \\ v_1 \vee v_2 &= v_1 \qquad \text{if } v_1 \in B_\perp \end{aligned}$$

extended pointwise, as usual, for functions. With sequentiality, there is no inconsistency involved. If the first argument is not ω, then the second argument is ignored. In particular, if the first argument is $\perp$, then the choice expression is also $\perp$. Since most implementations of logic languages implement ";" sequentially for the sake of efficiency, this would provide the appropriate semantics for such implementations.

The complete operational semantics of negation is outside the scope of this paper. But, let us briefly discuss how the problems mentioned in section 3 can be avoided by the use of negation. First of all, note that some negated equalities would be irreducible. So, they need to be treated as substitutions too.

$$\sigma ::= (x = t) \mid (\sigma_1 \wedge \sigma_2) \mid (\neg\, \sigma)$$

Now, we can rewrite the *member* function of example 1 using negation, as

$$member(a, b.x) \equiv (a = b \rightarrow true;\ \neg\, a = b \rightarrow member(a, x))$$

This definition makes explicit the assumption that an element appears only once in the list, by performing the recursive application of member only if $a \neq b$. First of all, this cuts down the computation when the arguments are ground. For example,

$$\begin{aligned} & member(A, A.B.Nil) \\ & \Longrightarrow A = A \rightarrow true;\ \neg\, A = A \rightarrow member(A, B.Nil) \\ & \Longrightarrow true; \omega \\ & \Longrightarrow true \end{aligned}$$

The recursive application of *member* is avoided, whereas in example 1 the search continues till the end of the list even though it is useless.

Secondly, if the second argument is a variable, it binds it to a list without duplicates. For example,

$$\begin{aligned} & member(A, l) \\ \Longrightarrow\ & (\exists x.\, l = A.x \rightarrow true); \\ & (\exists b.\, \exists x.\, l = b.x \wedge \neg\, b = A \rightarrow member(A, x)) \\ \Longrightarrow\ & (\exists x.\, l = A.x \rightarrow true); \\ & (\exists b.\, \exists x'.\, l = b.A.x' \wedge \neg\, b = A \rightarrow true); \\ & (\exists b.\, \exists b'.\, \exists x'.\, l = b.A.x' \wedge \neg\, b = A \wedge \neg\, b' = A \rightarrow member(A, x')) \\ \Longrightarrow\ & \quad \vdots \end{aligned}$$

We still get an infinite number of nondeterministic narrowings, but they all qualify the solutions by stating that the unbound elements of l are not equal to A. This is useful in evaluating an expression such as

$$member(A, l) \wedge member(A, l)$$

Evaluating this by the member function of example 1 bound l to lists with two occurrences of A. But, by our new definition of *member*, l cannot get bound to a list with duplicates. Notice the second narrowing of this expression:

$$
\begin{aligned}
&(\exists b.\, \exists x'.\, l = b.A.x' \wedge \neg\, b = A) \rightarrow member(A, b.A.x') \\
&\Longrightarrow \exists b.\, \exists x'.\, l = b.A.x' \wedge \neg\, b = A \rightarrow member(A, b.A.x') \\
&\Longrightarrow \;\; (\exists b.\, \exists x'.\, l = b.A.x' \wedge \neg\, b = A \rightarrow b = A \rightarrow true); \\
&\qquad (\exists b.\, \exists x'.\, l = b.A.x' \wedge \neg\, b = A \rightarrow \neg\, b = A \rightarrow member(A, A.x')) \\
&\Longrightarrow \omega;\ (\exists b.\, \exists x'.\, l = b.A.x' \wedge \neg\, b = A \rightarrow true) \\
&\Longrightarrow \exists b.\, \exists x'.\, l = b.A.x' \wedge \neg\, b = A \rightarrow true
\end{aligned}
$$

The first choice in the above fails while taking the composition of substitutions $\neg\, b = A$ and $b = A$. Thus the first element of l is prohibited from getting bound to A, no matter howmany times we test for the membership of A in l.

Similarly, the *retrieve* function has to be defined making explicit use of the assumption that an association list contains at most one pair for each key.

$$retrieve(key, (k, a).x) \equiv (key = k \rightarrow a;\ \neg\, key = k \rightarrow retrieve(key, x))$$

Now, consider the evaluation of *translate*.

$$
\begin{aligned}
&translate\ (Use, A).(Def, A).Nil \\
&\Longrightarrow \exists table.\ (Use, retrieve(A, table)).(retrieve(A, table) = 1 \rightarrow (Assign, A, 1)).Nil
\end{aligned}
$$

If the first *retrieve* application binds *table* to $(k, a).(A.a').t'$, it also places the condition (negative substitution) that $(\neg\, k = A)$. So, the second application of *retrieve* yields a' rather than a as its result, which then gets bound to 1. Thus, independent of which narrowing of the *retrieve* application is used, the result is of the form

$$\exists \overline{x}.(Use, 1).(Assign, A, 1).Nil$$

and hence the definition of *translate* is consistent.

5 Set abstraction and nonground outputs

Even though our basic narrowing mechanism introduced new free variables in the evaluation of an expression, we saw that the new variables are not really free but are implicitly existentially quantified. We made this quantification explicit in section 3. Existential quantification also gives us a way to introduce an unbound variable and bind it within its scope. But, it does not allow us to export the variable outside its scope and bind it elsewhere. For example, we cannot say

$$\textbf{let}\ a \equiv \exists x.\, x\ \textbf{in}\ a = 1$$

because the expression $\exists x.\, x$ is inconsistent and denotes $\top$.

This example may appear somewhat esoteric, but exporting logical variables from their scope is quite useful. In the permutation inversion problem, we assumed a primitive $vector(v, n)$ which binds v to a vector of n variables if v is a variable. Since, we know that v is actually an output of this operation, we may consider using a primitive (*a-vector n*) which yields a vector of n variables.

$$
\begin{aligned}
&\exists v.\ v = (a\text{-}vector\ n) \wedge \\
&\quad forall(ints\text{-}between(1, n),\ \lambda i.\ v(p(i)) = i) \\
&\quad \rightarrow v
\end{aligned}
$$

Id Nouveau [NPA86] has a primitive called *array* which behaves like *a-vector*. But, the primitive is inconsistent, just like $\exists x.\, x$, since it exports logical variables. Another example of the use of variable exportation is the following type inference program [Mil78]. The function $type(e, te)$ yields the type of e in the type environment te represented as an association list.

$$\begin{array}{lcl} type(x, te) & \equiv & (variable?\ x) \rightarrow retrieve(x, te) \\ type((Apply, f, e), te) & \equiv & \exists t.\, \exists u.\ type(f, te) = (Fun, t, u)\ \wedge type(e, te) = t \rightarrow u \\ type((Lambda, x, e), te) & \equiv & \exists t.(Fun, t, type(e, (x, t).te) \end{array}$$

The last equation can produce inconsistent results if t does not get bound in the subexpression $type(e, (x, t).te)$. For example,

$$type((Lambda, X, X), Nil) \Longrightarrow \exists t.\ (Fun, t, t)$$

This is an inconsistent expression and denotes $\top$.

The solution to these problems, suggested in [Red86a,DFP86], is to treat such terms not as denoting values, but as denoting *sets of values*. We use the binding construct $(\bigcup x.\ \{e\})$ for this purpose. Whereas $(\exists x.\ x)$ denotes $\top$, $(\bigcup x.\ \{x\})$ denotes the set of all values. $(\bigcup t.\ \{(Fun, t, t)\})$ denotes the set of all terms of the form (Fun, t, t). The expression $(Lambda, X, X)$ does not have a single type, but a whole set of types of the form (Fun, t, t). Similarly, there is not a single vector of length n, but a whole set of vectors. The construct $(\bigcup x.\ \{e\})$ is called *absolute set abstraction* in [DFP86] and a *free set* in [Red86a].

Absolute set abstraction is not the only way to handle exportation of logical variables. We could treat all expressions as being indeterminate (implicitly denoting sets), and use many-to-many mappings instead of functions. Church-Rosser property would then be lost. This approach is taken in [Smo86].

We introduce the set concepts into our language by extending it with two new syntactic domains:

Set, ranged over by s

Set-function, ranged over by ψ

$$\begin{array}{lcll} s & ::= & \emptyset & \text{- empty set} \\ & & \mid \{e\} & \text{- singleton} \\ & & \mid (s_1 \cup s_2) & \text{- set union} \\ & & \mid (e \parallel s) & \text{- apply to all} \\ & & \mid (\psi\ e) & \text{- set function application} \\ & & \mid (e \rightarrow s) & \text{- conditional set} \\ & & \mid (\bigcup x.\, s) & \text{- absolute set abstraction} \\ \psi & ::= & (\lambda t.\, s) & \text{- abstraction} \\ & & \mid (\psi_1 \cup \psi_2) & \text{- union of results} \end{array}$$

We also extend expressions by a new construct for testing set membership:

$$e \ ::= \ (e \in s)$$

Sets are not first-class values in the language. There are no functions on sets. There are also no higher type sets whose members are sets. However, the members of sets can be any values including functions.

We define the following constructs by syntactic sugar:

$$\begin{array}{rcl} \{e_1, e_2, \ldots, e_n\} & \overset{\text{def}}{=} & \{e_1\} \cup \{e_2\} \cup \cdots \cup \{e_n\} \\ \textbf{collect}\ t \leftarrow s\ \textbf{in}\ e & \overset{\text{def}}{=} & ((\lambda t.\ e) \parallel s) \\ (member\ s) & \overset{\text{def}}{=} & \exists x.\ x \in s \rightarrow x \end{array}$$

Th **collect** construct provides the qualified set abstraction used by Turner [Tur81]. Note that the binding of x in this construct is *strong*, whereas the binding in the similar-looking expression

$$(\bigcup \overline{x}.\ t \in s \rightarrow e)$$

is *weak*. This is similar to the difference between let binding and = binding. If all the members of a set s are consistent, we can get their lub using the construct (*member* s). Note that *member* cannot be defined as a function since functions cannot take sets as arguments.

The denotational semantics of the extended language uses new semantic domains:

$$\begin{aligned} [\![s]\!] &: Env \to P(D) \\ [\![\psi]\!] &: Env \to [D \to P(D)] \\ \text{where} & \\ Env &= Variable \to D \end{aligned}$$

We have not yet completely investigated the required structure of $P(D)$. But, since we do not have functions on sets, we believe it is adequate to use the Hoare powerdomain of D. Its elements are the nonempty, downward-closed and limit-closed subsets of D and its partial order is the subset order. The semantics of the new constructs follows:

$$\begin{aligned}
[\![\emptyset]\!]\eta &= \{\perp\} \\
[\![\{e\}]\!]\eta &= \{[\![e]\!]\eta\} \\
[\![s_1 \cup s_2]\!]\eta &= [\![s_1]\!]\eta \cup [\![s_2]\!]\eta \\
[\![(e \parallel s)]\!]\eta &= \{([\![e]\!]\eta\ v) \mid v \in [\![s]\!]\eta\} \\
[\![(\psi\ e)]\!]\eta &= ([\![\psi]\!]\eta\ [\![e]\!]\eta) \\
[\![(e \to s)]\!]\eta &= \begin{cases} [\![s]\!]\eta & \text{if } [\![e]\!]\eta = true \\ D & \text{if } [\![e]\!]\eta = \top \\ \emptyset & \text{otherwise} \end{cases} \\
[\![(\bigcup x.\, s)]\!]\eta &= \bigcup_{v \in Dcon} [\![s]\!](\eta; [v/x]) \\
[\![\lambda t.\, s]\!]\eta &= \lambda v.\ \text{if } \exists \rho \in FV(t) \to D \text{ such that } [\![t]\!]\rho = v \text{ then } [\![s]\!](\eta; \rho) \text{ else } \{\perp\} \\
[\![\psi_1 \cup \psi_2]\!]\eta &= \lambda v.\ ([\![\psi_1]\!]\eta\ v) \cup ([\![\psi_2]\!]\eta\ v) \\
[\![e \in s]\!]\eta &= \begin{cases} true & \text{if } [\![e]\!]\eta \in B, [\![e]\!]\eta \in [\![s]\!]\eta \\ \top & \text{if } [\![e]\!]\eta = \top \\ \perp & \text{otherwise} \end{cases}
\end{aligned}$$

The operational semantics of the new constructs closely follows the rules we have given before. To see the correspondence, note that $\emptyset$ is similar to ω, $\cup$ is similar to ";", and $\bigcup$ is similar to $\exists$. We again have several classes of rules: normal rules, failure rules, substitution propagation rules, failure (empty set) propagation rules and union propagation rules. The normal rules are given below. In addition, the rules APP1 through APP3′ hold for set function application, and COND and COND′ hold for set conditional.

APP-ALL:	$(e \parallel \{e'\})$	$\Longrightarrow$	$\{(e\ e')\}$
α-ABS:	$(\bigcup x.\, s)$	$\Longrightarrow$	$(\bigcup x'.\, [x'/x][s])$ if x is not free in s
ABS1:	$(\bigcup x.\, \{t\})$	$\Longrightarrow$	$\{t\}$ if x is not free in t
ABS2:	$(\bigcup x.\, \bigcup \overline{y}.\, x = t \to s)$	$\Longrightarrow$	$(\bigcup \overline{y}.\, s)$ if x is not free in t or s
ABS3:	$(\bigcup x.\, \bigcup \overline{y}.\, x = t \wedge \sigma \to s)$	$\Longrightarrow$	$(\bigcup \overline{y}.\, \sigma \to s)$ if x is not free in t, σ or s
MEMBER:	$e \in \{e'\}$	$\Longrightarrow$	$e = e'$

Note the additional condition in ABS1 which was not present in EXIST1. If x is free in t, then $(\bigcup x.\, \{t\})$ is a terminal form. The failure rules and failure propagation rules are:

SET-APP-FAIL2:	$((\lambda a.\, s)\ t)$	$\Longrightarrow$	$\emptyset$	if $t \neq a$
SET-APP-FAIL3:	$((\lambda(t_1,t_2).\, d)\ a)$	$\Longrightarrow$	$\emptyset$	
SET-COND-FAIL:	$(t \rightarrow s)$	$\Longrightarrow$	$\emptyset$	if $t \neq \mathit{true}$
FAIL-SINGLE:	$\{\omega\}$	$\Longrightarrow$	$\emptyset$	
FAIL-APP-ALL:	$(e \parallel \emptyset)$	$\Longrightarrow$	$\{(e\ \omega)\}$	
FAIL-ABS:	$\bigcup x.\, \emptyset$	$\Longrightarrow$	$\emptyset$	
FAIL-MEMBER:	$e \in \emptyset$	$\Longrightarrow$	ω	

The substitution propagation rules are:

PROP-APP-ALL2:	$(e \parallel (\bigcup \bar{x}.\, \sigma \rightarrow s))$	$\Longrightarrow$	$\bigcup \bar{x}.\, \sigma \rightarrow (\sigma[e] \parallel s)$ if $\bar{x}$ are not free in e
PROP-SET-APP2:	$((\lambda t.\, s)\ (\exists \bar{x}.\, \sigma \rightarrow e))$	$\Longrightarrow$	$(\bigcup \bar{x}.\, (\sigma[\lambda t.\, s]\ e))$ if t is not a variable, and $\bar{x}$ are not free in $\lambda t.\, s$
PROP-SET-COND:	$((\exists \bar{x}.\, \sigma \rightarrow e) \rightarrow s)$	$\Longrightarrow$	$\bigcup \bar{x}.\, \sigma \rightarrow (e \rightarrow \sigma[s])$ if $\bar{x}$ are not free in s
COMP-SET:	$(\sigma_1 \rightarrow (\bigcup \bar{x}.\, \sigma_2 \rightarrow s))$	$\Longrightarrow$	$(\bigcup \bar{x}.\, \sigma_1 \circ \sigma_2 \rightarrow s)$ if the variables $\bar{x}$ are not free in σ_1
PROP-ABS:	$(\bigcup x.\, \bigcup \bar{y}.\, \sigma \rightarrow s)$	$\Longrightarrow$	$(\bigcup \bar{y}.\, \sigma \rightarrow (\bigcup x.\, s))$ if x is not free in σ
PROP-MEMBER:	$e \in \bigcup \bar{x}.\, \sigma \rightarrow s$	$\Longrightarrow$	$\exists \bar{x}.\, \sigma \rightarrow (\sigma[e] \in s)$ if $\bar{x}$ are not free in e

Finally, the union rules are

UNION-ELIM1:	$\emptyset \cup s$	$\Longrightarrow$	s
UNION-ELIM2:	$s \cup \emptyset$	$\Longrightarrow$	s
UNION-COM:	$(s_1 \cup s_2)$	$\Longrightarrow$	$(s_2 \cup s_1)$
UNION-ASSOC:	$((s_1 \cup s_2) \cup s_3)$	$\Longrightarrow$	$(s_1 \cup (s_2 \cup s_3))$
UNION-APP-ALL2:	$(e \parallel (s_1 \cup s_2))$	$\Longrightarrow$	$(e \parallel s_1) \cup (e \parallel s_2)$
CHOICE-APP-ALL2:	$(\lambda t.\, d \parallel \{e_1; e_2\})$	$\Longrightarrow$	$(\lambda t.\, d \parallel \{e_1\}) \cup (\lambda t.\, d \parallel \{e_2\})$ if t is not a variable
UNION-SET-APP1:	$((\psi_1 \cup \psi_2)\ e)$	$\Longrightarrow$	$(\psi_1\ e) \cup (\psi_2\ e)$
CHOICE-SET-APP2:	$((\lambda t.\, s)\ (e_1; e_2))$	$\Longrightarrow$	$((\lambda t.\, s)\ e_1) \cup ((\lambda t.\, s)\ e_2)$ if t is not a variable
CHOICE-SET-COND1:	$((e_1; e_2) \rightarrow s)$	$\Longrightarrow$	$(e_1 \rightarrow s) \cup (e_2 \rightarrow s)$
UNION-SET-COND2:	$(\sigma \rightarrow (s_1 \cup s_2))$	$\Longrightarrow$	$(\sigma \rightarrow s_1) \cup (\sigma \rightarrow s_2)$
UNION-ABS:	$(\bigcup x.\, s_1 \cup s_2)$	$\Longrightarrow$	$(\bigcup x.\, s_1) \cup (\bigcup x.\, s_2)$
UNION-MEMBER:	$e \in (s_1 \cup s_2)$	$\Longrightarrow$	$e \in s_1;\ e \in s_2$

Conspicuously missing are the rules for propagating choice or quantifiers through the singleton construct. The problem is that a rule such as

$$\{(e_1; e_2)\} \Longrightarrow \{e_1\} \cup \{e_2\}$$

does not preserve the semantics. On the left, we have a singleton whose element is approximated by e_1 and e_2. On the right, we have a set of the two approximations, but the element itself is missing from it. But, we believe that a rule such as this would be practically useful. Further investigation is needed into the nature of this problem.

Let us now show some examples of the use of absolute set abstraction.

Example 5 (Vectors using sets) As promised at the beginning of the section, we can now write the permutation inversion function with exportation of logical variables. The primitive required for this is (*vectors* n) which yields the set of all vectors of length n as an absolute set abstraction:

$$\bigcup x_1.\bigcup x_2.\ldots\bigcup x_n.\{[x_1,x_2,\ldots,x_n]\}$$

Now, *inverse* can be defined as

$$\begin{array}{rl} \textit{inverse}\ p \equiv & \textbf{let}\ \ n \equiv (\textit{size}\ p) \\ & \textbf{in}\ \ \exists v.\,(v \in (\textit{vectors}\ n) \wedge \textit{forall}(\textit{ints-between}(1,n),\ \lambda i.v(p(i)) = i)) \rightarrow v \end{array}$$

This definition has the same effect as that in example 4 since, by rules PROP-MEMBER and MEMBER

$$(v \in \bigcup x_1.\ldots\bigcup x_n.\{[x_1,\ldots,x_n]\}) \Longrightarrow (\exists x_1.\ldots\exists x_n.\,v = [x_1,\ldots,x_n])$$

In Id Nouveau [NPA86], the array construct is found to be referentially opaque, since

$$\begin{array}{l} \textbf{let}\ \ x \equiv \textit{array}(1..n) \\ \textbf{in}\ \ (x,x) \end{array}$$

is not equivalent to

$$(\textit{array}(1..n),\ \textit{array}(1..n))$$

This can be avoided by using our *vectors* primitive:

$$\begin{array}{l} \textbf{collect}\ \ x \leftarrow (\textit{vectors}\ n) \\ \textbf{in}\ \ (x,x) \end{array}$$

is semantically different from

$$((\textit{vectors}\ n),\ (\textit{vectors}\ n))$$

The former is a set of pairs, and the latter is a pair of sets.

Example 6 (Retrieving multiple attributes) Suppose, unlike in example 2, we have association lists in which a key can appear in any number of pairs with different attributes. To obtain all the attributes of a key, we can use

$$\textit{retrieve-all}(\textit{key},\textit{alist}) \equiv \bigcup \textit{attr}.\ \textit{member}((\textit{key},\textit{attr}),\textit{alist}) \rightarrow \{\textit{attr}\}$$

The body of the definition is our syntax for what we would write in set theory as

$$\{\textit{attr} \mid \textit{member}((\textit{key},\textit{attr}),\textit{alist})\}$$

Here is a sample evaluation using it:

$$\begin{array}{l} \textit{retrieve-all}(A,(A,1).(A,2).\textit{Nil}) \\ \Longrightarrow \bigcup \textit{attr}.\ \textit{member}((A,\textit{attr}),\ (A,1).(A,2),\textit{Nil}) \rightarrow \{\textit{attr}\} \\ \Longrightarrow \bigcup \textit{attr}.\ (\textit{attr} = 1 \rightarrow \textit{true};\ \textit{attr} = 2 \rightarrow \textit{true}) \rightarrow \{\textit{attr}\} \\ \Longrightarrow \bigcup \textit{attr}.\ (\textit{attr} = 1 \rightarrow \{1\}) \cup (\textit{attr} = 2 \rightarrow \{2\}) \qquad \text{by CHOICE-SET-COND1} \\ \Longrightarrow (\bigcup \textit{attr}.\,\textit{attr} = 1 \rightarrow \{1\}) \cup (\bigcup \textit{attr}.\,\textit{attr} = 2 \rightarrow \{2\}) \qquad \text{by UNION-ABS} \\ \Longrightarrow \{1\} \cup \{2\} \qquad \text{by ABS2} \end{array}$$

Example 7 (Type inference) The type inference program discussed at the beginning of this section can be written using sets as

$$\begin{array}{lcl} \textit{types}(x,\textit{te}) & \equiv & (\textit{variable?}\ x) \rightarrow \{\textit{retrieve}(x,\textit{te})\} \\ \textit{types}((\textit{Apply},f,e),\textit{te}) & \equiv & \bigcup t.\bigcup u.\ (\textit{Fun},t,u) \in \textit{types}(f,\textit{te}) \wedge t \in \textit{types}(e,\textit{te}) \rightarrow \{u\} \\ \textit{types}((\textit{Lambda},x,e),\textit{te}) & \equiv & \bigcup t.\ \textbf{collect}\ u \leftarrow \textit{types}(e,(x,t).\textit{te}) \\ & & \qquad \textbf{in}\ (\textit{Fun},t,u) \end{array}$$

The multiple definitions for the function are combined using the *union* operator. So, these equations for *types* represent a definition of the form

$$\lambda(x,te).\ \ldots \cup \lambda((Apply,f,e),te).\ \ldots \cup \lambda((Lambda,x,e),te).\ \ldots$$

The type of the identity function is now inferred as:

$$\begin{aligned}
&types((Lambda,X,X),Nil)\\
&\Longrightarrow \bigcup t.\ \textbf{collect}\ u \leftarrow types(X,(X,t).Nil)\ \textbf{in}\ (Fun,t,u)\\
&\Longrightarrow \bigcup t.\ \textbf{collect}\ u \leftarrow \{t\}\ \textbf{in}\ (Fun,t,u)\\
&\Longrightarrow \bigcup t.\ \{(Fun,t,t)\}
\end{aligned}$$

and the type of a sample application is inferred as:

$$\begin{aligned}
&types((Apply,(Lambda,X,X),(Lambda,X,X)),\ Nil)\\
&\Longrightarrow \bigcup t.\bigcup u.\ ((Fun,t,u)\in \bigcup t'.\{(Fun,t',t')\})\ \wedge\ (t\in \bigcup t''.\{(Fun,t'',t'')\})\rightarrow\{u\}\\
&\Longrightarrow \bigcup t.\bigcup u.\ (\exists t'.\,t=t'\wedge u=t')\wedge(\exists t''.\,t=(Fun,t'',t''))\rightarrow\{u\}\\
&\Longrightarrow \bigcup t.\bigcup u.\ (\exists t''.\,t=(Fun,t'',t''))\wedge u=(Fun,t'',t''))\rightarrow\{u\}\\
&\Longrightarrow \bigcup t.\bigcup u.\bigcup t''.\,t=(Fun,t'',t'')\wedge u=(Fun,t'',t'')\rightarrow\{(Fun,t'',t'')\}\\
&\Longrightarrow \bigcup t''.\ \{(Fun,t'',t'')\}
\end{aligned}$$

6 Conclusion

We first described our ideas on using narrowing as the operational semantics of functional languages in [Red85]. We then proceeded to build an implementation of this semantics, whose features are described in [Red86b]. This implementation effort showed us that our understanding of narrowing was still inadequate, in particular its relationship to lazy evaluation. We also found several problems regarding efficiency which we had no easy language to describe in or investigate. Hence we were led to a formalization of the basic semantics, presented here.

Representing substitutions as expressions and using explicit existential quantifiers are the novel features of this presentation. This formalization shows that logical variables do not go well with lazy data structures. This is clear from the fact that, for lazy pairs,

$$\exists x.\,(e_1,e_2)=(\exists x.e_1,\exists x.e_2)$$

So, each component of a lazy data structure can use its own instantiations of logical variables. This is quite contrary to our intuitive ideas of logic programming. Much further investigation is required into the interaction between lazy evaluation and logical variables.

Another aspect brought up by this formalization is the importance of negation. The examples 1, 2 and 3 show this clearly. Representing substitutions as expressions allows us to easily extend them to negative substitutions. But, several questions still need to be answered. For example, what is meant by applying a negative substitution on an expression? Similar questions also arise in higher-order narrowing, i.e. in solving for function variables. We shall present some solutions to these problems in a future paper.

7 Acknowledgements

Discussions with Arvind and Paul Hudak contributed to the ideas presented in this paper.

This research was supported by the University of Illinois at Urbana-Champaign.

References

[BMS80] R. M. Burstall, D. B. MacQueen, and D. T. Sanella. Hope: an experimental applicative language. In *ACM LISP Conference*, pages 136–143, 1980.

[CKPR73] A. Colmerauer, H. Kanouri, R. Pasero, and P. Roussel. *Un Systeme de Communication Homme-machine en Francais.* Research Report, Groupe Intelligence Artificielle, Universite Aix-Marseille II, 1973.

[CM81] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, 1981.

[DFP86] J. Darlington, A. J. Field, and H. Pull. The unification of functional and logic languages. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations and Equations*, pages 37–70, Prentice-Hall, 1986.

[Fay79] M. Fay. First-order unification in an equational theory. In *Fourth Workshop on Automated Deduction*, pages 161–167, Austin, Texas, 1979.

[Hul80] J-M. Hullot. Canonical forms and unification. In *Conference on Automated Deduction*, pages 318–334, 1980.

[Kow79] R. A. Kowalski. *Logic for Problem Solving.* North-Holland, 1979.

[Kow83] R. Kowalski. Logic programming. In R. E. A. Mason, editor, *Information Processing*, pages 133–145, North-Holland, 1983.

[Lin85] G. Lindstrom. Functional programming and the logical variable. In *ACM Symposium on Principles of Programming Languages*, 1985.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Mil84] R Milner. A proposal for Standard ML. In *ACM Symposium on LISP and Functional Programming*, pages 184–197, 1984.

[NPA86] R.S. Nikhil, K. Pingali, and Arvind. *Id Nouveau.* Technical Report CSG 265, MIT, 1986.

[Red85] U. S. Reddy. Narrowing as the operational semantics of functional languages. In *Symposium on Logic Programming*, IEEE, Boston, 1985.

[Red86a] U. S. Reddy. On the relationship between logic and functional languages. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations and Equations*, pages 3–36, Prentice-Hall, 1986.

[Red86b] U.S. Reddy. *Logic Languages based on Functions: Semantics and Implementation.* Technical Report UIUCDCS-R-86-1305, University of Illinois at Urbana-Champaign, 1986. Ph.D. thesis done at University of Utah.

[Sla74] J. R. Slagle. Automated theorem-proving for theories with simplifiers, commutativity and associativity. *Journal of the ACM*, 21(4):622–642, 1974.

[Smo86] G. Smolka. Fresh: a higher-order language based on unification. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations and Equations*, pages 469–524, Prentice-Hall, 1986.

[SS84] M. Sato and T. Sakurai. Qute: a functional language based on unification. In *Intl. Conf. Fifth Generation Computer Systems*, pages 157–165, ICOT, 1984.

[Tur81] D.A Turner. The semantic elegance of applicative languages. In *Proceedings of the 1981 conference on Functional Programming Languages and Computer Architecture*, pages 85–92, ACM, 1981.

[Tur85] D.A. Turner. Miranda: a non-strict functional language with polymorphic types. In J-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 1–16, Springer-Verlag, 1985.