Higher-order Aspects of Logic Programming

Uday S. Reddy

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
Net: reddy@cs.uiuc.edu

Abstract. Are higher-order extensions to logic programming needed? We answer this question in the negative by showing that higher-order features are already available in pure logic programming. It is demonstrated that higher-order lambda calculus-based languages can be compositionally embedded in logic programming languages preserving their semantics and abstraction facilities. Further, we show that such higher-order techniques correspond to programming techniques often practiced in logic programming.
Keywords: Higher-order features, functional programming, lambda calculus, logic variables, concurrent logic programming, types, semantics.

1 Introduction

In an early paper [War82], Warren raised the question whether higher-order extensions to Prolog were necessary. He suggested that they were not necessary by modelling higher-order concepts in logic programs. His proposal modelled functions and relations *intensionally* by their names. As is well-known in Lisp community, such an encoding does not obtain the full power of higher-order programming. In particular, lambda abstraction and "upward" function values are absent.

An entirely different approach to the issue was pursued by Miller et. al. in Lambda Prolog [MN86]. The objective here seems to be not higher-order logic programming, but logic programming over higher-order terms. While this approach has many interesting applications, particularly in meta-logical frameworks, it still leaves open the question of whether higher-order programming is possible in a standard logic programming setting. A similar comment applies to other higher-order languages like HiLog [CKW89].

In this paper, we return to Warren's question and answer it in the *negative*. Using new insights obtained from the connection between linear logic and logic programming [Laf87, Abr91, Abr93, Red93b], we contend that idealized logic programming is "already higher-order" in its concept, even though one might wish to add some syntactic sugar to obtain convenience. A similar answer is also suggested by Saraswat's work [Sar92] showing that concurrent logic programming is higher-order in a categorical sense.

We argue that idealized logic programming is higher-order by presenting two forms of evidence:

– We exhibit a compositional, semantics-preserving translation from higher-order lambda calculus-based languages to logic programs.
– We show that higher-order programming techniques are already in use in logic programming (albeit in a limited form) and that these can be generalized.

Two features of logic programming play important role in this modelling: *logic variables* and *concurrency*. Using logic variables, one often constructs data structures with multiple occurrences of variables, e.g., difference lists [CT77]. Typically, one of the occurrences of the variable is designated for input, i.e., a binding is given to the variable via this occurrence, and other occurrences are used as output. If we form a pair (x, S_x) consisting of a variable x and a structure S_x with embedded occurrences of x (see Fig. 1), we have essentially formed a "function" value. The function is "applied" to an argument by binding the variable x and the result of application is obtained via S_x. Whereas this scheme allows S_x to be a data structure, using concurrency, one can similarly build *computational* structures with shared variables. S_x is then thought of as a "process" and a pair of variables involved in the process is viewed as a function value. All this provides "functions" that are good for only one function application. General functions that can be used multiple times are modelled by generating entire streams of such pairs, again using a process-oriented view. The streams are generated in a demand-driven fashion so that the user of the stream can terminate the stream at will. Mathematically, such streams are best viewed as *finite sets*.

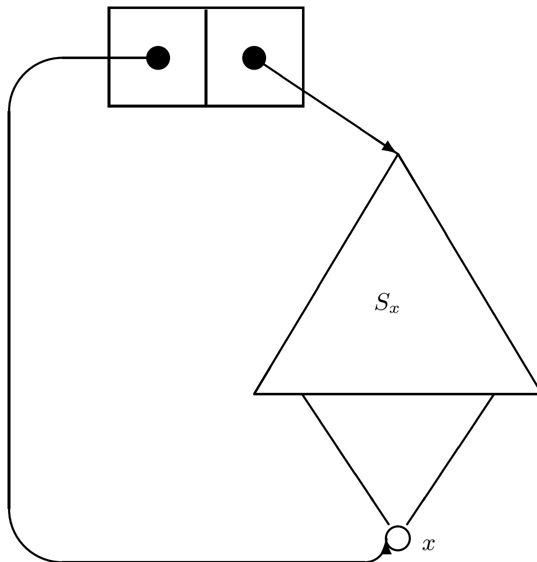

Fig. 1. Difference structures

Concurrency is often thought of as an extraneous feature of logic programming. However, idealized logic programs executed with *fair* SLD-resolution [LM84] are necessarily concurrent. Under a fair resolution scheme, no goal can be delayed forever. So, each goal of a query must be thought of as a separate process which cooperates with other processes to solve the problem. Our modelling of higher-order concepts in this framework. This cannot be duplicated in sequential logic programming languages like Prolog due to the crucial role of concurrency already mentioned. However, our scheme is applicable to almost all concurrent logic programming languages such as Concurrent Prolog [Sha83], Parlog [Gre87] and Guarded Horn clauses [Ued86] as well as languages with coroutine facilities like Mu-Prolog [Nai86].

1.1 Related work

From a theoretical point of view, the fact that higher-order functions can be simulated by relations seems to have been folklore in category theory for some time. See [Bar91] for an explicit presentation. Within the programming language theory, it is implicit in the work of [Laf87, Abr91, Red93b] which shows that linear logic proofs can be modelled as logic programs. Another piece of related work is Milner's embedding of lambda calculus in pi-calculus [Mil90]. His embedding only uses a subset that has a direct correspondence to concurrent logic programming languages. Thus, our presentation documents what has been implicitly known in various programming language circles.

Many of our ideas are also implicit in Shapiro's work on modelling objects in concurrent logic programming [Sha83, ST86]. Since objects provide collections of operations to their clients, modelling objects involves modelling function values as well. In fact, as shown in [Red93a], our translations easily extend to higher-order imperative programs (object-based programs) using the same ideas as Shapiro's. Somewhat more explicit is the work of Saraswat [Sar92] where it is shown in that the semantic model of concurrent constraint programs forms a cartesian closed category. In contrast to this work, we show here that standard logic programming has the same kind of properties.

2 The framework

We find it convenient to treat logic programs as defining *binary* relations. So, a predicate definition looks like:

$$\begin{array}{l} R : A \leftrightarrow B \\ R(x, y) \Leftrightarrow \psi \end{array}$$

where A and B are types, and ψ is a formula built from predicate applications, equality, conjunction, disjunction and existential quantification. (We often suppress the existential quantifiers and disjunctions by using Horn clause notation

$R(x, y) \leftarrow \psi$). Typically, the type A is a finite product type, in which case we write:

$$R : A_1 \times \ldots \times A_k \leftrightarrow B$$
$$R(x_1, \ldots, x_k, y) \Leftrightarrow \psi$$

Note that we suppress the parentheses around the tuple $(x_1, \ldots, x_k)$, to avoid clutter. The special case of $k = 0$ appears as:

$$R : \mathit{unit} \leftrightarrow B$$
$$R(y) \Leftrightarrow \psi$$

Types include

- primitive types such as *int* and *bool*,
- finite products, including the nullary product called *unit*, and
- finite disjoint unions, including the nullary version $\emptyset$.

We often write disjoint unions with *constructor* symbols for injections, *e.g.*,

type *result* = fail(*unit*) + succ(*int*)

which is syntactic sugar for $\mathit{unit} + \mathit{int}$. All symbols other than constructors of this kind are variables (usually x, y, z and X, Y, Z). We do not use a case convention for variables. We also allow polymorphic type constructions with recursive definitions, *e.g.*,

type *list*(α) = nil(*unit*) + α.*list*(α)

which denotes the least set L such that $L = \mathit{unit} + \alpha \times L$. All these constructions have well-defined semantics. See, for example, [LR91].

3 Embedding PCF

PCF (Programming Language for Computable Functions), due to Scott [Sco69], is a higher-order functional programming language. (See [Plo77] and [Gun92, 4.1] for published descriptions.) PCF is a typed lambda calculus with a single primitive type for integers. (The boolean type can be omitted without loss of expressiveness.) In this section, we show how PCF can be embedded in a logic programming language. The modelling of functions being the crux of the problem, extension to other kinds of data structures is more or less immediate.

Examples We start by showing some examples. First, consider "linear" functions, *i.e.*, functions that use their arguments exactly once. This restriction allows us to introduce a simpler model first. Here is an example:

$\mathit{inc} : \mathit{int} \rhd \mathit{int}$	$\mathit{inc} : \mathit{int} \leftrightarrow \mathit{int}$
inc(x) = x + 1	inc(x, y) $\Leftrightarrow$ add(x, 1, y)

On the left, we have a term $x + 1$ treated as a function of x. We always use the symbol $\triangleright$ to denote this kind of implicit function that occurs at the top level. This function is modelled as a relation between integers. Such modelling of functions by relations is elementary for any one with some knowledge of set theory.

In lambda calculus, one can abstract over x to form a function term:

$succ : \ \triangleright\, int \to int$	$succ : unit \leftrightarrow (int \times int)$
succ = λx. x + 1	succ((x, y)) ⇔ add(x, 1, y)

The function space $int \to int$ is modelled by the product $int \times int$. If this seems surprising, consider the fact that a relation $A \leftrightarrow B$ is really like a function $A \to \mathbb{P}B$. Due to the implicit power set construction, smaller cardinalities suffice in the second argument position of the relation. The point of this example is to illustrate that lambda-abstraction is modelled by *pair formation* in a logic program. This is the same kind of pair formation as that shown in Fig. 1 except that the "difference" between the input and output is a computation rather than a data structure.

A generic function composition term is modelled as follows:

$compose : (A \to B) \times (B \to C) \ \triangleright (A \to C)$
compose(f, g) = λx. g (f x)

$compose : (A \times B) \times (B \times C) \leftrightarrow (A \times C)$
compose(f, g, (x, z)) ← f = (x, y), g = (y, z)

Notice that a function application, such as fx, is modelled by pair decomposition: $f = (x, y)$. We can use *compose* to make a twice composition of the *succ* function as follows:

succ(f), succ(g), compose(f, g, h)

Note that h denotes the required composition: $h = (x, z)$ if and only if $z = x+2$. This would correspond to the PCF term $compose(succ, succ) = \lambda x.\, (\lambda x'.\, x' + 1)$ $((\lambda x'.x' + 1)x)$. Abstraction over f and g yields a composition "function" as a term:

$comp : \ \triangleright (A \to B) \to ((B \to C) \to (A \to C))$
comp = λf. λg. λx. g (f x)

$comp : unit \leftrightarrow (A \times B) \times ((B \times C) \times (A \times C))$
comp((f, (g, (x, z)))) ← f = (x, y), g = (y, z)

Consider the PCF term *comp succ*. This denotes a higher-order function which, when applied to any function $g : int \to int$, yields the function $\lambda x.\, g(x + 1)$. Its effect is obtained by the logic programming query:

succ(f), comp(f, h)

Note that $h = (g, (x, z))$ if and only if $g = (x + 1, z)$. Thus, the above scheme of translation works for functions of any order as long as the arguments of functions are used linearly.

Control Let us pause to examine the control issues. A query involving higher-order values, e.g., $succ(f), succ(g), compose(f, g, h), h = (1, z)$, must not be evaluated sequentially. Notice that $succ(f)$ reduces to $add(x, 1, y)$ which has an infinite number of solutions. The best order of evaluation would be the following:

$\leftarrow$ succ(f), succ(g), compose(f, g, h), h = (1, z)
$\leftarrow$ succ(f), succ(g), compose(f, g, (1, z))
$\leftarrow$ succ(f), succ(g), f = (1, y), g = (y, z)
$\leftarrow$ succ((1, y)), succ((y, z))
$\leftarrow$ add(1, 1, y), add(y, 1, z)
$\leftarrow$ add(2, 1, z)
$\leftarrow$

This order of evaluation is achieved if

- every goal literal is suspended until it matches (becomes an instance of) the head of a clause, and
- literals of primitive predicates, like *add*, are suspended until their input arguments are instantiated.

In essence, we wish to treat our clauses as *Guarded Horn clauses* [Ued86]. While most of our clauses have empty guards, the translation of conditional expressions will involve guards as well. It is interesting that the translation of functional programs should automatically give rise to GHC programs.

General functions To model general functions, we must handle multiple uses of arguments. Keeping in mind that arguments are often themselves functions, we model such multiple-use arguments as finite sets of values. For example, here is the *twice* function that composes its argument with itself:

$twice : (A \to A) \rhd (A \to A)$
twice(f) = λx. f (f x)

$twice : \mathbb{F}(A \times A) \leftrightarrow (A \times A)$
twice(F, (x, z)) $\leftarrow$ F = {f1, f2}, f1 = (x, y), f2 = (y, z)

Here, the input to the *twice* relation is a finite set of pairs, *i.e.*, a finite piece of a function graph. The *twice* function extracts two (not necessarily distinct) pairs from such a piece.

How do we make such pieces of function graphs? We first illustrate it for the *succ* function:

$mksucc : unit \leftrightarrow \mathbb{F}(int \times int)$
mksucc({}) $\leftarrow$
mksucc({(x, y)}) $\leftarrow$ add(x, 1, y)
mksucc(F1 $\cup$ F2) $\leftarrow$ mksucc(F1), mksucc(F2)

Note that *mksucc* makes as many pairs of the *succ* function as needed. In particular, the query $mksucc(F), F = \{f1\} \cup \{f2\}$ is equivalent to:

f1 = (x1, y1), add(x1, 1, y1), f2 = (x2, y2), add(x2, 1, y2)

Now, we can create the twice composition of the *succ* function by the following simpler goal:

mksucc(F), twice(F, h)

which corresponds to the PCF term *twice*(*succ*). If we would like to use one portion of the *succ* function graph for use with *twice* and keep the rest for other purposes, that is easy enough to do too:

mksucc(F ∪ F'), twice(F, h), . . . F' . . .

We now generalize the idea involved in *mksucc* to make finite portions of any graph. Notice that the relations of interest to us are, in general, of the form:

$$\begin{array}{l} R : A_1 \times \ldots \times A_n \leftrightarrow C \\ R(\bar{x}, u) \leftarrow \bar{x} = \bar{t}, \psi \end{array}$$

The terms $\bar{t}$ decompose the inputs $\bar{x}$, the goal ψ performs some computation using the inputs and u is the output term. We assume that existential quantification is used in ψ such that $FV(\psi) \subseteq FV(\bar{t}) \cup FV(u)$. We can denote such relations compactly as input-computation-output triples written as $\bar{t} \mid \psi \mid u$.

Suppose $R = (\bar{t} \mid \psi \mid u)$ is a relation of type $\mathbb{F}A_1 \times \ldots \times \mathbb{F}A_n \leftrightarrow C$. Then, define a generic definition scheme (or "macro") for making graphs of the outputs as follows:

$$\begin{array}{l} \textbf{mkgraph}[\bar{t} \mid \psi \mid u] : \mathbb{F}A_1 \times \ldots \times \mathbb{F}A_n \leftrightarrow \mathbb{F}C \\ \textbf{mkgraph}[\bar{t} \mid \psi \mid u] = P \text{ where} \\ \quad P(\bar{x}, \{\}) \leftarrow \bar{x} = \{\} \\ \quad P(\bar{x}, \{u\}) \leftarrow \bar{x} = \bar{t}, \psi \\ \quad P(\bar{x}, y_1 \cup y_2) \leftarrow \bar{x} = \bar{x}_1 \cup \bar{x}_2, P(\bar{x}_1, y_1), P(\bar{x}_2, y_2) \end{array}$$

Here, $\bar{x} = \{\}$ as well as $\bar{x} = \bar{x}_1 \cup \bar{x}_2$ mean their component-wise expansions.

Using the **mkgraph** scheme, we can represent the translations of PCF terms compactly. For example:

succ = λx. x + 1
  mkgraph[| add(x, 1, y) | (x, y)]
comp = λf. λg. λx. g (f x)
  mkgraph[| ∃y. F = {(x, y)}, G = {(y, z)} | (F, (G, (x, z)))]
twice = λf. comp f f
  mkgraph[Comp | Comp = {(F1, (F2, h))} | (F1 ∪ F2, h)]

One is often tempted to ask the somewhat meaningless question, "how many elements are there in a finite set produced by an instance of **mkgraph**"? Well, the answer is "as many as needed". A **mkgraph** predicate is able to produce all possible subsets of a graph. However, treating its clauses as guarded Horn clauses allows the predicate to be suspended until a client determines the surface

structure of the finite set. The **mkgraph** predicate then fills in the elements of the finite set.

The finite set type may be implemented in a logic program as an abstract data type:

$$\textbf{type}\ \ \mathbb{F}(\alpha) = \text{null} + \text{single}(\alpha) + \text{union}(\mathbb{F}(\alpha), \mathbb{F}(\alpha))$$

$$\begin{aligned}
\text{union}(X, \text{null}) &\equiv X \\
\text{union}(\text{null}, X) &\equiv X \\
\text{union}(X_1, \text{union}(X_2, X_3)) &\equiv \text{union}(\text{union}(X_1, X_2), X_3) \\
\text{union}(X_1, X_2) &\equiv \text{union}(X_2, X_1) \\
\text{union}(X, X) &\equiv X
\end{aligned}$$

The first four axioms above are the equations of a commutative monoid. The symbols *null*, *single* and *union* are constructors for finite sets. For readability, we write *null* as $\{\}$, *single*(f) as $\{t\}$, and *union*(X, Y) as $X \cup Y$. We also use the short hand $\{t_1, \ldots, t_n\}$ for $\{t_1\} \cup \ldots \cup \{t_n\}$.

It is easy to verify that the **mkgraph** scheme preserves these equivalences.

Lemma 1. *Let* $P : \mathbb{F}A_1 \times \ldots \times \mathbb{F}A_k \leftrightarrow \mathbb{F}C$ *be an instance of the* **mkgraph** *scheme, and* $y_1, y_2 \in \mathbb{F}C$ *such that* $y_1 \equiv y_2$*. Then, for all* $\bar{x}_1 \in \mathbb{F}A_1 \times \ldots \times \mathbb{F}A_k$ *such that* $P(\bar{x}_1, y_1)$*, there exists* $\bar{x}_2 \in \mathbb{F}A_1 \times \ldots \times \mathbb{F}A_k$ *such that* $\bar{x}_1 \equiv \bar{x}_2$ *and* $P(\bar{x}_2, y_2)$*.*

Proof. We show two cases. The remaining cases are similar.

- If $y_1 = Y \cup \{\}$ and $y_2 = Y$ then $\bar{x}_1 = \bar{X} \cup \{\}$, for some $\bar{X}$, and $P(\bar{X}, Y)$. Let $\bar{x}_2 = \bar{X}$.
- If $y_1 = Y \cup Y'$ and $y_2 = Y' \cup Y$ then $\bar{x}_1 = \bar{X} \cup \bar{X}'$, for some $\bar{X}$, $\bar{X}'$, and $P(\bar{X}, Y)$ and $P(\bar{X}', Y')$. Let $\bar{x}_2 = \bar{X}' \cup \bar{X}$.

□

Another method, available in a committed-choice language, is to represent finite sets by streams. Union is then obtained by merge. Here is the definition of the **mkgraph** scheme using this representation:

$$\begin{aligned}
&\textbf{mkgraph}[\bar{t} \mid \psi \mid u] : \mathit{list}(A_1) \times \ldots \mathit{list}(A_n) \leftrightarrow \mathit{list}(C) \\
&\textbf{mkgraph}[\bar{t} \mid \psi \mid u] = P \text{ where} \\
&\quad P(\bar{x}, [\,]) \leftarrow \bar{x} = [\,] \\
&\quad P(\bar{x}, u.ys) \leftarrow \mathit{merge}(\bar{x}_1, \bar{x}_2, \bar{x}), \bar{x}_1 = \bar{t}, \psi, P(\bar{x}_2, ys)
\end{aligned}$$

We can profitably use Shapiro's metaphor [Sha83] and think of a **mkgraph** predicate as an "object" and the finite set argument as a stream of "messages" sent to the object.

3.1 Translation

Table 1 gives a description of PCF types, terms and the type rules for terms. The type rules are expressed in terms of judgements of the form

$$x_1 : A_1, \ldots, x_n : A_n \ \triangleright M : C$$

where $x_1, \ldots, x_n$ are the free variables of M. We use $\Gamma, \Delta, \ldots$ to range over sequences of variable typings $x_1 : A_1, \ldots, x_n : A_n$ (called "type contexts"). All the variables in a type context must be distinct. The first type rule (called Exchange rule) allows the typings in a type context to be freely rearranged. So, the order of typings is essentially insignificant. The type rules of Table 1 look different from the usual formulations, but they are equivalent to them. They are designed to give the cleanest possible translation.

Types : A, B, C
$A ::= \mathit{int} \mid A_1 \to A_2$
Variables : x, y, z
Numbers : $n ::= 0 \mid 1 \mid \ldots$
Terms : M, N, V
$M ::= x \mid n \mid M + N \mid M - N \mid \textbf{if}\ M\ N_1\ N_2 \mid \lambda x.\, M \mid MN \mid \mathrm{rec}\ M$
Type rules :

$$\frac{\Gamma, x : A, y : B, \Gamma' \ \triangleright M : C}{\Gamma, y : B, x : A, \Gamma' \ \triangleright M : C} \qquad \frac{\Gamma, y_1 : A, y_2 : A \ \triangleright M : C}{\Gamma, x : A \ \triangleright M[x/y_1, x/y_2] : C} \qquad \frac{\Gamma \ \triangleright M : C}{\Gamma, x : A \ \triangleright M : C}$$

$$\frac{}{x : A \ \triangleright x : A} \qquad \frac{\Gamma \vdash M : A \quad \Delta, x : A \vdash N : C}{\Gamma, \Delta \vdash N[M/x] : C}$$

$$\frac{}{\triangleright n : \mathit{int}} \qquad \frac{\Gamma \ \triangleright M : \mathit{int} \quad \Delta \ \triangleright N : \mathit{int}}{\Gamma, \Delta \ \triangleright M + N : \mathit{int}} \qquad \frac{\Gamma \ \triangleright N_1 : A \quad \Gamma \ \triangleright N_2 : A}{\Gamma, x : \mathit{int} \ \triangleright \textbf{if}\ x\ N_1\ N_2 : A}$$

$$\frac{\Gamma, x : A \ \triangleright M : B}{\Gamma \ \triangleright \lambda x.\, M : A \to B} \qquad \frac{\Gamma \ \triangleright M : A \to B}{\Gamma, x : A \ \triangleright Mx : B} \qquad \frac{}{x : A \to A \ \triangleright \mathrm{rec}\ x : A}$$

Table 1. Definition of PCF

The translation of PCF programs to logic programs as follows. The types are translated by the mapping $(\ )^\circ$:

$$\mathit{int}^\circ = \mathit{int}$$
$$(A \to B)^\circ = \mathbb{F}A^\circ \times B^\circ$$

The translations represents the intuitions explained earlier in this section.

A term with a typing of the form $x_1 : A_1, \ldots, x_n : A_n \ \triangleright M : C$ is translated to a relation R of type $\mathbb{F}A_1^\circ \times \ldots \times \mathbb{F}A_n^\circ \leftrightarrow C^\circ$. We denote such relations using

the notation $\bar{t} \mid \psi \mid u$ where $\bar{t}$ is a tuple of terms of type $\mathbb{F}A_1^\circ \times \ldots \times \mathbb{F}A_n^\circ$, u a term of type C° and ψ a formula. This is meant to denote a relation R defined by

$$R(x_1, \ldots, x_n, u) \leftarrow x_1 = t_1, \ldots, x_n = t_n, \psi$$

The translation will be defined by induction on type derivations. For each type rule, we give a translation rule in terms of relation triples. Using these one can construct a derivation tree of relations, each relation corresponding to a PCF term in the type derivation. Each use of the **mkgraph** primitive generates a recursive predicate definition. The collection of such predicate definitions forms a logic program and the relation triple corresponding to the overall term becomes the query.

The translations for the first three rules (called the structural rules of Exchange, Contraction and Weakening) are straightforward:

$$\frac{\Gamma, x : A, y : B, \Gamma' \;\rhd M : C}{\Gamma, y : B, x : A, \Gamma' \;\rhd M : C} \quad \frac{\bar{t}, u_1, u_2, \bar{t}' \mid \psi \mid v}{\bar{t}, u_2, u_1, \bar{t}' \mid \psi \mid v}$$

$$\frac{\Gamma, y_1 : A, y_2 : A \;\rhd M : C}{\Gamma, x : A \;\rhd M[x/y_1, x/y_2] : C} \quad \frac{\bar{t}, u_1, u_2 \mid \psi \mid v}{\bar{t}, u_1 \cup u_2 \mid \psi \mid v}$$

$$\frac{\Gamma \;\rhd M : C}{\Gamma, x : A \;\rhd M : C} \quad \frac{\bar{t} \mid \psi \mid v}{\bar{t}, \{\} \mid \psi \mid v}$$

The next two rules (called Identity and Cut) are translated as follows:

$$\frac{}{x : A \;\rhd x : A} \quad \frac{}{\{z\} \mid \mathit{true} \mid z}$$

$$\frac{\Gamma \vdash M : A \quad \Delta, x : A \vdash N : C}{\Gamma, \Delta \vdash N[M/x] : C} \quad \frac{\bar{t}_1 \mid \psi_1 \mid u \quad \bar{t}_2, u' \mid \psi_2 \mid v}{\bar{x}_1, \bar{t}_2 \mid \mathbf{mkgraph}[\bar{t}_1 | \psi_1 | u](\bar{x}_1, u'), \psi_2 \mid v}$$

The complexity of the Cut rule is due to the fact that the term u in the first premise is of type A° whereas u' in the second premise is of type $\mathbb{F}A^\circ$. Therefore, we must generate a graph of u's and match it to u'.

The primitives are translated as follows:

$$\frac{}{\rhd n : \mathit{int}} \quad \frac{}{\mid \mathit{true} \mid n}$$

$$\frac{\Gamma \;\rhd M : \mathit{int} \quad \Delta \;\rhd N : \mathit{int}}{\Gamma, \Delta \;\rhd M + N : \mathit{int}} \quad \frac{\bar{t}_1 \mid \psi_1 \mid u_1 \quad \bar{t}_2 \mid \psi_2 \mid u_2}{\bar{t}_1, \bar{t}_2 \mid \psi_1, \psi_2, \mathit{add}(u_1, u_2, z) \mid z}$$

$$\frac{\Gamma \;\rhd N_1 : A \quad \Gamma \;\rhd N_2 : A}{\Gamma, x : \mathit{int} \;\rhd \mathbf{if}\; x\; N_1\; N_2 : A} \quad \frac{\bar{t}_1 \mid \psi_1 \mid v_1 \quad \bar{t}_2 \mid \psi_2 \mid v_2}{\bar{x}, \{x\} \left| \begin{array}{l} (x = 0, \bar{x} = \bar{t}_1, \psi_1, y = v_1) \vee \\ (x \neq 0, \bar{x} = \bar{t}_2, \psi_2, y = v_2) \end{array} \right| y}$$

Notice that the translation of the conditional gives rise to a disjunction. In a concurrent logic programming language, one must treat $x = 0$ and $x \neq 0$ as guards of the two branches.

Finally, the higher-order terms are translated by:

$$\frac{\Gamma, x : A \rhd M : B}{\Gamma \rhd \lambda x.\, M : A \to B} \qquad \frac{\bar{t}, u \mid \psi \mid v}{\bar{t} \mid \psi \mid (u, v)}$$

$$\frac{\Gamma \rhd M : A \to B}{\Gamma, x : A \rhd Mx : B} \qquad \frac{\bar{t} \mid \psi \mid u}{\bar{t}, X \mid \psi, u = (X, z) \mid z}$$

$$\frac{}{x : A \to A \rhd \mathrm{rec}\, x : A} \qquad \frac{}{F \mid \mathrm{recurse}(F, z) \mid z}$$

Notice that abstraction is pair formation and application is pair decomposition. The *recurse* predicate used in the last rule is defined as follows:

$recurse : \mathbb{F}(\mathbb{F}A \times A) \leftrightarrow A$
recurse(F, x) $\leftarrow$ F = {(X, x)} $\cup$ F', **mkgraph**[F | recurse(F, x) | x](F', X)

As an example of the translation, consider the *succ* function:

$$\frac{\dfrac{\dfrac{}{x : int \rhd x : int} \quad \dfrac{}{\rhd 1 : int}}{x : int \rhd x + 1 : int}}{\rhd \lambda x.\, x + 1 : int} \qquad \frac{\dfrac{\dfrac{}{\{x\} \mid true \mid x} \quad \dfrac{}{\mid true \mid 1}}{\{x\} \mid add(x, 1, y) \mid y}}{\mid add(x, 1, y) \mid (\{x\}, y)}$$

The translation triple of this term denotes a relation $succ : \mathbb{F}int \leftrightarrow int$ defined by:

succ(({x}, y)) $\leftarrow$ add(x, 1, y)

This differs from the translation given at the beginning of Sec. 3 in that there we did not model the input of *succ* as of type $\mathbb{F}int$. The translation given here treats all types uniformly. But, *int* being a pure data type with no higher-order values, the use of $\mathbb{F}$ can be dropped. This and other optimizations are discussed in Sec. 3.3 below.

We show the correctness of the translation as follows:

Lemma 2 (type soundness). *The translation of a PCF term* $x_1 : A_1, \ldots, x_n : A_n \rhd M : C$ *is a relation of type* $\mathbb{F}A_1^\circ \times \ldots \times \mathbb{F}A_n^\circ \leftrightarrow C^\circ$.

Theorem 3. *The following equivalences of lambda terms are preserved by the translation:*

$$(\lambda x.\, M)\, N \equiv M[N/x]$$
$$\lambda x.\, Mx \equiv M$$
$$\mathrm{rec}\, M \equiv M\, (\mathrm{rec}\, M)$$

Proof. For $(\lambda x.\, M)\, N$ we have a translation of the following form:

$$\frac{\Gamma \rhd N : A \qquad \dfrac{\dfrac{\Delta, x : A \rhd M : B}{\Delta \rhd \lambda x.\, M : A \to B}}{\Delta, y : A \rhd (\lambda x.\, M) y : B}}{\Gamma, \Delta \rhd (\lambda x.\, M) N : B} \qquad \frac{\bar{t}' \mid \psi' \mid u' \qquad \dfrac{\dfrac{\bar{t}, u \mid \psi \mid v}{\bar{t} \mid \psi \mid (u, v)}}{\bar{t}, X \mid \psi, (u, v) = (X, z) \mid z}}{\bar{x}', \bar{t} \mid \mathbf{mkgraph}[\bar{t}' \mid \psi' \mid u'](\bar{x}', X), \psi, (u, v) = (X, z) \mid z}$$

By unifying (u, v) with (X, z) and substituting for X and z, we obtain the translation of $M[N/x]$:

$$\frac{\varGamma \vdash N : A \quad \varDelta, x : A \vdash M : C}{\varGamma, \varDelta \vdash M[N/x] : C} \quad \frac{\bar{t}' \mid \psi' \mid u' \quad \bar{t}, u \mid \psi \mid v}{\bar{x}', \bar{t} \mid \mathbf{mkgraph}[\bar{t}' | \psi' | u'](\bar{x}', u), \psi \mid v}$$

The preservation of the *eta* equivalence follows similarly from the properties of unification and the recursion equivalence follows from the definition of the *recurse* predicate. □

3.2 Data structures

The translation can be readily extended to deal with data structures. Product and sum types of a functional language are translated as follows:

$$\begin{aligned} \mathit{unit}^\circ &= \emptyset \\ (A \times B)^\circ &= \mathrm{fst}(A^\circ) + \mathrm{snd}(B^\circ) \\ (c(A) + d(B))^\circ &= c(\mathbb{F} A^\circ) + d(\mathbb{F} B^\circ) \end{aligned}$$

In the case of product types, *fst* and *snd* are assumed to be constructor symbols not used elsewhere. Notice that the translation of product types exhibits the same kind of reduction in the cardinality as encountered with function types previously. Think of a pair-typed value as a process that responds to *fst* and *snd* messages.

The translations of terms can be easily derived using the translation of types as a heuristic. For binary products:

$$\frac{\varGamma \rhd M : A \quad \varGamma \rhd N : B}{\varGamma \rhd (M, N) : A \times B} \quad \frac{\bar{t}_1 \mid \psi_1 \mid u_1 \quad \bar{t}_2 \mid \psi_2 \mid u_2}{\bar{x} \left| \begin{array}{l} (y = \mathrm{fst}(u_1), \bar{x} = \bar{t}_1, \psi_1) \vee \\ (y = \mathrm{snd}(u_2), \bar{x} = \bar{t}_2, \psi_2) \end{array} \right| y}$$

$$\frac{}{x : A \times B \rhd \mathrm{fst}\ x : A} \quad \frac{}{\{\mathrm{fst}(z)\} \mid \mathit{true} \mid z}$$

The relation for the pair has two clauses. One of them will get exercised when one of the components is selected. Similarly, the translation of terms of sum types is:

$$\frac{\varGamma \rhd M : A}{\varGamma \rhd c(M) : c(A) + d(B)} \quad \frac{\bar{t} \mid \psi \mid u}{\bar{t} \mid \psi \mid c(u)}$$

$$\frac{\varGamma, x_1 : A \rhd M : C \quad \varGamma, x_2 : B \rhd N : C}{\varGamma, x : c(A) + d(B) \rhd \mathbf{case} x \mathbf{of} c(x_1) \Rightarrow M \mid d(x_2) \Rightarrow N : C}$$

$$\frac{\bar{t}_1, t_1 \mid \psi_1 \mid u_1 \quad \bar{t}_2, t_2 \mid \psi_2 \mid u_2}{\bar{x}, \{x\} \left| \begin{array}{l} (x = c(t_1), \bar{x} = \bar{t}_1, z = u_1) \vee \\ (x = d(t_2), \bar{x} = \bar{t}_2, z = u_2) \end{array} \right| z}$$

Many data structures occurring in practice take the form of sum-of-products. For example, lists are given by a type definition of the form:

type list = nil(unit) + cons(int × list)

In translating such a type to the logic programming context, we can make use of the isomorphisms:

$$\mathbb{F}(A + B) \cong \mathbb{F}A \times \mathbb{F}B \qquad \mathbb{F}\emptyset \cong \mathit{unit}$$

So, the translation of the list type can be given as:

list$^\circ$ = nil(unit) + cons($\mathbb{F}$ int × $\mathbb{F}$ list$^\circ$)

3.3 Optimizations

The type $\mathbb{F}\mathit{int}$ denotes finite sets of integers. Since functional programs give determinate results, such a set contains at most one integer. The empty set, $\{\}$, appears when a program ignores its integer input. A singleton set appears when the program uses the input. The use of finite sets for data values thus allows lazy data structures. The above type list° makes available lazy lists in a concurrent logic program.

If we are not interested in lazy data structures, we can eliminate all uses of $\mathbb{F}$ in data structures. To model this, we use the following translation:

$$\begin{aligned} \mathit{int}^* &= \mathit{int} \\ \mathit{unit}^* &= \mathit{unit} \\ (A \times B)^* &= A^* \times B^* \\ (c(A) + d(B))^* &= c(A^*) + d(B^*) \end{aligned}$$

For example,

list* = nil(unit) + cons(int × list*)

The function type $A \to B$ is then translated to $A^* \times B^\circ$ instead of $\mathbb{F}A^\circ \times B^\circ$. We will see applications of this translation in the next section.

Another opportunity for optimization occurs when the argument of a function is used *linearly*, i.e., if the function uses its argument precisely once. In this case, we can simplify the translation of functions to

$$(A \to B)^\circ \;=\; A^\circ \times B^\circ$$

This optimization is already illustrated in the examples at the beginning of Sec. 3.

3.4 Linear logic

The reader familiar with linear logic would notice that our translation closely parallels Girard's embedding of intuitionistic logic in linear logic [Gir87]. Indeed, our debt to linear logic is considerable. Logic programming (more precisely, the category of sets and relations) forms a model of linear logic: $\&$ and $\oplus$ are interpreted as disjoint union, $\otimes$ and $\multimap$ are interpreted as product, and ! is interpreted as finite sets. The translation given above is essentially a restatement of Girard's translation in this new light.

4 Example applications

The results of Sec. 3 establish a close correspondence between the higher-order programming techniques of functional programming and those of logic programming. Two features of logic programming play a role in this correspondence:

- logic variables, and
- concurrency.

The higher-order programming techniques are available only to a limited extent in sequential logic programming due to the crucial role of concurrency in this correspondence. However, they are still available and logic programmers frequently make use of them. In this section, we draw a correspondence between some of the common programming techniques used in functional and logic programming paradigms in the light of our translation. We hope that this will lead to a reexamination of the logic programming techniques in the light of "higher-order programming" and pave the way for further exchange of ideas and techniques between the two paradigms.

Difference lists form a familiar logic programming technique which allow for constant time concatenation [CT77]. A very similar technique was developed in functional programming using function values [Hug86]. Our correspondence identifies the two techniques, i.e., if we translate the functional programming representation of difference lists using the translation of Sec. 3, we obtain the logic programming representation.

The functional programming technique is shown in Table 2. A difference list is represented as a function from lists to lists. The representation of an ordinary list $[x_1, \ldots, x_n]$ is the function $\lambda l.\, x_1.x_2.\ldots.x_n.l$. (See the function *rep*.) Concatenation is then just function composition, which works in constant time.

type diff_list = list → list
rep : list → diff_list
rep [ ] = λl. l
rep x.xs = λl. x.((rep xs) l)
concat : diff_list → diff_list → diff_list
concat x y = λl. x (y l)

Table 2. Difference lists in functional programming

The logic programming technique is shown in Table 3. Here, a difference list is a pair of lists, but one of the lists is typically a logic variable which is shared in the other list. The representation of a list $[x_1, \ldots, x_n]$ is a pair $(x_1.x_2.\ldots.x_n.l, l)$ where l is a variable.

To see the correspondence between the two versions, notice that the functional difference list uses its list argument linearly. Thus, as noted in Sec. 3.3,

type diff_list = list × list
rep : list ↔ diff_list
rep([ ], (l',l)) ← l = l'
rep(x.xs, (l',l)) ← rep(xs, (m, l)), l' = x.m
concat : diff_list × diff_list ↔ diff_list
concat(x, y, (l', l)) ← y = (m, l), x = (l', m)

Table 3. Difference lists in logic programming

list → *list* can be translated to *list* × *list* in the logic programming context. (The second component corresponds to the input list and the first component to the output list.) A close examination of the two programs is quite striking. They correspond operator by operator according to the translation of Sec. 3.

A difference structure, in general, involves some data or computational structure with a placed holder inside the structure captured by a logic variable. That is essentially what a function is. It is a computation with a place holder for the argument. Thus, the example we found here is an instance of a general concept.

Difference lists of Table 3 can only be used linearly. Our technique also allows us to build difference lists that can be used multiple times. For example,

mkrep : list ↔ $\mathbb{F}$ *diff_list*
mkrep(l, [ ]) ←
mkrep(l, d.D) ← rep(l, d), mkrep(l, D)

allows a copy of a difference list to be produced on demand. To check if a difference list is empty, we can then use a goal of the form:

D = d.D', d = (l, [ ]), l = [ ]

This kind of a goal has the problem that it makes an entire copy of a difference list to check if it is empty. We can optimize it by using lazy lists instead of standard lists. (Cf. Sec. 3.3).

Another important application for function values is in abstract syntax. Consider a language that has variable-binding operators, e.g., lambda calculus or predicate calculus. In writing a processor for such a language (evaluator, theorem prover *etc.*), there arise delicate issues of variable renaming, substitution *etc.* Pfenning and Elliott [PE88] proposed that such issues can be treated in modular fashion by using higher-order abstract syntax, i.e., syntactic representations that involve function values. To see how this works, consider the first-order and higher-order representations for lambda terms:

term = var(symbol) + lam(symbol × term) + ap(term × term)
hoterm = lam(hoterm → hoterm) + ap(hoterm × hoterm)

The higher-order representation (*hoterm*) is symbol-free. The variable binding operator, *lam*, takes a function mapping terms (the arguments) to terms (the

result of embedding the argument in the body of the abstraction). For example, the Church numeral $\mathbf{2} = \lambda s.\, \lambda x.\, s(sx)$, has the following representations:

first-order: lam(“s”, lam(“x”, ap(var(“s”), ap(var(“s”), var(“x”)))))
higher-order: lam(λt. lam(λu. ap(t, ap(t, u))))

Table 4 shows a parser that converts a first-order representation to the higher-order one and a normalizer that operates on higher-order representations. Notice that substitution is handled cleanly in the definition of *apply lam*(f) n. The result of substituting n for the formal parameter is simply fn. In effect, all substitution is performed at parse-time with the result that processing of the language is considerably simplified.

---

type term = var(symbol) + lam(symbol × term) + ap(term × term)
type hoterm = lam(hoterm → hoterm) + ap(hoterm × hoterm) + fvar(hoterm)
type subst = symbol → hoterm
lookup : symbol → subst → hoterm
lookup s e = e s
update : symbol → hoterm → subst → subst
update s x e = λs’. if s’ = s then x else e s’

parse : term → subst → hoterm
parse var(s) e = lookup s e
parse lam(s, m) e = lam(λx. parse m (update s x e))
parse ap(m, n) e = ap(parse m e, parse n e)

norm : hoterm → hoterm
reduce : hoterm → hoterm
norm m = reduce (wkreduce m)
reduce lam(f) = lam(λx. norm (f fvar(x)))
reduce m = m *for other cases of m*

wkreduce : hoterm → hoterm
apply : hoterm × hoterm → hoterm
wkreduce lam(f) = lam(f)
wkreduce ap(m, n) = apply (wkreduce m) n
wkreduce fvar(t) = fvar(t)
apply lam(f) n = wkreduce (f n)
apply m n = ap(m, n) *for other cases of m*

Table 4. Higher-order abstract syntax in functional programming

Precisely the same effect can be achieved in logic programming. One can replace all the occurrences of symbols by logic variables so that substitution is achieved by simply binding the logic variables. This technique has been used, for instance, in the polymorphic type inference algorithm of Typed Prolog [LR91, Lak91]. We would like to argue that this is not merely achieving the same effect as

higher-order abstract syntax, but it is in fact the *same* technique. It is an instance of our translation. However, the standard technique used in logic programming is not as general as the higher-order technique. In particular, it does not work if the terms substituted for variables have bound variables. In that case, one has to rename bound variables with fresh logical variables and the cleanliness of the technique is compromised. On the other hand, our translation handles the general case as well. The naive logic programming technique is the special case where inputs of functions are assumed to be data objects with no embedded functions.

In Table 5, we show the naive logic programming technique for a normalizer of lambda terms. Notice that lambda abstractions are represented by terms of type $lam(hoterm \times hoterm)$. When an abstraction $\lambda s.\, m_s$ is converted to this representation, a fresh logical variable x is created and all occurrences of the symbol s are replaced with x. See the clause for $parse(lam(s, m), e, t)$. The resulting representation is $lam(x, m_x)$. To reduce an application of such an abstraction term, in the clause $apply(lam(f), n, t)$, we bind x to the argument term n and reduce m_n.

type term = var(symbol) + lam(symbol × term) + ap(term × term)
type hoterm = lam(hoterm × hoterm) + ap(hoterm × hoterm) + fvar(symbol)

parse : term × subst ↔ hoterm
parse(var(s), e, t) ← lookup(s, e, t)
parse(lam(s, m), e, t) ← update(s, x, e, e'), parse(m, e', t'), t = lam((x, t'))
parse(ap(m, n), e, t) ← parse(m, e, t1), parse(n, e, t2), t = ap(t1, t2)

norm : hoterm ↔ hoterm
reduce : hoterm ↔ hoterm
norm(m, t) ← wkreduce(m, t'), reduce(t', t)
reduce(lam(f), t) ← f = (fvar(x), m), norm(m, t'), t = lam((x, t'))
reduce(m, t) ← t = m   *for other cases of m*

wkreduce : hoterm ↔ hoterm
apply : hoterm × hoterm ↔ hoterm
wkreduce(lam(f), t) ← t = lam(f)
wkreduce(ap(m, n), t) ← wkreduce(m, t'), apply(t', n, t)
wkreduce(fvar(x), t) ← t = fvar(x)
apply(lam(f), n, t) ← f = (n, m), wkreduce(m, t)
apply(m, n, t) ← t = ap(m, n)   *for other cases of m*

Table 5. A naive logic program for higher-order abstract syntax

The point is that the logic programming representation $lam(hoterm \times hoterm)$ is a naive encoding of the functional representation. It works in the following cases:

- every m_x contains precisely one occurrence of x, *i.e.*, lambda terms are linear, or
- the argument term has no bound variables, *i.e.*, argument terms are pure data.

To handle the general case, we must use the general translation scheme outlined in Sec. 3. The result is shown in Table 6. We have used the optimization of treating *term* and *symbol* as pure data. Moreover, the arguments of *norm*, *reduce* and *wkreduce* and the first argument of *apply* are treated as linear inputs. These optimizations give considerable simplifications, but the reader might still find the resulting program hard to comprehend. One would want to devise usable syntactic mechanisms to cut down this complexity. (The focus of this paper being the expressive power of pure logic programming, we have refrained from doing so.)

In this program, *mkparse* is used to generate multiple *hoterm*'s for any given first-order *term*. Each *hoterm* receives a fresh set of logic variables to represent its bound variables. (See *parselam*.) No extra mechanism for variable renaming is necessary.

Many other programming techniques of higher-order functional programming can be similarly adapted to the logic programming context. See [BW88, Pau91, Wad92]. The reader is invited to try some of these using the translation scheme of Section 3.

5 Conclusion

We have demonstrated the expressive power of logic programming by modelling higher-order programming features in pure logic programs. Further, these programming techniques, in a limited context, correspond to well-known techniques of logic programs.

However, the significant abstraction facilities made possible by higher-order techniques are not available in sequential logic programming. Thus, we believe that sequential logic programming is a poor approximation to logic programming and that efforts must be made to incorporate concurrency and fairness. Concurrent logic programming languages as well as coroutining facilities [Nai86] are an important step in this direction. These systems need to be extended to deal with backtracking nondeterminism. Formal semantics and type systems must be developed to place them on a firm foundation.

The correspondences drawn in this work are only a first step in exploiting the richness of the logic programming paradigm. These techniques must be further explored and applied to practical contexts.

References

[Abr91] S. Abramsky. Computational interpretation of linear logic. Tutorial Notes, International Logic Programming Symposium, San Diego, 1991, 1991.

type term = var(symbol) + lam(symbol $\times$ term) + ap(term $\times$ term)
type hoterm = lam($\mathbb{F}$($\mathbb{F}$ hoterm $\times$ hoterm)) + ap($\mathbb{F}$ hoterm $\times$ $\mathbb{F}$ hoterm)
+ fvar($\mathbb{F}$ hoterm)
type subst = symbol $\times$ hoterm
lookup : symbol $\times$ $\mathbb{F}$ *subst* $\leftrightarrow$ *hoterm*
lookup(s, E, t) $\leftarrow$ E = {(s, t)}
update : symbol $\times$ $\mathbb{F}$ *hoterm* $\times$ $\mathbb{F}$ *subst* $\leftrightarrow$ *subst*
update(s, X, E, (s', x')) $\leftarrow$ s' = s, X = {x'}, E = {}
update(s, X, E, (s', x')) $\leftarrow$ s' $\neq$ s, X = {}, E = {(s', x')}

parse : term $\times$ $\mathbb{F}$ *subst* $\leftrightarrow$ *hoterm*
parselam : symbol $\times$ *term* $\times$ $\mathbb{F}$ *subst* $\leftrightarrow$ *hoterm*
parse(var(s), E, t) $\leftarrow$ lookup(s, E, t)
parse(lam(s, m), E, t) $\leftarrow$ mkparselam(s, m, E, F), t = lam(F)
parse(ap(m, n), E, t) $\leftarrow$ E = E1 $\cup$ E2,
mkparse(m, E1, T1), mkparse(n, E2, T2), t = ap(T1, T2)
parselam(s, m, E, f) $\leftarrow$ mkupdate(s, X, E, E'), parse(m, E', t'), f = (X, t')

norm : hoterm $\leftrightarrow$ *hoterm*
reduce : hoterm $\leftrightarrow$ *hoterm*
norm(m, t) $\leftarrow$ wkreduce(m, t'), reduce(t', t)
reduce(lam(F), t) $\leftarrow$ mkfvar(X, TX), F = {(TX, m)}, norm(m, t'), t = lam((X, t'))
reduce(m, t) $\leftarrow$ t = m *for other cases of m*

wkreduce : hoterm $\leftrightarrow$ *hoterm*
apply : hoterm $\times$ $\mathbb{F}$ *hoterm* $\leftrightarrow$ *hoterm*
wkreduce(lam(F), t) $\leftarrow$ t = lam(F)
wkreduce(ap(M, N), t) $\leftarrow$ M = {m}, wkreduce(m, t'), apply(t', N, t)
wkreduce(fvar(X), t) $\leftarrow$ t = fvar(X)
apply(lam(F), N, t) $\leftarrow$ F = {(N, m)}, wkreduce(m, t)
apply(m, N, t) $\leftarrow$ t = ap({m}, N) *for other cases of m*

[*mkfvar, mkupdate, mkparse, and mkparselam are instances of* **mkgraph**.]

Table 6. A general logic program for higher-order abstract syntax

[Abr93] S. Abramsky. Computational interpretations of linear logic. *Theoretical Comp. Science*, 111(1-2):3–57, 1993.

[Bar91] M. Barr. *-Autonomous categories and linear logic. *Math. Structures in Comp. Science*, 1:159–178, 1991.

[BW88] R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice-Hall International, London, 1988.

[CKW89] W. Chen, M. Kifer, and D. S. Warren. HiLog: A first-order semantics for higher-order logic programming constructs. In L. Lusk, E and R. A. Overbeek, editors, *Logic Programming: Proc. of the North American Conf. 1989*, pages 1090–1144. MIT Press, 1989.

[CT77] K. L. Clark and S. A. Tarnlund. A first-order theory of data and programs. In *Information Processing*, pages 939–944. North-Holland, 1977.

[Gir87] J.-Y. Girard. Linear logic. *Theoretical Comp. Science*, 50:1–102, 1987.

[Gre87] S. Gregory. *Parallel Logic Programming in PARLOG: The Language and its Implementation*. Addison-Wesley, Reading, Mass., 1987.

[Gun92] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. MIT Press, 1992.

[Hug86] R. J. M. Hughes. A novel representation of lists and its application to the function "reverse". *Information Processing Letters*, 22:141–144, Mar 1986.

[Laf87] Y. Lafont. Linear logic programming. In P. Dybjer, editor, *Porc. Workshop on Programming Logic*, pages 209–220. Univ. of Goteborg and Chalmers Univ. Technology, Goteborg, Sweden, Oct 1987.

[Lak91] T. K. Lakshman. Typed prolog: Type checking/type reconstruction system (version 1.0). Software available by anonymous FTP from cs.uiuc.edu, 1991.

[LM84] J.-L. Lassez and M. J. Maher. Closures and fairness in the semantics of programming logic. *Theoretical Computer Science*, pages 167–184, May 1984.

[LR91] T.K. Lakshman and U. S. Reddy. Typed Prolog: A Semantic Reconstruction of the Mycroft-O'Keefe Type System. In V. Saraswat and K. Ueda, editors, *Logic Programming: Proceedings of the 1991 International Symposium*, pages 202 – 217. MIT Press, Cambridge, Mass., 1991.

[Mil90] R. Milner. Functions as processes. In *Proceedings of ICALP 90*, volume 443 of *Lect. Notes in Comp. Science*, pages 167–180. Springer-Verlag, 1990.

[MN86] D. A. Miller and G. Nadathur. Higher-order logic programming. In *Intern. Conf. on Logic Programming*, 1986.

[Nai86] Lee Naish. *Negation and control in Prolog*, volume 238 of *Lect. Notes in Comp. Science*. Springer-Verlag, New York, 1986.

[Pau91] L. C. Paulson. *ML for the Working Programmer*. Cambridge Univ. Press, Cambridge, 1991.

[PE88] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *ACM SIGPLAN '88 Conf. Program. Lang. Design and Impl.*, pages 22–24. ACM, 1988.

[Plo77] G. D. Plotkin. LCF considered as a programming language. *Theoretical Comp. Science*, 5:223–255, 1977.

[Red93a] U. S. Reddy. Higher-order functions and state-manipulation in logic programming. In R. Dyckhoff, editor, *Fourth Workshop on Extensions of Logic Programming*, pages 115–126, St. Andrews, Scotland, Mar 1993. St. Andrews University.

[Red93b] U. S. Reddy. A typed foundation for directional logic programming. In E. Lamma and P. Mello, editors, *Extensions of Logic Programming*, volume 660 of *Lect. Notes in Artificial Intelligence*, pages 282–318. Springer-Verlag, 1993.

[Sar92] Vijay Saraswat. The category of constraint systems is Cartesian-closed. In *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 341–345, Santa Cruz, California, 22–25 June 1992. IEEE Computer Society Press.

[Sco69] D. S. Scott. A type theoretical alternative to CUCH, ISWIM and OWHY. Unpublished manuscript, Oxford University, 1969.

[Sha83] E. Y. Shapiro. A subset of Concurrent Prolog and its interpreter. Technical Report TR-003, ICOT- Institute of New Generation Computer Technology, January 1983. (Reprinted in [Sha87].).

[Sha87] E. Shapiro. *Concurrent Prolog: Collected Papers*. MIT Press, 1987. (Two volumes).

[ST86] E. Shapiro and A. Takeuchi. Object-oriented programming in Concurrent Prolog. *New Generation Computing*, 4(2):25–49, 1986. (reprinted in [Sha87].).

[Ued86] K. Ueda. Guarded Horn clauses. In E. Wada, editor, *Logic Programming*, pages 168–179. Springer-Verlag, 1986. (reprinted in [Sha87].).

[Wad92] P. Wadler. The essence of functional programming. In *ACM Symp. on Princ. of Program. Lang.*, 1992.

[War82] D. H. D. Warren. Higher-order extensions to Prolog: Are they needed? In D. Michie, editor, *Machine Intelligence, 10*, pages 441–454. Edinburgh University Press, 1982.

This article was processed using the LaTeX macro package with LLNCS style