A Typed Foundation for Directional Logic Programming

Uday S. Reddy

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

Abstract. A long standing problem in logic programming is how to impose *directionality* on programs in a safe fashion. The benefits of directionality include freedom from explicit sequential control, the ability to reason about algorithmic properties of programs (such as termination, complexity and deadlock-freedom) and controlling concurrency. By using Girard's linear logic, we are able to devise a type system that combines types and modes into a unified framework, and enables one to express directionality declaratively. The rich power of the type system allows outputs to be embedded in inputs and *vice versa*. Type checking guarantees that values have unique producers, but multiple consumers are still possible. From a theoretical point of view, this work provides a "logic programming interpretation" of (the proofs of) linear logic, adding to the concurrency and functional programming interpretations that are already known. It also brings logic programming into the broader world of typed languages and types-as-propositions paradigm, enriching it with static scoping and higher-order features.

Keywords Directionality, modes, types, concurrent logic programming, linear logic, Curry-Howard isomorphism, sequent calculus, logic variables.

1 Introduction

Logic programming was heralded with the slogan [29]:

Algorithm = Logic + Control

While the "logic" part of the algorithm, formalized by declarative semantics, has been well-studied (see, *e.g.*, [35]), procedural semantics—the semantics of control—has not progressed far. Since the theory offers only certain unreachable mechanisms for complete strategies [33], practical logic programming languages have resorted to low-level operational mechanisms that are not entirely satisfactory. The sequential control of Prolog [9] and the read-only variables of Concurrent Prolog [49] are examples of such mechanisms. At the same time, the desire for higher-level *declarative* mechanisms for specifying control has been certainly present, as seen in the mode declarations of Parlog [21] and MU-Prolog [41].

We believe that a formal approach to procedural semantics is necessary so that one can reason about algorithmic properties of programs such as termination, complexity and deadlock freedom *etc.* To address such questions one needs to specify not only the "logic" of an algorithm but also how the logic is to be used, *i.e.*, what kind of parameters we intend to supply to a predicate.

We propose the concept of a *procedure* as the foundation for algorithmically sound logic programming. A procedure is like a predicate, but has a specific input-output directionality. It is not necessary for an entire argument of a predicate to be an input or output. Outputs can be embedded in input arguments and inputs can be embedded in output arguments. Thus, the interesting applications of logic variables such as partially instantiated data structures and back communication in concurrent programs are still possible. What is required is a certain well-formedness condition: If a value is given as an input to a procedure, there must be another procedure (in fact, precisely one procedure) which produces the value as an output. We call logic programming via such procedures *directional logic programming.*[1] Conventional logic programs can then be thought of as short hand notations which stand for multiple directional logic programs.

Historically, this work was begun in [42, 43] where a "simple theory of directionality" was outlined. The qualifier "simple" signifies the fact that entire arguments of predicates are treated as inputs or outputs. (Undoubtedly, such a simple theory is not adequate to handle all logic programs of interest). Ideas similar to this proposal were used in Parlog [8, 21] and MU-Prolog [41] for controlling evaluation order, and by Deville [13] for reasoning about algorithmic properties of logic programs. However, the subset of programs treated by the simple modes is too weak, and all these applications found need for extensions which, unfortunately, lacked the conviction of the simple modes. The author had been of the opinion that "combining modes with types" would provide a way to obtain a richer theory of directionality. But, this project had never been carried to its conclusion. With Girard's invention of *linear logic* [17], all the technical tools necessary for the synthesis of types and modes were suddenly found to be in place. This paper reports on the application of linear logic to the problem of directionality.

Linear logic was formulated by Girard as a constructive logic that is "larger" than intuitionistic logic. In particular, it incorporates features of classical logic such as the law of negation $\neg\neg A = A$ and a version of the excluded middle: $A \vee \neg A$. This feat is accomplished by controlling the use of *structural rules*, weakening and contraction, which would otherwise allow a hypothesis to be used an arbitrary number of times (as in classical or intuitionistic logic). In fact, Girard notes that intuitionistic logic itself owes its constructive nature to restrictions on the structural rules. By making these restrictions uniform, we obtain a constructive logic that is able to go beyond the intuitionistic one.

[1] The concept of procedures is, in fact, present in most concurrent logic programming languages including Concurrent Prolog [49], Parlog [21], Janus [45] and other concurrent constraint languages [46]. Our notion of directional logic programming includes them and provides a *theory* for their well-formedness.

The proofs in a constructive logic are recipes for computation, *i.e.,* they are programs. For instance, functional programs are textual representations of the proofs of intuitionistic logic. (See [2, 10, 20, 37, 52]). The proofs of linear logic must similarly have a computational interpretation as programs. What kind of programs? It seems that the symmetries of linear logic permit a wide spectrum of interpretations. Girard himself formulated a concurrent computation framework called proof nets [17, 19]. Lafont [31] uses similar structures called "interaction nets". However, the abstract nature of this framework does not readily lend itself to use as a programming language. Abramsky [2] was the first to propose a concrete computational interpretation as a concurrent programming language in the framework of the "chemical abstract machine" [5]. Filinski [14] and Reddy [44] proposed functional programming interpretations in terms of continuations and "acceptors" respectively. It seems that the asymmetry between inputs and outputs present in functional programming does not fit well with linear logic. The present work forms yet another interpretation of the proofs of linear logic: as directional logic programs. Not only do logic programs avoid the asymmetry of functional programs, but many of their features, such as Horn clause notation, find a ready correspondence with linear logic concepts.

Curry-Howard isomorphism In interpreting this work, one must separate in one's mind two distinct ways of relating programming languages and logic. In the *declarative programming* tradition, one has the correspondence:

$$\begin{aligned} \text{programs} &\longleftrightarrow \text{propositions} \\ \text{computations} &\longleftrightarrow \text{proofs} \end{aligned}$$

This is sometimes called "proofs as computations" correspondence. There is a large body of work in using this correspondence to interpret linear logic for declarative programming (including logic programming [3, 24, 26, 30] and concurrent systems [4, 6, 15, 22, 36]). On the other hand, as outlined in the previous paragraph, a deeper correspondence exists with *constructive* logics:

$$\begin{aligned} \text{types} &\longleftrightarrow \text{propositions} \\ \text{programs} &\longleftrightarrow \text{proofs} \\ \text{computation} &\longleftrightarrow \text{proof normalization} \end{aligned}$$

This correspondence is often called "propositions as types" correspondence or Curry-Howard isomorphism (following the work of [11, 27]). Here, programs are treated as purely recipes for computation and their potential declarative reading remains in the background. To emphasize this aspect—and to prevent confusion between the two correspondences—we will avoid the terminology of "predicate" and "goal formula" and use "procedure" and "command" in their place. The role of propositions in the programming context is played not by programs but by their types. To see why this is so, think of the type of a program as a weak form of specification. If p is a constructive proof of a proposition A, then the recipe for computation involved in p satisfies the specification corresponding to A. In fact, types need not really be weak specifications. By enriching types to include

equality propositions and dependent type constructors, one obtains a rich type system that can express arbitrary specifications [10, 37, 52]. We will say no more about this important topic. The reader may refer to the above references and [20] for further discussion of the Curry-Howard correspondence.

Directionality It is well-known that the selection of literals for resolution in logic programming crucially determines the search space. (See [43, 50] for a discussion). Take, for example, the list reversal program:

```
reverse(nil, nil).
reverse(a.x, y) ← reverse(x,z), append(z, a.nil, y).
```

and the goal `reverse(`L`, x)` where L is a specific list. Using left-to-right literal selection, the evaluation of the goal terminates in $O(n)$ time with one solution. But, using right-to-left literal selection, the same goal takes $O(n^2)$ time for the first solution and fails to terminate upon backtracking. These effects are roughly reversed for the goal `reverse(x, `L`)`. Therefore, logic programs are well-behaved for only certain sets of goals. We want to be able to specify the acceptable sets of goals and use such specifications for literal selection.

Suppose we specify input-output modes such as

```
mode reverse(in, out).
mode append(in, in, out).
```

Using a "strong" interpretation of modes as in [43], the acceptable goals are those which have ground terms in the input argument positions and variables in the output positions. Thus, `reverse(`L`, x)` is acceptable, but `reverse(x, `L`)` is not. A concurrent evaluation strategy can be devised where goals are suspended until the arguments in input positions have sufficient information for evaluation. The undesirable effects mentioned in the previous paragraph are thus eliminated.

However, specifying acceptable goals is not enough. We must also ensure that only acceptable goals are produced during evaluation. This involves checking that every variable in a clause appears in a single output position and cyclic waits do not arise. The "theory of directionality" of [43] and the "directionality check" of [21] are automatic methods of ensuring this.

The limitation of simple modes is that entire arguments are considered input or output. We cannot embed outputs in inputs and *vice versa*. For instance, we cannot use `reverse` to reverse a list of unbound variables. As argued by Shapiro [48] embedding outputs in inputs is crucial for concurrent applications. Parlog weakens its mode system to allow such embedding, but does not provide a way to check whether it is used consistently.

A little thought reveals that a solution for this problem lies in considering types. The type

```
reverse : Pred (List(t), List(t))
```

is polymorphic in the element type t [23, 32, 40], *i.e.*, `reverse` works uniformly for all types of list elements. If we combine mode information with types, this

polymorphism is enough to indicate that ground terms, unbound variables and all other kinds of mixtures are acceptable as list elements for `reverse`. In the linear-logic based type system proposed here, every type t has a dual type $t^\perp$ whose meaning is to switch the input-output notion of t. We can then specify the (directional) type of `reverse` as

```
reverse : proc (List(t), List(t)⊥)
```

This states that the first argument is an *input* of type `List(t)` and the second argument is an *output* of type `List(t)`. If we instantiate `t` to, say, `int`, we obtain the effect that a list of integers is reversed. On the other hand, if we instantiate it to $\texttt{int}^\perp$, we have the effect that a list of place holders for integers (such as unbound variables) is reversed.

Overview The rest of this paper is organized as follows. We first give a brief overview of linear logic in Section 2 and mention its meaning in terms of propositions as well types. In Section 3, we informally motivate the concepts of directional logic programming. A concrete logic programming language is presented in Section 4 together with its type rules and procedural semantics. We also give several examples to illustrate its use and the constraints of the type system. Its theoretical properties are mentioned in Section 5. Finally, in Section 6, we consider various extensions such as adding functions, nondeterminism *etc.*

2 Overview of linear logic

The syntax of propositions (or types) in linear logic is as follows. Let β range over atomic propositions (atomic type terms) and $A, B, \ldots$ over propositions. Then,[2]

$$\begin{array}{rll} A ::= & \beta \mid A^\perp & \\ & \mid A \otimes B \mid A \parallel B \mid 1 \mid \perp & \text{– multiplicatives} \\ & \mid A \,\&\, B \mid A \oplus B \mid \top \mid 0 & \text{– additives} \\ & \mid\, !A \mid ?A & \text{– exponentials} \end{array}$$

That is a lot of connectives for a propositional fragment of a logic! We will describe the meaning of these connectives (from a type-theoretic point of view) in the remainder of the paper. For now, we note that each classical connective has two versions in linear logic—a multiplicative one and an additive one. On each of these rows, the listed connectives correspond to $\wedge$, $\vee$, true and false of classical logic. The operator $(\ )^\perp$ corresponds to negation. The operators ! and ? are new. They indicate permission to carry out the structural rules of *weakening* (representing $A \Rightarrow$ true and false $\Rightarrow A$) and *contraction* ($A \Rightarrow A \wedge A$ and $A \vee A \Rightarrow A$) on the propositions.

[2] The type constructor "∥" is written as an upside down & in Girard's original notation. Our change of notation is due to typographical reasons.

The most important feature of linear logic is that negation is involutive (just as in classical logic), *i.e.*, $A^{\perp\perp} = A$.[3] Moreover, all other connectives satisfy the following De Morgan laws:

$$
\begin{array}{cc}
(A \otimes B)^\perp = A^\perp \parallel B^\perp & (A \parallel B)^\perp = A^\perp \otimes B^\perp \\
1^\perp = \perp & \perp^\perp = 1 \\
(A \,\&\, B)^\perp = A^\perp \oplus B^\perp & (A \oplus B)^\perp = A^\perp \,\&\, B^\perp \\
\top^\perp = 0 & 0^\perp = \top \\
(!A)^\perp = ?A^\perp & (?A)^\perp = !A^\perp
\end{array}
$$

This means that many of the expressible propositions are equal to each other. We obtain unique representations (negation normal forms) if we restrict negation to atomic propositions. Implication is defined in the classical fashion: $A \multimap B = A^\perp \parallel B$. So, $A^\perp$ can also be thought of as $A \multimap \perp$.

We are interested in the use of linear logic as a "constructive" logic, *i.e.*, in modelling programs as proofs of linear logic propositions. A linear logic proof from a collection of hypotheses Γ must use each of the hypotheses precisely once. This is called the *linearity* property. Now, consider, what is a proof of $A^\perp$? It should be a "proof method" that transforms proofs of A into contradictions. We can think of such a method as a "pure consumer" of proofs of A. How about a proof of $A^{\perp\perp}$? A method for transforming proofs of $A^\perp$ into contradictions must implicitly produce a proof of A and use it for contradicting $A^\perp$. So, it is effectively a proof of A. What we have just given is an informal argument that proofs of $A^{\perp\perp}$ and A are isomorphic. The requirement of linearity is crucial to this. Without it, we cannot be sure if the proof method for $A^{\perp\perp}$ is just ignoring $A^\perp$ or using it multiple times. In the first case, it would have produced no proof of A at all, and, in the second, it would have produced multiple proofs of A. Thus, by imposing the restriction of linearity, we obtain the all-important negation law: $A^{\perp\perp} = A$.

Let us switch gears and think of propositions as types and proofs as computations. If a computation of type A produces a value of type A, then a computation of type $A^\perp$ *consumes* values of type A. So, a computations of type $A^{\perp\perp}$ consumes consumers of A-typed values. As argued above, it effectively produces an A-typed value. So, the negation operator $(\ )^\perp$ means *inverting the input-output notion* of a type. An input of type A is an output of type $A^\perp$ and an output of type A is an input of type $A^\perp$. Thus, input-output modes are incorporated into types.

The best way to study linear logic proofs is by a sequent calculus [16, 18, 20, 51]. A derivation of a sequent $\Gamma \vdash \Delta$ represents a proof of (the disjunction of) the conclusions Δ from (the conjunction of) the hypotheses Γ. In a logic with involutive negation, it is not strictly necessary to treat hypotheses and conclusions separately. The sequent $\Gamma \vdash \Delta$ is equivalent to a right-sided sequent $\vdash \Gamma^\perp, \Delta$. This is the format used in [2, 17]. However, we will find it convenient to work with symmetric sequents. In Figure 1, we give a sequent calculus presentation of linear logic which uses symmetric sequents, but still does most of its work on the right hand side alone. For the most part, the logical intuitions

[3] We adopt the convention that $(\ )^\perp$ binds closely. So $A^{\perp\perp}$ is to be parsed as $(A^\perp)^\perp$.

$$\text{Exchange}\quad \frac{\Gamma, A, B, \Gamma' \vdash \Delta}{\Gamma, B, A, \Gamma' \vdash \Delta} \quad \frac{\Gamma \vdash \Delta, A, B, \Delta'}{\Gamma \vdash \Delta, B, A, \Delta'}$$

$$\mathsf{Id}\quad \frac{}{A \vdash A} \qquad \mathsf{Cut}_R\quad \frac{\Gamma \vdash A, \Delta \quad \Gamma' \vdash A^{\perp}, \Delta'}{\Gamma, \Gamma' \vdash \Delta, \Delta'}$$

$$(\,)^{\perp}\mathcal{R}\quad \frac{\Gamma, A \vdash \Delta}{\Gamma \vdash A^{\perp}, \Delta} \qquad (\,)^{\perp}\mathcal{L}\quad \frac{\Gamma \vdash A, \Delta}{\Gamma, A^{\perp} \vdash \Delta}$$

$$\otimes\mathcal{R}\quad \frac{\Gamma \vdash A, \Delta \quad \Gamma' \vdash B, \Delta'}{\Gamma, \Gamma' \vdash A \otimes B, \Delta, \Delta'} \qquad \parallel \mathcal{R}\quad \frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \parallel B, \Delta}$$

$$\mathbf{1}\,\mathcal{R}\quad \frac{}{\vdash \mathbf{1}} \qquad \perp\mathcal{R}\quad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \perp, \Delta}$$

$$\oplus\mathcal{R}\quad \frac{\Gamma \vdash A, \Delta}{\Gamma \vdash A \oplus B, \Delta} \quad \frac{\Gamma \vdash B, \Delta}{\Gamma \vdash A \oplus B, \Delta} \qquad \&\,\mathcal{R}\quad \frac{\Gamma \vdash A, \Delta \quad \Gamma \vdash B, \Delta}{\Gamma \vdash A \,\&\, B, \Delta}$$

$$\text{no rule for } \mathbf{0}\,\mathcal{R} \qquad \top\mathcal{R}\quad \frac{}{\Gamma \vdash \top, \Delta}$$

$$!\mathcal{R}\quad \frac{!\Gamma \vdash A, ?\Delta}{!\Gamma \vdash !A, ?\Delta} \qquad ?\mathcal{R}\quad \frac{\Gamma \vdash A, \Delta}{\Gamma \vdash ?A, \Delta}$$

$$?\mathcal{W}\quad \frac{\Gamma \vdash \Delta}{\Gamma \vdash ?A, \Delta} \qquad ?\mathcal{C}\quad \frac{\Gamma \vdash ?A, ?A, \Delta}{\Gamma \vdash ?A, \Delta}$$

Fig. 1. Sequent calculus for linear logic

given earlier should explain the rules. The annotations $\mathcal{L}$, $\mathcal{R}$, $\mathcal{W}$ and $\mathcal{C}$ stand for left-introduction, right-introduction, weakening and contraction respectively. In $!\mathcal{R}$, the notation $!\Gamma$ ($?\Delta$) means a sequence all of whose formulas are of the form $!X$ ($?Y$). The reason for this requirement is explained in Section 6.1. See [34, 47] for informal introductions to linear logic and [2, 17] for fuller treatments.

3 Towards directional logic programming

In this section, we informally motivate the concepts of directional logic programming. Our starting point is Typed Prolog [32]. A predicate definition in Typed Prolog is of the form:

$$\begin{array}{l} P : \mathit{pred}\ (A_1 \times \ldots \times A_k) \\ P(\bar{t}_1) \leftarrow \phi_1 \\ \quad \vdots \\ P(\bar{t}_n) \leftarrow \phi_n \end{array}$$

For our purpose, it will be better to work with iff-completions of the Horn clauses as defined in [7]:

$$P(\bar{x}) \iff (\exists \bar{y}_1, \bar{z}_1.\, \bar{x} = \bar{t}_1, \phi_1);\ \ldots;\ (\exists \bar{y}_n, \bar{z}_n.\, \bar{x} = \bar{t}_n, \phi_n)$$

($\bar{y}_i$ are the variables of $\bar{t}_i$ and $\bar{z}_i$ are the variables of ϕ_i which do not occur in $\bar{t}_i$). We will also use a similar iff-completion for type definitions. If

$$f_1 : A_1 \to \beta \quad \dots \quad f_n : A_n \to \beta$$

are *all* the function symbols with the atomic type β as their range type, then we assume a (possibly recursive) definition of β as

$$\mathit{type}\ \beta = f_1\ A_1 + \cdots + f_n\ A_n$$

Here, "+" is the *disjoint union* (or *sum*) type constructor and the symbols f_i are taken to be the "constructor tags" used for injection of the summands into β.

SLD-resolution with respect to the original Horn clause program can now be viewed as a certain reduction semantics with respect to the iff-completion. Treat each predicate definition as a *rewrite rule*. In addition, add the following reductions:

$$
\begin{array}{rcll}
x = t, \phi[x] & \longrightarrow & x = t, \phi[t] & \text{if } x \notin V(t) \quad (1)\\
x = x & \longrightarrow & & (2)\\
x = t & \longrightarrow & \mathit{false} & \text{if } x \in V(t) \text{ and } t \neq x \quad (3)\\
(t_1, \dots, t_k) = (u_1, \dots, u_k) & \longrightarrow & t_1 = u_1, \dots, t_k = u_k & (4)\\
f\,t = f\,u & \longrightarrow & t = u & (5)\\
f\,t = g\,u & \longrightarrow & \mathit{false} & \text{if } f \neq g \quad (6)\\
\mathit{false}, \phi & \longrightarrow & \mathit{false} & (7)\\
\exists x.\, \phi & \longrightarrow & \phi & \text{if } x \notin V(\phi) \quad (8)\\
\exists x.\, x = t, \phi & \longrightarrow & \phi & \text{if } x \notin V(\phi) \quad (9)\\
\mathit{false}; \phi & \longrightarrow & \phi & (10)\\
(\phi_1; \phi_2), \phi' & \longrightarrow & (\phi_1, \phi'); (\phi_2, \phi') & (11)
\end{array}
$$

In reduction (1), $\phi[x]$ denotes a formula with one or more occurrences of x. Reductions (1-6) handle unification, (7) handles failure, (8-9) handle "garbage collection" and (10-11) handle backtracking. The reduction system is Church-Rosser. (In traditional theory [35], this result is stated as the independence of the answer substitutions on the selection function). Now, the reduction of a goal of the form $P(\bar{t})$, if it converges, yields a possibly infinite normal form:

$$\exists \bar{y}_1.\, \bar{x} = \bar{u}_1;\ \exists \bar{y}_2.\, \bar{x} = \bar{u}_2;\ \dots$$

which denotes the answer substitutions obtained by SLD-resolution. (Handling the divergent case requires some more work; but that would lead us too far astray). For example, given the Typed Prolog program:

```
nil : List α
"." : α × List α → List α
append : pred (List α × List α × List α)
append(nil, ys, ys) ←
append(x.xs, ys, x.zs) ← append(xs,ys,zs)
```

we have the iff-completion:

```
type List α = nil + α.(List α)
append : pred (List α × List α × List α)
append(xs, ys, zs) ⟺
    (∃ ys'. xs = nil, ys = ys', zs = ys');
    (∃ x,xs',ys',zs'. xs = x.xs', ys = ys', zs = x.zs',
       append(xs', ys', zs'))
```

The goal `append(1.nil, 2.nil, ans)` reduces to the normal form

```
ans = 1.2.nil
```

From now on, we refer to this as *relational* logic programming in order to distinguish it from *directional* logic programming.

Procedural notions Predicate calls in logic programs involve various procedural notions. In using a predicate as a *condition*, we typically give it ground arguments and use both the true and false cases of the result. In using it as a *test*, we use only the true case of the result; the false case is considered a "failure". Such use as tests is found, for example, in generate-and-test programs. Finally, in using a predicate as a *procedure*, we give it both ground and nonground arguments, expecting some or all of the variables in the arguments to get bound at the end of the procedure. A further distinction can be made based on *determinacy*. Deterministic procedures are always expected to succeed and yield a single answer substitution upon success. A failure of a deterministic procedure is tantamount to an "error", *e.g.*, accessing the top of an empty stack. A nondeterministic procedure may yield multiple answer substitutions upon backtracking or fail indicating that there are no further answers. The above description paints a rich and varied picture of procedural notions used in logic programming. The challenge is to formulate a procedural semantics that supports such notions.

Our proposal is mainly directed at the use of predicates as procedures, even though conditions and tests are supported by indirect means. Even among procedures, we concentrate mainly on deterministic procedures. Our understanding of nondeterministic procedures is still preliminary.

Inputs and outputs In using a predicate as a procedure, we need to distinguish between inputs and outputs. While in the declarative reading of a predicate, all its arguments are formally treated as inputs, in the procedural reading, some quantities among the arguments are actually "place holders" for outputs, *e.g.*, `ans` in the call `append(1.nil, 2.nil, ans)`. One way to handle these distinctions is to annotate arguments as input or output. But, linear logic suggests a much more comprehensive treatment. For every type A, it provides a dual type $A^\perp$ which switches the input-output notion of A. That is, an input of type $A^\perp$ is really an output of type A and an output of type $A^\perp$ is really an input of type A. It is reasonable to think of $A^\perp$ as the type of place holders for A-typed values. Such place holders, when given to procedures, must necessarily be filled

in with values. This is a property guaranteed by the type system. So, giving an argument of type $A^\perp$ has the effect of demanding an A-typed value as an output.

Using these notions, we can define an append procedure with the type:

```
append : proc (List α ⊗ List α ⊗ (List α)⊥)
```

This constrains the append procedure so that its first two arguments are always list inputs and the third argument is always a list output. Other versions of append procedures can be similarly defined with different input-output directionality. A single predicate of relational logic programming can thus correspond to many procedures in directional logic programming. All of them share the *same* declarative semantics, but have different procedural semantics.

The advantage of combining input-output modes with types is that directionality specification is not limited to procedure arguments. It can also apply to data structures or parts of data structures. For instance, the type $List\,\alpha \otimes (List\,\alpha)^\perp$ is the type of pairs which have lists as their first components and place holders for lists as their second components. Such a type is useful in formulating difference lists, for example.

Every type constructor of Typed Prolog splits into two versions in directional logic programming: one that plays the standard input-output role and the other that plays the dual role. This is summarized in the following table:

Typed Prolog	Standard direction	Dual direction
Unit	$\mathbf{1}$	$\perp$
$\times$	$\otimes$	$\parallel$
$+$	$\oplus$	$\&$

This means that a single type of Typed Prolog can have many directional versions. For example, the directional incarnations of $A \times B$ include $A \otimes B$, $A^\perp \otimes B$, $A \otimes B^\perp$, $A^\perp \otimes B^\perp$ and four other types which use $\parallel$ in place of $\otimes$. And, even more combinations if we look inside A and B!

The "terms" of Typed Prolog split into two kinds of constructs in directional logic programming: *patterns*, when used in their input role (*e.g.*, left hand sides of clauses), and *terms*, when used in their output role (*e.g.*, as arguments of procedures). The unification operation "=" becomes a (bi-) directional unification operation "$\Leftrightarrow$". Whereas in $t_1 = t_2$, t_1 and t_2 are of the same type A, in $t_1 \Leftrightarrow t_2$, t_1 and t_2 are of dual types: A and $A^\perp$ respectively. So, t_1 is a producer of A's and t_2 is a consumer of A's and the unification $t_1 \Leftrightarrow t_2$ makes them communicate.

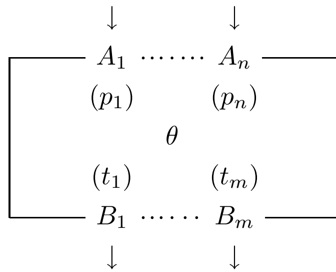
In the next section, we present a concrete programming language based on these ideas.

4 The core language

The language we present is quite large. Notice that there are 10 type constructors and, for each of them, there must be a way to produce a value of the type and a way to consume a value of the type (with some exceptions). That gives something

like 20 type rules! The symmetries of linear logic mean that only half of them are essential. The other half are just variants. However, defining the language by only half the rules sacrifices convenience and understandability. We use a compromise where the "official" definition of the language (Fig. 1, 2) chooses economy but our explanation of it chooses convenience. In this section, we present the core language which involves the type constructors $\otimes$, $\mathbf{1}$ and $\oplus$ (together with their duals $\|$, $\bot$ and $\&$). We omit $\top$ and $\mathbf{0}$ because they seem to play no role in a programming language, and relegate exponentials to a cursory treatment in Section 6.

A *computation* in a directional logic program, in general, accepts some number of *inputs* (say, of types $A_1, \ldots, A_n$) and some number of *outputs* (say, of types $B_1, \ldots, B_m$). Such a computation can be viewed as a network of the following form:[4]



In the concrete syntax, the inputs are represented by what are called *patterns* $(p_1, \ldots, p_n)$, outputs by *terms* $(t_1, \ldots, t_m)$ and the internal connections of the network by a *command* (θ).[5] We also represent the same computation as a sequent:

$$p_1 : A_1, \ldots, p_n : A_n \vdash \{\theta\}\; t_1 : B_1, \ldots, t_m : B_m$$

We use the symbols Γ and Δ to denote collections of typings on the left and right hand sides of $\vdash$. So, for example, $\Gamma \vdash \{\theta\}\Delta$ is a schematic sequent.

The directional versions of predicates are called *procedures.* Their definitions are of the form:

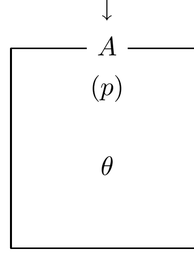
$$\begin{array}{l} P : \mathit{proc}\ A \\ P\, p \quad = \quad \theta \end{array}$$

Note that only patterns occur on the left hand side of a definition and the procedure body only contains a command. So, a procedure is a computation of

[4] In [17], these computations are called "proof nets".

[5] Note that, in traditional logic programming, input-output distinctions are ignored. So, patterns and terms merge into a single class, *viz.*, terms. Commands map to their non-directional versions, formulas.

the form:



It does not produce a *direct* output though it may produce outputs indirectly by assigning values to its output arguments.

The context-free syntax of patterns, terms and commands is as follows. Assume the syntactic classes

$$x \in \mathit{Variable}$$
$$P \in \mathit{Procedure}$$

The set of variables is partitioned into two isomorphic subsets:

$$\mathit{Variable} = \mathit{Variable}^{+} \uplus \mathit{Variable}^{-}$$
$$(\ )^{\perp} : \mathit{Variable}^{+} \cong \mathit{Variable}^{-} : (\ )^{\perp}$$

If x is a variable in one of the subsets, then its image in the other subset is denoted $x^{\perp}$. This is similar to the naming convention used in CCS [39]. The abstract syntax is given by

$$\begin{array}{rcl} p, q & \in & \mathit{Pattern} \\ t, u & \in & \mathit{Term} \\ \theta, \phi & \in & \mathit{Command} \end{array}$$

$$\begin{array}{rl} p ::= & x \mid () \mid (p_1, p_2) \mid [\,] \mid [p_1, p_2] \\ & \mid \mathit{case}(\mathit{inl}\, p_1 \Rightarrow \theta_1 \mid \mathit{inr}\, p_2 \Rightarrow \theta_2) \mid \langle p_1, _ \rangle \mid \langle _, p_2 \rangle \\ t ::= & x \mid () \mid (t_1, t_2) \mid [\,] \mid [t_1, t_2] \\ & \mid \mathit{inl}\, t \mid \mathit{inr}\, t \mid \langle \{\theta_1\} t_1,\ \{\theta_2\} t_2 \rangle \\ \theta ::= & \mathit{succ} \mid p \Leftarrow t \mid t_1 \Leftrightarrow t_2 \mid \theta_1, \theta_2 \mid \nu x.\theta \mid P\, t \\ & \mid \mathit{error} \mid \mathit{fail} \mid \theta_1 ; \theta_2 \end{array}$$

The meaning of these constructs is explained in the remainder of the section together with their type rules.

4.1 Traditional data structures

We start with the constructs that the reader can easily relate to: term formation rules for traditional data structures. $A \otimes B$ is the type of pairs, $\mathbf{1}$ is the unit

type (consisting of the empty tuple) and $A \oplus B$ is the sum type (consisting of variants). The type rules are as follows:

$$\frac{}{x : A \vdash \{succ\}\ x : A}\ \mathsf{Id}$$

$$\frac{\mathit{\Gamma} \vdash \{\theta\}\ t : A, \mathit{\Delta} \quad \mathit{\Gamma}', p : A \vdash \{\phi\}\mathit{\Delta}'}{\mathit{\Gamma}, \mathit{\Gamma}' \vdash \{\theta, p \Leftarrow t, \phi\}\mathit{\Delta}, \mathit{\Delta}'}\ \mathsf{Cut}$$

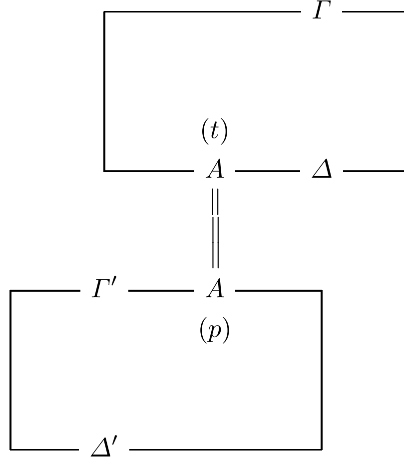
$$\frac{\mathit{\Gamma}_1 \vdash \{\theta_1\}\ t : A, \mathit{\Delta}_1 \quad \mathit{\Gamma}_2 \vdash \{\theta_2\}\ u : B, \mathit{\Delta}_2}{\mathit{\Gamma}_1, \mathit{\Gamma}_2 \vdash \{\theta_1, \theta_2\}\ (t, u) : A \otimes B, \mathit{\Delta}_1, \mathit{\Delta}_2}\ \otimes\mathcal{R} \qquad \frac{\mathit{\Gamma}, p : A, q : B \vdash \{\theta\}\mathit{\Delta}}{\mathit{\Gamma}, (p, q) : A \otimes B \vdash \{\theta\}\mathit{\Delta}}\ \otimes\mathcal{L}$$

$$\frac{}{\vdash \{succ\}\ () : \mathbf{1}}\ \mathbf{1}\mathcal{R} \qquad \frac{\mathit{\Gamma} \vdash \{\theta\}\ \mathit{\Delta}}{\mathit{\Gamma}, () : \mathbf{1} \vdash \{\theta\}\mathit{\Delta}}\ \mathbf{1}\mathcal{L}$$

$$\frac{\mathit{\Gamma} \vdash \{\theta\}\ t : A, \mathit{\Delta}}{\mathit{\Gamma} \vdash \{\theta\}\ inl\, t : A \oplus B, \mathit{\Delta}}\ \oplus\mathcal{R}_1 \qquad \frac{\mathit{\Gamma} \vdash \{\theta\}\ u : B, \mathit{\Delta}}{\mathit{\Gamma} \vdash \{\theta\}\ inr\, u : A \oplus B, \mathit{\Delta}}\ \oplus\mathcal{R}_2$$

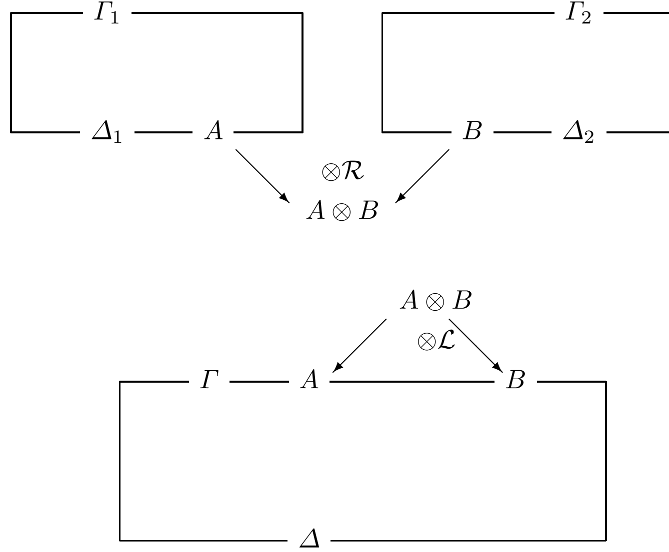
$$\frac{\mathit{\Gamma}, p : A \vdash \{\theta\}\ \mathit{\Delta} \quad \mathit{\Gamma}, q : B \vdash \{\phi\}\ \mathit{\Delta}}{\mathit{\Gamma}, case(inl\, p \Rightarrow \theta \mid inr\, q \Rightarrow \phi) : A \oplus B \vdash \{succ\}\ \mathit{\Delta}}\ \oplus\mathcal{L}$$

The rules are mostly self-explanatory, but some minor comments are in order. In the rule Id, *succ* (short for “succeed”) is to be thought of as the empty command which always succeeds (similar to *true* in Prolog). We often omit writing $\{succ\}$ in a sequent. In the rule Cut, the command $p \Leftarrow t$ means a pattern match and “,” denotes symmetric composition. The computation constructed by the Cut rule is of the following form:



The A-typed output of the first computation is plugged into the A-typed input of the second computation. Similarly, the computations formed by $\otimes\mathcal{R}$ and $\otimes\mathcal{L}$

are



Note that a pair is produced by producing its components from two *separate* computations. In contrast, the consumption of a pair involves consuming both its components in the *same* computation. This difference in how pairs are produced and consumed is crucial.

In the $\oplus\mathcal{R}$ rules, *inl* and *inr* are constructors to indicate which summand the value is in. A pattern of the form $case(inl\, p \Rightarrow \theta \mid inr\, q \Rightarrow \phi)$ constructed in the rule $\oplus\mathcal{L}$, can only be matched with $inl\, t$ or $inr\, u$. One of the cases is selected, as appropriate, and the other case is discarded. It may be surprising that the *case* pattern introduced in this rule has to be so complex. The pattern cannot be something as simple as $case(inl\, p|inr\, q)$ because the commands that need to be performed in each case have to be *suspended* until one of the cases is selected. Hence, the need to associate commands with the cases. Another special feature of $\oplus\mathcal{L}$ is that the contexts Γ and Δ in the two premises must be identical. If we need to join premises with different pattern/term constructions, new variables can be introduced (using Cut) so that the patterns/terms of the premises match up. For example, the derivation

$$\dfrac{\dfrac{\dfrac{}{x : X \vdash x : X}\,\mathsf{Id} \qquad p' : X, p : A \vdash \{\theta\}\, t : Y}{x : X, p : A \vdash \{p' \Leftarrow x, \theta\}\, t : Y}\,\mathsf{Cut} \qquad \dfrac{}{y : Y \vdash y : Y}\,\mathsf{Id}}{x : X, p : A \vdash \{p' \Leftarrow x, \theta, y \Leftarrow t\}\, y : Y}\,\mathsf{Cut}$$

replaces p' and t with fresh variables x and y.

In practice, this notation for case-patterns would be too cumbersome. We use Horn-clause syntax as “syntactic sugar” for writing the cases separately.

The rule is that a predicate definition

$$P\ (p[case(inl\ q_1 \Rightarrow \theta_1 \mid inr\ q_2 \Rightarrow \theta_2)]) \quad = \quad \phi$$

can be written as two separate clauses:

$$\begin{array}{c} P\ (p[inl\ q_1]) \leftarrow \theta_1, \phi \\ P\ (p[inr\ q_2]) \leftarrow \theta_2, \phi \end{array}$$

and each of them can be further split into multiple clauses, if necessary.

Linearity We must now state an important property of the type system (which also extends to the remainder of the system to be presented later). Note that, as long as $\oplus\mathcal{L}$ is not used, all derivable sequents have *two* occurrences of each variable. The Id rule introduces two occurrences and the other rules preserve this property.[6] Of the two occurrences of a variable, one is an input occurrence and the other is an output occurrence. Thus, linearity means that each variable has a unique "producer" and a unique "consumer". Similar restrictions are used in Doc [25] and Janus [45], and the "directed logic variables" of Kleinman *et. al.* [28]. Also, as in these languages, linear logic allows duplication operators (Section 6.1). By using a duplication operator as the consumer of a value, we obtain the effect that two copies of the value are generated. Thus, linearity is not a restriction in *practice.*

The property of linearity, suitably generalized, holds for the use of $\oplus\mathcal{L}$ rule as well. The basic difference is that, for the pattern $case(inl\ p \Rightarrow \theta \mid inr\ q \Rightarrow \phi)$ a free occurrence of x in θ as well as ϕ counts as a single occurrence.

Any variable that is both produced and consumed within a command is really a "local" variable and it is treated as such in the theory. In practice, it would be clearer to use an explicit restriction operator to indicate such binding. Let $V^+(\theta)$ and $V^-(\theta)$ denote the set of variables that have free (unrestricted) occurrences in θ as terms and patterns respectively. Then, we can introduce restriction by:

$$\frac{\Gamma \vdash \{\theta\}\Delta}{\Gamma \vdash \{\nu x.\, \theta\}\Delta}\ \mathsf{Restrict} \qquad \text{if } x \in V^+(\theta) \cap V^-(\theta)$$

Reduction semantics To formalize the meaning of the above constructs, we first impose a few equivalences:

$$\theta_1, \theta_2 = \theta_2, \theta_1 \tag{12}$$

$$\theta_1, (\theta_2, \theta_3) = (\theta_1, \theta_2), \theta_3 \tag{13}$$

$$succ, \theta = \theta \tag{14}$$

The equations state that the order of the occurrence of commands in a composite command are immaterial and that *succ* is the empty command. Note that these

[6] Whenever two sequents are joined using a type rule, their respective variables must be renamed apart from each other, unless the rule requires otherwise. An example of the latter is $\oplus\mathcal{L}$ where Γ and Δ in the two premises must be identical.

are essentially the equations of the linear CHAM [2, 5]. The semantics of the constructs is defined by the following reduction rules:

$$x \Leftarrow t, \theta[x]^T \longrightarrow \theta[t] \tag{15}$$

$$p \Leftarrow x, \theta[x]^P \longrightarrow \theta[p] \tag{16}$$

$$() \Leftarrow () \longrightarrow \mathit{succ} \tag{17}$$

$$(p_1, p_2) \Leftarrow (t_1, t_2) \longrightarrow p_1 \Leftarrow t_1,\ p_2 \Leftarrow t_2 \tag{18}$$

$$\mathit{case}(\mathit{inl}\ p \Rightarrow \theta_1 \mid \mathit{inr}\ q \Rightarrow \theta_2) \Leftarrow \mathit{inl}\ t \longrightarrow p \Leftarrow t,\ \theta_1 \tag{19}$$

$$\mathit{case}(\mathit{inl}\ p \Rightarrow \theta_1 \mid \mathit{inr}\ q \Rightarrow \theta_2) \Leftarrow \mathit{inr}\ u \longrightarrow q \Leftarrow u,\ \theta_2 \tag{20}$$

$\theta[x]^T$ and $\theta[x]^P$ denote commands that contain at least one occurrence of x as a term and pattern respectively. Note that linearity means that there is exactly one such occurrence (modulo the proviso for *case* patterns). The semantics of procedure call is given by

$$P\,t \quad \longrightarrow \quad p \Leftarrow t, \theta \tag{21}$$

whenever P is defined by $P\,p = \theta$.

Note that the reductions (15–16) are the directional versions of the reduction (1) of unification, (17–18) capture (4) and (19–20) capture (5). The reductions (2,3,6) of unification do not arise in directional logic programming. This is significant. It means that directional logic programs do not "fail" in the conventional sense. On the other hand, there is need for two forms of failure mechanisms. The first is a way to escape from an erroneous case. For this purpose, we use an *error* command:

$$\frac{}{\Gamma \vdash \{\mathit{error}\}\ \Delta}\ \mathsf{ERROR}$$

with the equivalence:

$$\mathit{error}, \theta \quad = \quad \mathit{error} \tag{22}$$

We often abbreviate $\mathit{case}(\mathit{inl}\ p \Rightarrow \mathit{error} \mid \mathit{inr}\ q \Rightarrow \theta)$ to $\mathit{case}(\mathit{inr}\ q \Rightarrow \theta)$ (and, similarly for *error* in the *inr* case). The second mechanism is an explicit failure command for terminating backtracking. (See Sections 4.3 and 6.2).

We also use an equivalence similar to (16) on Horn clause notation for syntactically simplifying clauses:

$$P\ (p[x]^P) \leftarrow (q \Leftarrow x, \theta) \quad = \quad P\ (p[q]) \leftarrow \theta \tag{23}$$

(Note that this equivalence is of no significance for evaluation because it is commands—not clauses—that are are evaluated).

4.2 Negation

The negation operator $(\ )^\perp$ switches the input-output notion of a type. An output of type $A^\perp$ is really an input of type A. Similarly, an input of type $A^\perp$ is really an output of type A. Since inputs and outputs are represented in the concrete syntax by patterns and terms respectively, we need a way to convert

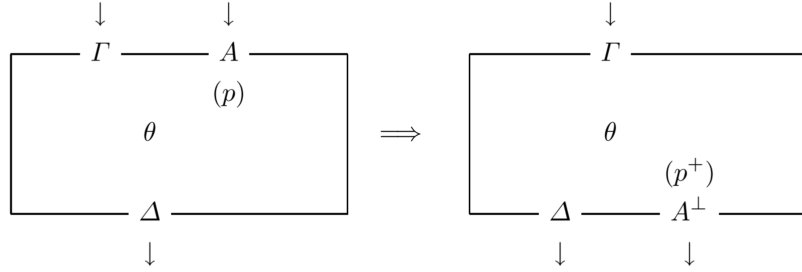
patterns to terms and *vice versa*. For this purpose, we will define two syntactic functions

$$(\,)^+ : \mathit{Pattern} \rightarrow \mathit{Term}$$
$$(\,)^- : \mathit{Term} \rightarrow \mathit{Pattern}$$

as inverses of each other. We refer to them as *dualizing maps*. Using these maps, the type rules for negation are expressed as follows:

$$\frac{\Gamma, p : A \vdash \{\theta\}\ \Delta}{\Gamma \vdash \{\theta\}\ p^+ : A^\perp, \Delta}\ ()^\perp\mathcal{R} \qquad \frac{\Gamma \vdash \{\theta\}\ t : A, \Delta}{\Gamma, t^- : A^\perp \vdash \{\theta\}\ \Delta}\ ()^\perp\mathcal{L}$$

In the $(\,)^\perp\mathcal{R}$ rule, an input p of type A is viewed as an output p^+ of type $A^\perp$. (Similarly, in $(\,)^\perp\mathcal{L}$, an output t of type A is viewed as an input t^- of type $A^\perp$). This is merely a change of view point:



The computation net remains the same in this transformation. Only the labels (and concrete syntax) change using the dualizing maps.

Negation and identity The syntax for dualizing variable symbols is the expected:

$$x^+ \;=\; x^\perp \qquad x^- \;=\; x^\perp \tag{24}$$

By combining negation rules with Id, we obtain two interesting variants as derived rules:

$$\frac{}{\vdash \{succ\}\ x^\perp : A^\perp, x : A}\ \mathsf{Id}_R \qquad \frac{}{x : A, x^\perp : A^\perp \vdash \{succ\}}\ \mathsf{Id}_L$$

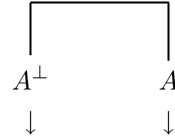
Since these rules are the key to understanding linear logic and directional logic programming, we will spend some time in explaining them.

The original Id rule denotes a computation of the following form:

This represents a "communication link" that accepts an A-typed value at its input-end and communicates it to its output-end. In traditional syntax, the

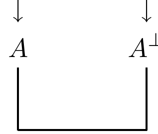
input occurrences are always used in variable binding positions and the output occurrences in term positions. Directional logic programming requires us to take a symmetric view of this phenomenon. In particular, it is admissible to have both the input and output occurrences of a variable link to be in term positions or both to be in binding positions.

The computation of Id_R can be pictured as



This says that an $A^\perp$-typed value and an A-typed value are simultaneously produced at two ends of the communication link. But, producing an $A^\perp$-typed value means the same as *receiving* an A-typed value. Thus, the link can be thought of as propagating an A-typed value from its $A^\perp$-end to its A-end. At the same time, whatever has been said about A-typed values can be said about $A^\perp$-typed values in the opposite direction. So, the link can *also* be thought of as propagating an $A^\perp$-typed value from right to left. Which of the two propagations "actually happens" depends on various considerations including irrelevant run-time factors. So, it is best to think of the two propagations, A-typed values from left to right and $A^\perp$-typed values from right to left, as being *equivalent*.

The computation denoted by Id_L is just another view of the communication link:



Here, either an A-typed value is communicated from left to right or, equivalently, an $A^\perp$-typed value from right to left.

Negation and cut Combining the Cut rule with negation rules gives the following interesting derivation:

$$\dfrac{\dfrac{\Gamma, p : A \vdash \{\theta\}\ \Delta}{\Gamma \vdash \{\theta\}\ p^+ : A^\perp, \Delta}\ ()^\perp\mathcal{R} \qquad \dfrac{\Gamma' \vdash \{\phi\}\ t : A, \Delta'}{\Gamma', t^- : A^\perp \vdash \{\phi\}\Delta'}\ ()^\perp\mathcal{L}}{\Gamma, \Gamma' \vdash \{\theta, t^- \Leftarrow p^+, \phi\}\Delta, \Delta'}\ \mathsf{Cut}$$

Since we maintain that the negation rules do not alter the underlying computation, the effect of this ought to be the same as cutting the original premises. In other words, $t^- \Leftarrow p^+$ must be equivalent to $p \Leftarrow t$. The dualizing functions make $\Leftarrow$ a "symmetric" operation.

However, the syntax of $\Leftarrow$ is badly asymmetric. The left hand side is required to be a pattern and the right hand side a term. To fix this problem, we take the

following version of the cut rule as more basic:

$$\frac{\Gamma \vdash \{\theta\}\, t : A, \Delta \quad \Gamma' \vdash \{\phi\}\, u : A^{\perp}, \Delta'}{\Gamma, \Gamma' \vdash \{\theta, t \Leftrightarrow u, \phi\}\, \Delta, \Delta'}\, \mathsf{Cut}_R$$

The notation $t \Leftrightarrow u$ has the advantage that both the operands are terms. We impose the equivalence

$$t \Leftrightarrow u \quad = \quad u \Leftrightarrow t \tag{25}$$

and define the traditional pattern match in terms of "$\Leftrightarrow$":

$$p \Leftarrow t \quad \stackrel{\text{def}}{=} \quad p^{+} \Leftrightarrow t$$

Thus, $t \Leftrightarrow u$ is equivalent to $t^{-} \Leftarrow u$ as well as $u^{-} \Leftarrow t$.

The command $t \Leftrightarrow u$ can be thought of as a *directional unification* operation. The term t produces an A-typed value and the term u produces an $A^{\perp}$-typed value, *i.e.*, u is prepared to *accept* an A-typed value. The execution of $t \Leftrightarrow u$ instantiates the variables of t and u such that the value produced by t is the same as the value accepted by u and *vice versa*. It is important to note that "$\Leftrightarrow$" denotes *bidirectional* communication. For instance, if $A = int \otimes int^{\perp}$, t produces the first component of the pair and consumes the second component whereas u does exactly the opposite. Such bidirectional communication, which gives logic programming its expressive power, is preserved in directional logic programming.

Example 1. The identity procedure can be defined as

```
id : proc (α ⊗ α⊥)
id (x, x⊥) = succ
```

For all types α, the identity procedure transfers the first argument of type α to the second argument as an output. However, since α can also be a negative type, the same procedure can also transfer the second argument to the first argument.

The identity procedure can also be defined, somewhat frivolously, as

```
id (x, y) = (x ⇔ y)
```

To type check this definition, we need to derive the sequent $x : \alpha, y : \alpha^{\perp} \vdash \{x \Leftrightarrow y\}$. This may be split into two sequents using Cut_R:

$$x : \alpha \vdash \{succ\}\, x : \alpha$$
$$y : \alpha^{\perp} \vdash \{succ\}\, y : \alpha^{\perp}$$

each of which is an instance of Id. ■

Example 2. Assume a polymorphic type definition of lists as follows:

```
List α = nil 1 ⊕ cons (α ⊗ List α)
```

The constructors `nil` and `cons` replace the formal constructors *inl* and *inr*.

A conventional procedure for appending lists may be defined as follows:

```
append : proc (List α ⊗ List α ⊗ (List α)⊥)
append(xs, ys, zs) =
    case(nil() ⇒ zs ⇔ ys
       | cons(x,xs') ⇒ ν zs'. append(xs',ys,zs'⊥),
                              zs ⇔ cons(x,zs'))
    ⇐ xs
```

Using Horn clause notation, the same definition can also be written as

```
append(nil(), ys, zs) ← zs ⇔ ys
append(cons(x,xs'), ys, zs) ←
    ν zs'. append(xs',ys,zs'⊥), zs ⇔ cons(x,zs')
```

Two comments are in order. First, we are unable to use pattern matching syntax for the third parameter of *append*. This is remedied in Section 4.3. Secondly, the *append* procedure, as defined above, can only be used to append lists. It cannot be used "backwards" to split a list into two parts. This restriction is directly stated in types. The first two arguments are list inputs and the third argument is a list output. Operationally, the pattern match `case(...)` $\Leftarrow$ `xs` cannot be executed until *xs* is instantiated to a *nil* or a *cons* term. Thus, directionality is built into the procedure. ■

It may be seem that we have lost some of the convenience of logic programming by not being able to run procedures like *append* backwards. But, note that sequential control, such as that of Prolog, also loses the ability to run procedures backwards in any practical sense. We noted this for the *reverse* predicate in Section 1. In fact, it appears that most Prolog programs are written with a specific input-output directionality in mind. Our objective is to formalize these directionality notions and, then, to replace the low-level operational mechanisms of control used in conventional logic languages with high-level declarative mechanisms.

4.3 Dual data structures

This section is essentially an exercise in syntax. All the computations were already introduced in Section 4.1 while, in Section 4.2, we introduced "change of view point". We must now introduce enough syntax to execute the change of view point. This involves defining new type constructors that are dual to the traditional ones, and pattern and term forms for these type constructors.

The dual type constructors are defined by the following equations:

$$\begin{aligned}(A \otimes B)^\perp &= A^\perp \parallel B^\perp \\ \mathbf{1}^\perp &= \perp \\ (A \oplus B)^\perp &= A^\perp \,\&\, B^\perp\end{aligned}$$

The new type constructors are dual to the old ones in that, forming an output of an old type is similar to forming an input of its dual type and forming an

input of an old type is similar to forming an output of its dual type. With this understanding, we propose the following syntax for new pattern/term forms:[7]

$$\begin{array}{rlrl} (p,q)^+ &= [p^+,q^+] & (t,u)^- &= [t^-,u^-] \\ ()^+ &= [\,] & ()^- &= [\,] \\ \mathit{case}(\mathit{inl}\ p \Rightarrow \theta \mid \mathit{inr}\ q \Rightarrow \phi)^+ &= \langle \{\theta\}p^+, \{\phi\}q^+ \rangle & (\mathit{inl}\ t)^- &= \langle t^-, _ \rangle \\ & & (\mathit{inr}\ u)^- &= \langle _, u^- \rangle \end{array} \tag{26}$$

The type rules for the new constructs are obtained as derived rules by combining the type rules of Section 4.1 with negation rules:

$$\frac{\Gamma \vdash \{\theta\}\ t : A, u : B, \Delta}{\Gamma \vdash \{\theta\}\ [t,u] : A \parallel B, \Delta} \parallel\mathcal{R} \qquad \frac{\Gamma_1, p : A \vdash \{\theta_1\}\ \Delta_1 \quad \Gamma_2, q : B \vdash \{\theta_2\}\ \Delta_2}{\Gamma_1, \Gamma_2, [p,q] : A \parallel B \vdash \{\theta_1, \theta_2\}\ \Delta_1, \Delta_2} \parallel\mathcal{L}$$

$$\frac{\Gamma \vdash \{\theta\}\ \Delta}{\Gamma \vdash \{\theta\}\ [\,] : \bot, \Delta} \bot\mathcal{R} \qquad \frac{}{[\,] : \bot \vdash \{\mathit{succ}\}} \bot\mathcal{L}$$

$$\frac{\Gamma \vdash \{\theta\}\ t : A, \Delta \quad \Gamma \vdash \{\phi\}\ u : B, \Delta}{\Gamma \vdash \{\mathit{succ}\}\ \langle \{\theta\}t, \{\phi\}u \rangle : A \,\&\, B,\ \Delta} \&\mathcal{R}$$

$$\frac{\Gamma, p : A \vdash \{\theta\}\ \Delta}{\Gamma, \langle p, _ \rangle : A \,\&\, B \vdash \{\theta\}\ , \Delta} \&\mathcal{L}_1 \qquad \frac{\Gamma, q : B \vdash \{\theta\}\ \Delta}{\Gamma, \langle _, q \rangle : A \,\&\, B \vdash \{\theta\}\ \Delta} \&\mathcal{L}_2$$

We briefly explain each of these constructions:

- The type constructor "$\parallel$" (read as "par" — short for *parallelization*) is dual to $\otimes$. Recall that a $\otimes$-pair is produced by producing its components independently and consumed by consuming its components in the same computation. Dually, a $\parallel$-pair is produced by producing both its components from the same computation and consumed by consuming its components in separate computations. We call pairs $[t,u]$ of type $A \parallel B$ "connected pairs" because the two components are connected via a computation. They are extremely useful for building data structures whose components are dependent on each other. (See Examples 4 and 5).
- The type $\bot$ is useful for constructing a dummy input, just as $\mathbf{1}$ is useful for constructing a dummy output. Note that $\bot\mathcal{L}$ is the only rule, other than Id_L, that builds an empty command from scratch. (See Example 6).
- $A \,\&\, B$ is the type of pairs whose computation is delayed until one of its components is selected. Hence, we call them *lazy pairs*. Note that precisely one component of a lazy pair may be used; the other component is discarded.

When a type is defined, we can introduce *selector* symbols for lazy pairs:

$$\mathtt{type}\ A = s_1\ B_1 \,\&\, \ldots \,\&\, s_n\ B_n$$

Given such a definition, the pattern $\langle _, \ldots, p_i, \ldots, _ \rangle$ can be written as $s_i\ p_i$ and the term $\langle \{\theta_1\}t_1, \ldots, \{\theta_n\}t_n \rangle$ can be written as $\langle s_1 \Rightarrow \{\theta_1\}t_1, \ldots, s_n \Rightarrow \{\theta_n\}t_n \rangle$.

If the list type is defined by

[7] Unfortunately, our notation for duals of tuples ("[. . .]") conflicts with the Prolog notation for lists. Note that we never use "[. . .]" for lists in this paper.

$$\texttt{List}\ \alpha\ \texttt{= nil 1}\ \oplus\ \texttt{cons}\ (\alpha \otimes \texttt{List}\ \alpha)$$

its dual type gets the definition

$$\texttt{(List}\ \alpha\texttt{)}^{\perp}\ \texttt{= nil}\ \perp\ \texttt{\&}\ \texttt{cons}\ (\alpha^{\perp} \parallel \texttt{(List}\ \alpha\texttt{)}^{\perp})$$

This definition says that an acceptor for lists is a lazy pair of two acceptors. In the *nil* case, we use the first component which is an acceptor of ()'s. The second component, used in the *cons* case, is a connected pair of an element acceptor and another list acceptor. Thus, an acceptor of lists does not have to be simply a "place holder". It can have interesting structure. To illustrate how this structure gets used, we look at the reduction semantics.

Reduction semantics First we complete the definition of dualizing maps:

$$\begin{array}{rlrl} [p,q]^+ &= (p^+, q^+) & [t,u]^- &= (t^-, u^-) \\ [\,]^+ &= () & [\,]^- &= () \\ \langle p, _\rangle^+ &= \mathit{inl}\ p^+ & \langle\{\theta\}t, \{\phi\}u\rangle^- &= \mathit{case}(\mathit{inl}\ t^- \Rightarrow \theta \mid \mathit{inr}\ u^- \Rightarrow \phi) \\ \langle _, q\rangle^+ &= \mathit{inr}\ q^+ & & \end{array} \tag{27}$$

The reduction semantics for the full language is shown in Figure 2. Recall that

Equivalences

$$\begin{array}{rl} \theta_1, \theta_2 &= \theta_2, \theta_1 \\ \theta_1, (\theta_2, \theta_3) &= (\theta_1, \theta_2), \theta_3 \\ \mathit{succ}, \theta &= \theta \\ t \Leftrightarrow u &= u \Leftrightarrow t \end{array}$$

Reductions

$$\begin{array}{rl} x \Leftrightarrow t, \theta[x^{\perp}] &\longrightarrow \theta[t] \\ [\,] \Leftrightarrow () &\longrightarrow \mathit{succ} \\ [t_1, t_2] \Leftrightarrow (u_1, u_2) &\longrightarrow t_1 \Leftrightarrow u_1, t_2 \Leftrightarrow u_2 \\ \langle\{\theta_1\}t_1, \{\theta_2\}t_2\rangle \Leftrightarrow \mathit{inl}\ u &\longrightarrow t_1 \Leftrightarrow u, \theta_1 \\ \langle\{\theta_1\}t_1, \{\theta_2\}t_2\rangle \Leftrightarrow \mathit{inr}\ u &\longrightarrow t_2 \Leftrightarrow u, \theta_2 \\ P\ t &\longrightarrow p^+ \Leftrightarrow t, \theta \quad \text{where } P\ p = \theta \text{ is in program} \end{array}$$

Fig. 2. Reduction semantics of the core language

we take the unification operation "$\Leftrightarrow$" to be basic and define "$\Leftarrow$" in terms of "$\Leftrightarrow$". Thus, commands in Fig. 2 do not have any patterns in them. Using the definition of dualizing maps, (24), (26), (27), the earlier reductions (15–20) are seen as syntactic variants of the official reduction rules in the figure. In addition, the following reductions are also obtained as syntactic variants:

$$[p_1, p_2] \Leftarrow [t_1, t_2] \longrightarrow p_1 \Leftarrow t_1, p_2 \Leftarrow t_2 \tag{28}$$

$$[\,] \Leftarrow [\,] \longrightarrow succ \tag{29}$$

$$\langle p, _ \rangle \Leftarrow \langle \{\theta\}t, \{\phi\}u \rangle \longrightarrow p \Leftarrow t, \theta \tag{30}$$

$$\langle _, q \rangle \Leftarrow \langle \{\theta\}t, \{\phi\}u \rangle \longrightarrow q \Leftarrow u, \phi \tag{31}$$

Example 3. We return to the *append* program of Example 2 and rewrite it using the new constructs:

```
append : proc (List α ⊗ List α ⊗ (List α)⊥)
append(nil(), ys, ys⊥) ← succ
append(cons(x,xs'), ys, cons[x⊥,zs'⊥]) ←
    append(xs', ys, zs'⊥)
```

The third parameter of *append* is an acceptor of lists. The definition of $(List\,\alpha)^\perp$ shows that such acceptors have an interesting structure. This structure is used in writing the pattern for the third parameter in the second clause. Type checking the clauses involves deriving the following sequents:

$$() : \mathbf{1},\ ys : L,\ ys^\perp : L^\perp\ \vdash\ \{succ\}$$
$$(x, xs') : \alpha \otimes L,\ ys : L,\ cons[x^\perp, zs'^\perp] : L^\perp\ \vdash\ \{append(xs', ys, zs'^\perp)\}$$

where L stands for $List\,\alpha$. We leave them for the reader to verify.

To see how the pattern of the third parameter works, suppose *append* is invoked with a variable $zs^\perp$ as the third argument. This involves the following pattern match:

$$cons[x^\perp, zs'^\perp] \Leftarrow zs^\perp$$

If its dual zs is passed to another procedure, it must eventually execute a case analysis on zs of the following form:

$$case(nil() \Rightarrow \theta_1 \mid cons(a, as) \Rightarrow \theta_2) \Leftarrow zs$$

This is equivalent to the unification

$$zs \Leftrightarrow \langle nil \Rightarrow \{\theta_1\}[\,],\ cons \Rightarrow \{\theta_2\}[a^\perp, as^\perp]\rangle$$

So, the acceptor-pair on the right may get substituted for $zs^\perp$ in the first pattern match. This gives

$$cons[x^\perp, zs'^\perp] \Leftarrow \langle nil \Rightarrow \{\theta_1\}[\,],\ cons \Rightarrow \{\theta_2\}[a^\perp, as^\perp]\rangle$$

which reduces to $x^\perp \Leftarrow a^\perp,\ zs'^\perp \Leftarrow as^\perp,\ \theta_2$.

On the other hand, the first pattern match is equivalent to the unification

$$zs^\perp \Leftrightarrow cons(x, zs')$$

So, $cons(x, zs')$ may be substituted for zs in the second pattern match. Thus, whether values are passed in one direction or acceptors in the opposite direction, the same results are obtained. ■

Example 4. The type of difference lists can be defined in a directional logic program as

```
type DList α = List α ∥ (List α)⊥
```

We will refer to the two components as the "head" and the "tail" respectively. The head of a difference list is a list and its tail is an acceptor of lists. The two components are *connected*: the list accepted in the tail is used in the head.

Dually, an acceptor of difference lists is of the type:

```
type (DList α)⊥ = (List α)⊥ ⊗ (List α)
```

It accepts the head of the difference list and gives a list for its tail.

Procedures for creation and concatenation of difference lists may be defined as follows:

```
new : proc (DList α)⊥
concat : proc (DList α ⊗ DList α ⊗ (DList α)⊥)

new (x, x⊥) = succ
concat([x,y⊥], [y,z⊥], (x⊥,z)) = succ
```

Note that the only parameter of *new* and the third parameter of *concat* are output parameters.

Type checking the definition of *concat* illuminates how $\otimes$ and $\parallel$ interact in derivations. Let L stand for *List* α. Then, we need to derive

$$[x, y^{\perp}] : L \parallel L^{\perp},\ [y, z^{\perp}] : L \parallel L^{\perp},\ (x^{\perp}, z) : L^{\perp} \otimes L\ \vdash\ \{succ\}$$

First, use $\otimes\mathcal{L}$ to decompose the pair $(x^{\perp}, z)$ into separate hypotheses $x^{\perp} : L^{\perp}$ and $z : L$. Next, use $\parallel \mathcal{L}$ to decompose $[x, y^{\perp}]$ giving the sequents:

$$x : L, x^{\perp} : L^{\perp} \vdash \{succ\}$$
$$y^{\perp} : L^{\perp}, [y, z^{\perp}] : L \parallel L^{\perp}, z : L \vdash \{succ\}$$

The first sequent follows by Id_L. The second sequent may be derived by further decomposing $[y, z^{\perp}]$ and using Id_L twice.

Here are some more operations on difference lists. (The reader is encouraged to work through the type derivations for these clauses).

```
addl : proc (α ⊗ DList α ⊗ (DList α)⊥)
addr : proc (DList α ⊗ α ⊗ (DList α)⊥)
out : proc (DList α ⊗ (List α)⊥)

addl (i, [h,t⊥], (cons[i⊥,h⊥], t)) = succ
addr ([h, cons[i⊥,t⊥]], i, (h⊥,t)) = succ
out ([h, nil[]], h⊥) = succ
```

Interestingly, it does not seem possible to define a procedure to delete the first element of a difference list in the same fashion as above. The problem is that the first element itself is dependent on the tail input of the difference list. So, it cannot be separated away from the tail input. ■

Example 5. We illustrate how to achieve back communication required in concurrent message passing programs. The following is a "stack manager" process that maintains an internal stack and services messages for pushing and popping values.

```
type Msgs α = done ⊥ ⊕ push(α ⊗ Msgs α) ⊕ pop(α⊥ ∥ Msgs α)
stack : proc (Msgs α)
stackloop : proc (Msgs α ⊗ List α)

stack ms = stackloop(ms, nil())
stackloop(done dummy, st) ← dump(st, dummy)
stackloop(push(x, ms'), st) ← stackloop(ms', cons(x,st))
stackloop(pop[x⊥, ms'], nil()) ← error
stackloop(pop[x⊥, ms'], cons(x,st')) ← stackloop(ms', st')
```

Note that a `push` message comes with an α, but a `pop` message comes with an acceptor for α's. The stack process sends back an α to the originator of the message via this acceptor. (See the last clause).

Traditionally, in concurrent logic languages, messages are structured as a stream (list) with components of the form *push* v and *pop* x. Expressing it in our type system would yield the following type:

```
type Msgs' α = List (push α ⊕ pop α⊥)
```

This type subtly differs from our *Msgs* type: the *pop*-acceptor is combined with the remaining stream using $\otimes$ rather than $\|$. This would mean that the last clause would not type check (because $x^\perp$ and ms' are consumed separately). More importantly, it does not allow the user process to use the popped value in future messages. Our *Msgs* type corrects these problems and provides the right structure for bidirectional communication.

In the first clause of `stackloop`, we would ideally like to have an empty clause body. But, that would mean that `st` is not used, violating linearity. (Such a clause body would not type check). We use here a procedure `dump` which magically discards the stack. See Example 6 for its definition. ■

Type checking in directional logic programming guarantees certain desirable properties for concurrent programming. For example, a *Msgs* stream can be consumed by one and only one stack process, and, unless there is an error, the stream is consumed completely and all the acceptors in it are satisfied. In general, there is a single process that has "write access" to a variable and this process is guaranteed to write the variable (unless there is an error or nontermination). In other words, no race conditions or deadlocks arise. The "directionality checks" of Parlog [21] are meant to guarantee similar properties. However, these checks are possible only for "strong" modes, not for "weak" modes. Strong modes would rule out back communication as in the above example. In contrast, our proposal allows the so-called weak modes and, at the same time, guarantees safety and liveness properties.

This guarantee is obtained, in large part, by distinguishing between the two kinds of pairs: *independent* pairs belonging to $\otimes$-types and *connected* pairs belonging to $\|$-types. The components of independent pairs are separately constructed; so, it is all right to use both the components in the same computation. On the other hand, the components of connected pairs are dependent on each other. Were we to assume that the components are independent and subsequently link them, we are likely to create cycles. For example, if we disregard the connectedness of the pair $[x^\perp, x]$ and treat is as being of type $A^\perp \otimes A$ then we can create the cyclic computation:

$$x^\perp \Leftrightarrow x$$

This command waits forever for the value of x in an effort to produce a value for x! Thus, the distinction between $\otimes$ and $\|$ is crucial.

However, there is a sense in which the distinction between $\otimes$ and $\|$ is overstated. While cycles result if we assume that $\|$-pairs are disconnected, nothing really goes wrong if we use disconnected components to form a $\|$-pair. In this case, $A \parallel B$ is the type of *possibly* connected pairs. Since the use of such pairs still assumes that the components may be connected, this is a safe loss of information. The easiest way to add such loss of information is via the so-called MIX rule [12, 17]:

$$\frac{\Gamma \vdash \{\theta\}\ \Delta \quad \Gamma' \vdash \{\theta'\}\ \Delta'}{\Gamma, \Gamma' \vdash \{\theta; \theta'\}\ \Delta, \Delta'}\ \textsf{MIX}$$

The command $\theta; \theta'$ denotes the formal "joining together" of the separate computations θ and θ'. There can be no interactions between them. We will see, in Section 6.2, that the ";" operator allows backtracking-nondeterminism. This motivates the nullary version of the MIX rule:

$$\frac{}{\vdash \{\mathit{fail}\}}\ \textsf{FAIL}$$

The following equivalences hold for ";" and *fail*:

$$\theta_1; \theta_2 = \theta_2; \theta_1 \tag{32}$$

$$\theta_1; (\theta_2; \theta_3) = (\theta_1; \theta_2); \theta_3 \tag{33}$$

$$\mathit{fail}; \theta = \theta \tag{34}$$

See Section 6.2 for applications of these constructs.

5 Properties of Directional Logic Programs

In this section, we briefly mention the important theoretical properties of the type system presented here.

First, we establish the *linearity* property. This is done in a somewhat roundabout fashion. We define functions $V(\cdot)$ which give the free variables of terms, patterns and commands as *multisets* of variables. The definition is "conditional".

Then we show, by induction on type derivations, that the condition is satisfied and, at the same time, the derivable sequents are linear and acyclic. The free variables of a term are defined by:

$$V(x) = \{x\}$$

$$\begin{array}{rlrl} V(()) &= \emptyset & V([\,]) &= \emptyset \\ V((t,u)) &= V(t) \cup V(u) & V([t,u]) &= V(t) \triangle V(u) \\ V(\mathit{inl}\, t) &= V(t) & V(\langle\{\theta\}t, \{\phi\}u\rangle) &= V(\theta) \triangle V(t) \\ V(\mathit{inr}\, t) &= V(t) & &= V(\phi) \triangle V(u) \end{array}$$

where $V_1 \triangle V_2 = (V_1 \setminus V_2^{\perp}) \cup (V_2 \setminus V_1^{\perp})$. The definition involves a condition for the case of lazy pairs. Free variables of patterns are defined by dualization: $V(p) = V(p^{+})$. For commands, we have the definition:

$$\begin{array}{rl} V(\mathit{succ}) &= \emptyset \\ V(t_1 \Leftrightarrow t_2) &= V(t_1) \cup V(t_2) \\ V(\theta_1, \theta_2) &= V(\theta_1) \triangle V(\theta_2) \\ V(\nu x.\, \theta) &= V(\theta) \qquad \text{if } x, x^{\perp} \notin V(\theta) \\ V(P\, t) &= V(t) \end{array}$$

Theorem 1. *Suppose*

$$p_1 : A_1, \ldots, p_n : A_n \vdash \{\theta\} t_1 : B_1, \ldots, t_m : B_m$$

is a derivable sequent. Then, for all p_i $(i = 1, n)$, θ, t_j $(j = 1, m)$,

- $V(\cdot)$ *is well-defined.*
- linearity*: $V(\cdot)$ is a set,* i.e.*, has at most one occurrence of a variable x.*
- acyclicity*: $V(\cdot)$ does not have both x and $x^{\perp}$ for any variable x.*
- $V(\theta) \triangle V([t_1, \ldots, t_m]) = V((p_1, \ldots, p_n))^{\perp}$.

We have that evaluation preserves types (called "semantic soundness" in [40]).

Theorem 2 (subject reduction). *If $\Gamma \vdash \{\theta\}$ is a derivable sequent and $\theta \longrightarrow^{*} \theta'$ by the reduction system, then $\Gamma \vdash \{\theta'\}$ is derivable.*

Note that this means, in particular, that the evaluation maintains linearity and acyclicity.

The confluence or Church-Rosser property follows by a proof similar to Abramsky's [2]:

Theorem 3 (confluence). *If $\theta \longrightarrow^{*} \theta_1$ and $\theta \longrightarrow^{*} \theta_2$, then there exists a command ϕ such that $\theta_1 \longrightarrow^{*} \phi$ and $\theta_2 \longrightarrow^{*} \phi$.*

This means that the results of evaluation are independent of the evaluation order. In conventional terminology, the results are independent of the "selection function".

We now show the correspondence between directional logic programs and relational logic programs. The correspondence is defined by a series of translations

$(\cdot)^\circ$ from the directional type system to Typed Prolog. For types, we have the translation:

$$
\begin{aligned}
(A^\perp)^\circ &= A^\circ \\
\mathbf{1}^\circ \;=\; \perp^\circ &= \mathit{Unit} \\
(A \otimes B)^\circ \;=\; (A \parallel B)^\circ &= A^\circ \times B^\circ \\
(A \oplus B)^\circ \;=\; (A \,\&\, B)^\circ &= A^\circ + B^\circ
\end{aligned}
$$

The translation of a directional term t is a pair (t', ϕ) which we write as $(t' \mathsf{where} \phi)$, and, when ϕ is empty, abbreviate to t'. In the following equations, assume $t^\circ = (t' \text{ where } \phi_1)$ and $u^\circ = (u' \text{ where } \phi_2)$.

Terms :

$$
\begin{aligned}
x^\circ \;=\; (x^\perp)^\circ &= x \\
()^\circ \;=\; [\,]^\circ &= () \\
(t, u)^\circ \;=\; [t, u]^\circ &= (t', u') \text{ where } \phi_1, \phi_2 \\
(\mathit{inl}\ t)^\circ &= \mathit{inl}\ t' \text{ where } \phi_1 \\
(\mathit{inr}\ t)^\circ &= \mathit{inr}\ t' \text{ where } \phi_1 \\
\langle \{\theta_1\} t, \{\theta_2\} u \rangle^\circ &= x \text{ where } (x = \mathit{inl}\ t', \phi_1, \theta_1^\circ;\ x = \mathit{inr}\ u', \phi_2, \theta_2^\circ)
\end{aligned}
$$

Patterns :

$$p^\circ = (p^+)^\circ$$

Commands:

$$
\begin{aligned}
\mathit{succ}^\circ &= \text{the empty formula} \\
(t \Leftrightarrow u)^\circ &= t' = u', \phi_1, \phi_2 \\
(\theta_1, \theta_2)^\circ &= \theta_1^\circ, \theta_2^\circ \\
(\nu x.\, \theta)^\circ &= \exists x.\, \theta^\circ \\
(P\, t)^\circ &= P\, t',\ \phi_1
\end{aligned}
$$

Finally, to translate procedure definitions

$$
\begin{aligned}
&P : \mathit{proc}\ A \\
&P\, p = \theta
\end{aligned}
$$

we assume that p is a pattern made of distinct variables and tupling constructs "$(\ldots)$" and "$[\ldots]$". (If it is not originally in that form, it can be easily converted to that form using the equivalence (23) backwards). The translation of the procedure definition is

$$
\begin{aligned}
&P : \mathit{pred}\ A^\circ \\
&P\, p^\circ = \theta^\circ
\end{aligned}
$$

It is rather obvious that:

Lemma 4. *The translation of a well-typed directional program is a well-typed Typed Prolog program.*

Next, we show that the translation preserves semantics. Let $\longrightarrow_T$ denote the Typed Prolog reduction relation defined by (1–11) and $\longrightarrow_D$ denote the reduction relation of directional programs defined in Fig. 2. Then, we have:

Theorem 5. *Let $\mathcal{P}$ be a directional logic program and $\Gamma \vdash \{\theta\}$ a well-typed command with respect to $\mathcal{P}$. (So, "$\Gamma^\circ \vdash \theta^\circ$ Formula" is a well-typed goal with respect to $\mathcal{P}^\circ$). Then, whenever $\theta \longrightarrow_D^* \theta'$, $\theta^\circ \longrightarrow_T^* \theta'^\circ$.*

The converse does not hold. There are fewer reductions in directional setting than in the relational setting. Essentially, a consumer of a variable can "fire" in Typed Prolog even without the variable being bound, whereas in the directional program it is made to wait. Thus, the best we can aim for is the following result. Call a type *lazy* if it has an unnegated & constructor. A command $\Gamma \vdash \{\theta\}$ is said to be lazy if $V(\Gamma)$, which is the same as $V(\theta)^{\perp}$, contains a variable of a lazy type.

Theorem 6. *Let $\mathcal{P}$ be a directional logic program and $\Gamma \vdash \{\theta\}$ a non-lazy command with respect to $\mathcal{P}$. (We have that "$\Gamma^\circ \vdash \theta^\circ$ Formula" is a well-typed goal with respect to $\mathcal{P}^\circ$). Whenever $\theta^\circ \longrightarrow_T^* \phi$ and the reduction of θ° is convergent, there exist θ', ϕ' such that $\theta \longrightarrow_D^* \theta'$, $\theta'^\circ = \phi'$ and $\phi \longrightarrow_T^* \phi'$.*

Since Typed Prolog can fire the consumer of a variable even without the variable being bound, to simulate the reduction in the directional setting, we have to first evaluate the producer of the variable. If $\phi \longrightarrow_T^* \phi'$ represents the evaluation of the producer, then we can represent the combination $\theta^\circ \longrightarrow_T^* \phi \longrightarrow_T^* \phi'$ in the directional reduction system.

We should point out, however, that there are reduction rules, called *commutative* reductions, which achieve the effect of firing consumers. See [20, 17]. In our setting, commutative reductions are expressed as:

$$\begin{array}{rcl} u[\langle\{\theta_1\}t_1, \{\theta_2\}t_2\rangle] & \longrightarrow & \langle\{\theta_1\}u[t_1], \{\theta_2\}u[t_2]\rangle \\ \langle\{\theta_1\}t_1, \{\theta_2\}t_2\rangle \Leftrightarrow u,\ \phi & \longrightarrow & \langle\{\theta_1, \phi\}t_1, \{\theta_2, \phi\}t_2\rangle \Leftrightarrow u \end{array}$$

These rules transport computations into lazy pairs (equivalently, case branches) so that they can proceed even without a choice being made. Note, in particular, the similarity between the second reduction and the backtracking rule (11). Thus, directional logic programming *can* be made to perform all the reductions available in the relational setting. But, it is not our program to do so.

6 Extensions

In this section, we briefly review various extensions and pragmatic concerns.

6.1 Repetition

The language considered so far is completely linear (except for procedures). Each value has a single producer and a single consumer. However, we do need to use some values multiple times. This is tricky business. As noted in [25, 28], allowing multiple consumers implicitly allows multiple producers as well because the consumed value may have an embedded acceptor. But, multiple producers lead to inconsistencies.

Linear logic includes a safe treatment of multiply usable values. However, the constructs involved in this treatment are extremely rich and powerful. In particular, they go beyond Horn clause logic. It is not yet clear what is the best

way to incorporate these features into directional logic programming. For the sake of completeness, we briefly indicate the features provided by linear logic and point to the issues they raise.

The type "$!A$" (read "of course" A) denotes computations that may be discarded or duplicated (in addition to being used linearly). If such computations use other computations, it is easy to see that the latter must in turn be discardable/duplicatable. For, in discarding/duplicating the final computation, we are also discarding/duplicating its subcomputations. This explains the intricacies involved in the $!\mathcal{R}$ rule below:

$$!\mathcal{R} \quad \frac{p_1 : !B_1, \ldots, p_k : !B_k \vdash \{\theta\}\, t : A}{x_1 : !B_1, \ldots, x_k : !B_k \vdash \{succ\}\, (![p_1 \Leftarrow x_1, \ldots, p_k \Leftarrow x_k]\{\theta\}t) : !A}$$

$$!\mathcal{L} \quad \frac{\Gamma, p : A \vdash \{\theta\}\, \Delta}{\Gamma, !p : !A \vdash \{\theta\}\, \Delta}$$

$$!\mathcal{W} \quad \frac{\Gamma \vdash \{\theta\}\, \Delta}{\Gamma, _ : !A \vdash \{\theta\}\, \Delta} \qquad !\mathcal{C} \quad \frac{\Gamma, p_1 : !A, p_2 : !A \vdash \{\theta\}\, \Delta}{\Gamma, p_1 @ p_2 : !A \vdash \{\theta\}\, \Delta}$$

The $!\mathcal{R}$ rule "promotes" a linear computation to a discardable/duplicatable computation. The term constructed by such promotion encapsulates the entire computation, including the command θ, and introduces fresh variables for the interface $x_1, \ldots, x_k$. (The reason for this complexity becomes clear once we look at the reduction semantics below). The rule $!\mathcal{L}$ allows an $!A$-typed value to be used once (linearly), $!\mathcal{W}$ allows it to be discarded and $!\mathcal{C}$ allows it to be duplicated. The reduction semantics of these constructs is as follows:

$$!q \Leftarrow (![p_1 \Leftarrow t_1, \ldots]\{\theta\}u) \longrightarrow p_1 \Leftarrow t_1, \ldots, \theta, q \Leftarrow u \tag{35}$$

$$_ \Leftarrow (![p_1 \Leftarrow t_1, \ldots]\{\theta\}u) \longrightarrow _ \Leftarrow t_1, \ldots \tag{36}$$

$$\begin{aligned} q_1 @ q_2 \Leftarrow (![p_1 \Leftarrow t_1, \ldots]\{\theta\}u) \longrightarrow\ & y_1 @ z_1 \Leftarrow t_1, \ldots, \\ & q_1 \Leftarrow (![p_1 \Leftarrow y_1, \ldots]\{\theta\}u), \\ & q_2 \Leftarrow (![p_1 \Leftarrow z_1, \ldots]\{\theta\}u) \end{aligned} \tag{37}$$

Note that, in reductions (36) and (37), the demands to discard and duplicate a promoted computation are propagated to its inputs. The complexity of the promotion construction is due, in part, to the need for such propagation.

Linear logic's promotion construct takes us beyond Horn clause logic programming. Since entire computations can be discarded (reduction 36), lazy evaluation is required in computing with ! types. Similarly, reduction (37) requires entire computations to be explicitly copied. The effect of copying can be achieved by sharing in some instances, but not always. For example, consider $x_1 @ x_2 \Leftarrow d$ where d is a difference list. Since d has an internal local variable, it cannot be simply shared from x_1 and x_2. Instead, separate copies of d must be bound to the two variables. It seems that copying can be avoided by restricting the use of "!" to "positive" types, *i.e.*, types built from primitive types, $\otimes$, $\mathbf{1}$ and $\oplus$. Similarly, the lazy evaluation may be avoided by requiring θ to be empty in the promotion rule. All of this needs further investigation.

Example 6. To be able to "dump" the stack of Example 5, we must require that all its elements are discardable. Then, we can define *dump* as:

```
dump : proc (List !α ⊗ ⊥)
dump(nil(), []) ← succ
dump(cons(__, st), dummy) ← dump(st, dummy)
```

One may wonder if we could discard the stack directly, instead of discarding its elements individually. It is possible to do so by using a type of "nonlinear" lists:

```
type NList α = nil 1 ⊕ cons (!α ⊗ !(NList α))
```

Then, *stackloop* can be redefined as follows:

```
stackloop : proc (Msgs !α ⊗ !(NList α))

stackloop(nil[], nst) ←
    __ ⇐ nst
stackloop(push(x, ms'), nst) ←
    stackloop(ms', ![x' ⇐ x, nst' ⇐ nst] cons(x', nst'))
stackloop(pop[x⊥, ms'], !st) ←
    case st of
       nil() ⇒ error
     | cons(y, nst') ⇒ x⊥ ⇐ y, stackloop(ms', nst')
```

■

6.2 Nondeterminism

Just as we introduced dual type constructors for data structures in Section 4.3, we can introduce a dual constructor for "!". This is written as "?" and pronounced "why not". The type $?A$ denotes computations whose acceptors can be multiply used. Thus, $?A$-typed computations are nondeterministic: they denote multiple A-typed values. The constructions for ?-types are obtained by extending the dualizaton maps:

$$\begin{aligned} (![p_1 \Leftarrow t_1, \ldots]\{\theta\}u)^- &= (?[p_1 \Leftarrow t_1, \ldots]\{\theta\}u^-) \\ (!p)^+ &= ?p^+ \\ _^+ &= noval \\ (p_1 @ p_2)^+ &= p_1^+ \ or \ p_2^+ \end{aligned}$$

For example, we can define a procedure for producing all the members of a list as a nondeterministic value as follows:

```
members : proc ((?α)⊥ ⊗ List α)
members(x⊥, nil()) ←
    x⊥ ⇔ noval, fail
members(x⊥, cons(y,ys)) ←
    x⊥ ⇔ (?y) or x', (succ; members(x'⊥, ys))
```

Note that the goal `member(x⊥, cons(1,cons(2,nil())))` reduces to the following command:

```
x⊥ ⇔ ?1 or x1, (succ;
                 x1⊥ ⇔ ?2 or x2, (succ;
                                   x2⊥ ⇔ noval, fail))
```

which is nothing but a representation of the backtracking behaviour.

6.3 Higher-order features

If we add unrestricted "of course" types, the language becomes higher-order. The type *proc* A, denoting procedures that accept A-typed arguments, is essentially equivalent to the type $!A^{\perp}$. Consider the notations $\varepsilon p : A.\, \theta$ for procedure abstraction and $t_1\, t_2$ for procedure application. These can be defined as

$$\begin{aligned} \varepsilon p : A.\, \theta &\stackrel{\text{def}}{=} ![\bar{x} \Leftarrow \bar{x}]\{\theta\}p^{+} \\ t_1\, t_2 &\stackrel{\text{def}}{=} !P \Leftarrow t_1, P \Leftrightarrow t_2 \end{aligned}$$

Therefore, procedure abstraction can be made a legitimate term and treated as a first-class value. Note that adding higher-order features to directional logic programming does not involve higher-order "unification" as in Lambda Prolog [38]. We have procedure application, but not equality testing of procedures.

6.4 Conditions and functions

We have deliberately avoided introducing functions into the language to keep it simple and "logic programming-like". However, it is clear that at least one kind of function is necessary, *viz.*, condition. In our procedural view of logic programs, the unification operator can only be used to bind variables. It cannot be used to test for the equality of two values. (See [25] for a similar sentiment). In fact, since the language is higher-order, a universal equality test is not possible. So, we need at least a type `Bool` and a function `=` : $\alpha \otimes \alpha \multimap$ `Bool` where α is restricted to testable types. One quickly sees the need for defining new conditions. So, why not introduce a general function abstraction construct:

$$\multimap \mathcal{R} \quad \frac{\Gamma, p : A \vdash \{\theta\}\, t : B}{\Gamma \vdash \{\theta\}\, \lambda p{:}\, A.\, t : A \multimap B} \qquad \text{if } V(\Gamma) \cap V(p) = \emptyset$$

In fact, using the definition $A \multimap B = A^{\perp} \parallel B$, $\lambda p{:}\, A.\, t$ is equivalent to $[p^{+}, t]$. Function application $f\, t$ is equivalent to $\{f \Leftrightarrow (t, x^{\perp})\}x$. One can follow the ideas of [44] to make the language of terms as rich as the language of commands.

6.5 Sequencing

The language described here is inherently concurrent. So, it is not translatable to Prolog. There are two possible approaches. Notice that the only place where dependencies crop up in the operational semantics is in the reduction rules for the additive type constructors. One can perform static analysis on programs to determine if all such dependencies are in the forward direction. A sequential evaluation would then be complete.

A second possibility is the following: There exists a translation from the language presented here into the functional language of [44]. (Both the languages are computational interpretations of the same logic). This gives a sequential implementation of some sort. It would still not be translatable into sequential Prolog because the translation into the functional language uses suspensions (lazy evaluation) to simulate concurrency.

7 Related work

7.1 Distributed logic programming

A significant class of concurrent logic programming languages evolved from Hirata's proposal to make logic variables linear and directed [25, 28, 45]. The motivation behind these restrictions is the same as that of directional logic programming. Many of the technical details also bear a close similarity. For example, Janus appears to be an untyped (*i.e.*, dynamically typed) version of the language presented in Sections 4.1 and 4.2. Patterns and terms correspond to "ask" and "tell" terms respectively. Even our $x^\perp$ notation appears in Janus as the related "!" constructor: permission to tell a constraint on a variable. However, an important difference between $x^\perp$ and $!x$ must be noted. Whereas $x^\perp$ notation is merely a syntactic annotation (nothing would change even if we chose to write $x^\perp$ as merely x), the $!x$ annotation is an integral part of the operational semantics of Janus. Essentially, some of the type checking is done at run-time. Moreover, since the dual data structures of Section 4.3 are not present in Janus, cyclic deadlocks are possible. From our point of view, all the concerns we raise about relational logic programming are present for distributed logic programming with an even greater force. How does one specify the behaviour of a program? The type system presented here makes a contribution in this regard.

7.2 Computational interpretation of linear logic

Our work draws considerable inspiration from Abramsky's computational interpretation of linear logic [2]. The reduction semantics in Fig. 2 is essentially a notational variant of the linear chemical abstract machine defined there. (For example, Abramsky writes our $t_1 \Leftrightarrow t_2$ as $t_1 \perp t_2$). The main difference is that we use symmetric sequents in place of his right sided sequents. This leads to a more accessible notation and gives a direct correspondence with the notation of logic programming.

In separate work, carried out independently from ours, Abramsky, Jagadeesan and Panangaden showed that the proofs of linear logic can be interpreted in a `cc` language. (See [1]). In addition, Lafont [30] seems to foresee many of our ideas.

8 Conclusion

We have given a typed foundation for logic programs with input-output directionality. Our type system combines (what were previously called) types and modes and provides a framework for writing safe logic programs with good algorithmic properties. In addition, this work forms a concrete computational interpretation of Girard's linear logic based on the Curry-Howard correspondence.

The proposed type system extends "simple" modes with facilities for embedding outputs in input arguments and *vice versa*. It provides a theoretical basis for previous extensions of the same nature made in languages like Parlog and MU-Prolog. It also gives a safe type system for distributed logic programming languages.

On the theoretical front, it brings logic programming into the mainstream of typed languages by interpreting logic programs as proofs of a constructive logic. In this respect, it is closely related to the chemical abstract machine and continuation-based interpretations of linear logic. Working out the precise connections with these other interpretations should enrich our understanding of logic programs as well as the other paradigms.

Acknowledgements I am indebted to Dick Kieburtz and Boris Agapiev for introducing me to the fascinating world of linear logic. The work of Abramsky [2] formed the impetus for this work and many of his ideas form an integral part of this type system. Discussions with Peter O'Hearn, Radha Jagadeesan, Sam Kamin, Dale Miller and Phil Wadler helped clarify many issues I failed to notice. In particular, Phil Wadler persuaded me to use the $x^{\perp}$ notation for dual occurrences of variables which seem to clarify the presentation greatly.

References

1. S. Abramsky. Computational interpretation of linear logic. Tutorial Notes, International Logic Programming Symposium, San Diego, 1991.
2. S. Abramsky. Computational interpretations of linear logic. Research Report DOC 90/20, Imperial College, London, Oct 1990. (available by FTP from theory.doc.ic.ac.uk; to appear in *J. Logic and Computation*).
3. J.-M. Andreoli and R. Pareschi. Logic programming with linear logic. In P. Schroeder-Heister, editor, *Extensions of Logic Programming*, (Lect. Notes in Artificial Intelligence). Springer-Verlag, Berlin, 1991. (LNAI).
4. A. Asperti, G. L. Ferrari, and R. Gorrieri. Implicative formulae in the "proofs as computations" analogy. In *Seventeenth Ann. ACM Symp. on Princ. of Program. Lang.*, pages 59–71. ACM, 1990.

5. G. Berry and G. Boudol. The chemical abstract machine. In *Seventeenth Ann. ACM Symp. on Princ. of Program. Lang.*, pages 81–94. ACM, 1990.
6. C. Brown and D. Gurr. A categorical linear framework for petri nets. In *Fifth Ann. Symp. on Logic in Comp. Science*, pages 208–218. IEEE, June 1990.
7. K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.
8. K. L. Clark and S. Gregory. Parlog: A parallel logic programming language. Research Report DOC 83/5, Imperial College of Science and Technology, London, May 1983.
9. 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.
10. R. L. Constable, *et. al.* *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, New York, 1986.
11. H. B. Curry and R. Feys. *Combinatory Logic.* North-Holland, Amsterdam, 1958.
12. V. Danos and L. Regnier. The structure of multiplicatives. *Archive for Mathematical Logic*, 28:181–203, 1989.
13. Y. Deville. *Logic Programming: Systematic Program Development.* Addison-Wesley, Wokingham, 1990.
14. A. Filinski. Linear continuations. In *ACM Symp. on Princ. of Program. Lang.*, pages 27–38. ACM, Jan 1992.
15. V. Gehlot and C. Gunter. Normal process representatives. In *Symp. on Logic in Comp. Science*, pages 200–207. IEEE, June 1990.
16. G. Gentzen. *The Collected Papers of Gerhard Gentzen, edited by M. E. Szabo.* North-Holland, Amsterdam, 1969.
17. J.-Y. Girard. Linear logic. *Theoretical Comp. Science*, 50:1–102, 1987.
18. J.-Y. Girard. *Proof Theory and Logical Complexity*, volume 1. Bibliopolis, Napoli, 1987.
19. J.-Y. Girard. Towards a geometry of interaction. In J. W. Gray and A. Scedrov, editors, *Categories in Computer Science and Logic*, pages 69–108, Boulder, Colorado, June 1987. American Mathematical Society. (Contemporary Mathematics, Vol. 92).
20. J-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types.* Cambridge Univ. Press, Cambridge, 1989.
21. S. Gregory. *Parallel Logic Programming in PARLOG: The Language and its Implementation.* Addison-Wesley, Reading, Mass., 1987.
22. C. Gunter and V. Gehlot. Nets as tensor theories. Technical Report MS-CIS-89-68, University of Pennsylvania, Oct 1989.
23. M. Hanus. Horn clause programs with polymorphic types: Semantics and resolution. *Theoretical Comp. Science*, 89:63–106, 1991.
24. J. Harland and D. Pym. The uniform proof-theoretic foundation of linear logic programming. Technical Report ECS-LFCS-90-124, University of Edinburgh, Nov 1990.
25. M. Hirata. Programming language Doc and its self-description, or $x = x$ considered harmful. In *Third Conference of Japan Society of Software Science and Technology*, pages 69–72, 1986.
26. J. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. In *Sixth Ann. Symp. on Logic in Comp. Science.* IEEE, 1991.

27. W. A. Howard. The formulae-as-types notion of construction. In J. R. Hindley and J. P. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, New York, 1980.
28. A. Kleinman, Y. Moscowitz, A. Pnueli, and E. Shapiro. Communication with directed logic variables. In *Eighteenth Ann. ACM Symp. on Princ. of Program. Lang.*, pages 221–232. ACM, 1991.
29. R.A. Kowalski. Algorithm = logic + control. *Communications of the ACM*, 22:424–431, 1979.
30. 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.
31. Y. Lafont. Interaction nets. In *Seventeenth Ann. ACM Symp. on Princ. of Program. Lang.*, pages 95–108. ACM, Jan 1990.
32. 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.
33. J.-L. Lassez and M. J. Maher. Closures and fairness in the semantics of programming logic. *Theoretical Computer Science*, pages 167–184, May 1984.
34. P. Lincoln. Linear logic. *SIGACT Notices*, 23(2):29–37, 1992.
35. J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, second edition, 1987.
36. N. Marti-Oliet and J. Meseguer. From Petri nets to linear logic. *Math. Structures in Comp. Science*, 1:69–101, 1991.
37. P. Martin-Löf. Constructive mathematics and computer programming. In L. J. Cohen, J. Los, H. Pfeiffer, and K.-P. Podewski, editors, *Proc. Sixth Intern. Congress for Logic, Methodology and Philosophy of Science*, pages 153–175. North-Holland, 1982.
38. D. A. Miller and G. Nadathur. Higher-order logic programming. In *Intern. Conf. on Logic Programming*, 1986.
39. R. Milner. *A calculus for communicating systems.* (LNCS). Springer-Verlag, 1979.
40. A. Mycroft and R. A. O'Keefe. A polymorphic type system for Prolog. In *Logic Programming workshop*, pages 107–122, Universidade Nova de Lisboa, 1983.
41. L. Naish. Automating control of logic programs. *J. Logic Programming*, 2(3):167–183, 1985.
42. U. S. Reddy. Transformation of logic programs into functional programs. In *Intern. Symp. on Logic Programming*, pages 187–197. IEEE, 1984.
43. 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.
44. U. S. Reddy. Acceptors as values: Functional programming in classical linear logic. Preprint, Univ. Illinois at Urbana-Champaign, Dec 1991.
45. V. A. Saraswat, K. Kahn, and J. Levy. Janus: A step towards distributed constraint programming. In S. Debray and M. Hermenegildo, editors, *North American Conf. on Logic Programming*, pages 431–446. MIT Press, Cambridge, 1990.
46. V. A. Saraswat and M. Rinard. Concurrent constraint programming. In *Seventeenth Ann. ACM Symp. on Princ. of Program. Lang.*, pages 232–245. ACM, 1990.
47. A. Scedrov. A brief guide to linar logic. *Bulletin of the European Assoc. for Theoretical Computer Science*, 41:154–165, June 1990.

48. E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):412–510, 1989.
49. E. Y. Shapiro. A subset of concurrent prolog and its interpreter. Technical Report TR-003, ICOT- Institute of New Generation Computer Technology, January 1983.
50. L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, Cambridge, Mass., 1986.
51. G. Takeuti. *Proof Theory*, volume 81. North-Holland, Amsterdam, 1987. Second edition.
52. S. Thompson. *Type Theory and Functional Programming*. Addison-Wesley, Wokingham, England, 1991.

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