Acceptors as Values

Functional Programming in Classical Linear Logic

(Technical Summary)

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

December 9, 1991

Abstract

Girard's linear logic has been previously applied to functional programming for performing state-manipulation and controlling storage reuse. These applications only use *intuitionistic* linear logic, the subset of linear logic that embeds intuitionistic logic. Full linear logic (called *classical* linear logic) is much richer than this subset. In this paper, we consider the application of classical linear logic to functional programming. The negative types of linear logic are interpreted as denoting *acceptors*. An acceptor is an entity which takes an input of some type and returns no output. Acceptors generalize continuations and also single assignment variables, as found in data flow languages and logic programming languages. The parallel disjunction operator allows such acceptors to be used in a nontrivial fashion. Finally, the "why not" operator of linear logic gives rise to nondeterministic values. We define a typed functional language based on the these ideas and demonstrate its use via examples. The language has a reduction semantics that generalizes typed lambda calculus, and satisfies strong normalization and Church-Rosser properties.

Keywords Linear logic, propositions-as-types isomorphism, sequent calculus, continuations, single assignment variables, logic variables, acceptors, bivalent types, nondeterminism, strong normalization, confluence.

1 Introduction

The correspondence between typed functional programming and intuitionistic logic is now well known [1, 8, 15, 20, 27, 44]. In general, any constructive logic can be interpreted as a programming language by treating its propositions as types and their proofs (constructions) as programs. The type systems of usual functional languages thus correspond to (the propositional fragment of) first-order intuitionistic logic and polymorphic functional languages correspond to (the propositional fragment of) second-order intuitionistic logic. Predicate logic concepts too have been used to give types to abstract data types and modules [26, 29].

In a surprising *tour de force*, Girard [12] proceeded to give a "constructivization" of classical logic. The essential trick is to control the use of the structural rules, *weakening* and *contraction*, which allow a hypothesis (or a variable) to be used an arbitrary number of times in classical or intuitionistic logic. The resulting logic is termed *linear logic* since, by default, a variable can be used exactly once. The most remarkable property of linear logic is that it contains an involutive negation operator: $A^{\perp\perp} \cong A$, i.e., the constructions for "not not A" and "A" are the same.

By the propositions-as-types (and proofs-as-programs) correspondence, the constructions for linear logic have a computational interpretation as programs. The first to be studied is the so-called *intuitionistic linear logic*, the subset of linear logic that is used to embed intuitionistic logic. It has been used to devise type systems for functional languages with control over storage reuse and state [19, 24, 47, 46, 45]. However, intuitionistic linear logic is much weaker than full linear logic (called *classical* linear logic). In particular, it lacks the negation operator, the "parallel" disjunction operator and the indeterminacy "why not" operator. Girard himself has suggested *concurrent* programming as the appropriate computational interpretation for linear logic [12, 14, 13] and other researchers have pursued this line [1, 2, 3, 5, 11, 23, 28]. The only application of classical linear logic to functional programming has been by Filinski [10] who uses negative types to model continuations, following the work of [9, 17, 30, 31].

In the present work, we propose a new interpretation: linear negation as representing the types of *variables* (or, more accurately, the *acceptor* part of variables). Conventionally, a variable is a place holder of values, e.g., in $\lambda x.\, x * x$, x is the place holder for the input of the function. When a function is applied, two steps are carried out. First, the rule of the function is instantiated, in the process creating a new place-holder for the variable which is different from all other place-holders. Second, the place-holder is immediately filled with the argument of the function. Thus, place-holders in functional languages are rather short-lived. In contrast, certain languages allow a more flexible treatment of variables. In data-flow languages like Id [32, 22], a place-holder can be created and filled at a later time. In the meantime, any computation requiring a value from the place-holder is made to wait. Unfortunately, this treatment is somewhat *ad hoc* in that failure to fill a value or filling multiple values can create problems. Logic programming languages [7, 41, 48] similarly have a notion of "logic variables" whose instantiation can be delayed. Efforts have also been made to extend functional languages with logic variables [25, 34]. Our recent study of imperative programs [43, 42] suggests that *references* of imperative programming are similar to variables in many ways and they too exhibit the capacity for delayed instantiation. Such delayed instantiation provides considerable expressive power allowing the expression of many elegant and efficient algorithms. The literature on data

flow languages, logic programming and imperative programming contains several examples of this expressive power.

The linear logic decomposition of the function type $A \multimap B$ into $A^{\perp} \parallel B$ directly suggests a connection between variables and negative types.[1] In fact, this connection is already made by Girard who suggests terms like "acceptors", "receivers" and "questions" to refer to the concept of semantic variables. Other possible metaphors include "holes" and "input ports". The suggestion that negative types represent variables is implicitly present in Abramsky's linear calculus as well [1]. With regard to the continuations interpretation of negative types, it seems that continuations generalize to variables when parallel disjunction is added. We discuss this aspect in more detail in Section 2.

Our presentation here involves a type system called *bivalent type system* which captures (the propositional fragment of) full linear logic, but in an intuitionistic format.[2] By intuitionistic format, we mean sequents of the form $\Gamma \vdash A$ and $\Gamma \vdash$. Expressing linear logic in intuitionistic format is tricky because some of its connectives do not have natural formulations in this format.[3] However, it is precisely the intuitionistic format which allows the logic to be interpreted as a functional language. Using right-sequents as in [1, 12] would give an interpretation as a concurrent language while using left-sequents would lead to a logic programming language.

Bivalent types explained A linear function space $A \multimap B$ contains functions which accept inputs of type A and produce results of type B (using the input exactly once in the process). Every such function can also be viewed as belonging to the type $A \times B^{\perp} \multimap \perp$. In this new interpretation, the function accepts a value of type A and an acceptor (place-holder, hole or generalized continuation) for values of type B, and produces a contradiction (using both the value and the variable exactly once in the process). Since the acceptor *must* be used and the only way it can be used is by assigning a value, we have the effect that a value of type B has been produced. While the form $A \multimap B$ is the type of a conventional function, the form $A \times B^{\perp} \multimap \perp$ is suited for a predicate in a *directional logic program.*[4] Suppose that, in using the function, we provide the second component, i.e., B-acceptor, but not the first. Then, what we are left with is $A \multimap \perp$ which is equivalent to $A^{\perp}$. Thus, another view of the same function is with the type $B^{\perp} \multimap A^{\perp}$. Producing a result of type $A^{\perp}$ has the same effect as accepting an input of type A because the hole represented by $A^{\perp}$ must be eventually filled elsewhere (just as the input of type A must be supplied elsewhere). In our formulation, the three types are isomorphic and functions can be converted from one form to the other.

The fact that a single function type can be represented in so many different ways goes to show that function types should be defined in terms of more primitive type constructors.

[1] For typographical reasons, we use the symbol "$\parallel$" to denote the parallel disjunction (*par*) operator, which is otherwise written as an upside down "&".

[2] "Bivalent" refers to the fact that we have positive as well as negative types. Classical logic is often called bivalent logic because propositions have precisely two truth values. We make an intentional allusion to this reference, but note that constructive propositions are more than mere truth values.

[3] In fact, the largest logic that can be naturally formulated in the intuitionistic format is the intuitionistic logic itself.

[4] Traditional logic programming exemplified by Prolog makes no explicit distinction between variables and values (inputs and outputs). But, directional logic programs introduced in [37, 35] and concurrent logic languages like Parlog [16] make such distinctions.

Indeed, linear logic defines $A \multimap B$ is $A^{\perp} \parallel B$. The constructor "$\parallel$" gives the type of "parallel pairs". The components of such pairs are produced using the same context, but they must be consumed separately. This is the dual of the $\times$ constructor which gives "independent pairs"—pairs whose components are produced separately, but consumed using the same context.

The ability to treat acceptors as first-class values allows one to decouple the creation of a variable from the assignment of a value. To see this, consider an identity function of type $A \multimap A$. By definition, the type is equivalent to $A^{\perp} \parallel A$, i.e., the identity function is a parallel pair with two components: an A-acceptor and an A-value. Moreover, the two components are linked internally so that if a value is assigned to the acceptor component of the pair, the same value is visible via the A component of the pair. This is essentially what a variable is. By representing the function type as $A^{\perp} \parallel A$, we can separate the two components and treat each one independently. In other words, we can assign a value to the variable in one part of the program and use it in another part of the program.[5]

To show how variables and acceptors may be used, we consider an example. A *difference list* is a parallel pair consisting of a list l and a list acceptor t such that the acceptor t is linked to some cell accessible from l. (These are directional versions of traditional difference lists familiar from logic programming [6]). This type may be defined in the bivalent type system as follows:

$$\begin{array}{lcl} \mathit{type}\ \mathit{List} & = & \mathit{nil} +:: (\mathit{Int}, \mathit{List}) \\ \mathit{type}\ \mathit{DList} & = & \mathit{List} \parallel \mathit{List}^{\perp} \end{array}$$

To append two difference lists we may use a function of the following form:[6]

$$\begin{array}{l} \mathit{append} : \mathit{DList} \times \mathit{DList} \times \mathit{DList}^{\perp} \multimap \perp \\ \mathit{append}\ ([l_1, t_1], [l_2, t_2], (l', t')) = \{t_1 := l_2, l' := l_1, t_2 := t'\} \end{array}$$

The first argument $[l_1, t_1]$ is an input, but t_1 is an output embedded within the input. Similarly, for the second argument. The third argument is an output, represented by accepting an acceptor for the output. In linear logic, we have the following equality by definition: $(\mathit{List} \parallel \mathit{List}^{\perp})^{\perp} = \mathit{List}^{\perp} \times \mathit{List}$. Thus, the acceptor (l', t') is an ordinary (independent) pair. To produce $\perp$ from these inputs, we must essentially use all the values and acceptors. This is done by assigning the head of the second *DList* to the tail of the first and the remaining components to l' and t' appropriately.

It is also possible to express *append* as function with the conventional type $\mathit{DList} \multimap \mathit{Dlist} \multimap \mathit{DList}$. But, this cannot be done using conventional functional notation because functional notation does not support par-types. We use a new binding operator ϵ to encapsulate a proof by contradiction, i.e., the production of an output by satisfying an acceptor. This allows:

$$\mathit{append}'\ [l_1, t_1]\ [l_2, t_2] \quad = \quad \epsilon(l', t').\ (t_1 := l_2, l' := l_1, t_2 := t')$$

Outline The rest of the paper is organized as follows. In Section 2, we introduce the bivalent type system. In Section 3, we present a functional programming language based on the type system. Finally, we review some preliminary technical results in Section 4.

[5] Reynolds [38, 39] notes a similar decomposition of variables in imperative programming. It is not yet clear how it relates to the linear logic decomposition presented here.

[6] We are using here notation that is different from the formal language of Section 3 in inessential ways.

2 Bivalent types

Let β range over atomic types and A, B, C over types. The syntax of type terms is

$$A ::= 1 \mid \bot \mid \beta \mid \beta^{\bot} \mid A_1 \times A_2 \mid A_1 \parallel A_2 \mid A \,\&\, B \mid A + B \mid !A \mid ?A$$

The operator $(\cdot)^{\bot}$ is extended to type terms by the following definition

$$\begin{array}{rclrcl} 1^{\bot} & = & \bot & \bot^{\bot} & = & 1 \\ (A \times B)^{\bot} & = & A^{\bot} \parallel B^{\bot} & (A \parallel B)^{\bot} & = & A^{\bot} \times B^{\bot} \\ (A \,\&\, B)^{\bot} & = & A^{\bot} + B^{\bot} & (A + B)^{\bot} & = & A^{\bot} \,\&\, B^{\bot} \\ (!A)^{\bot} & = & ?A^{\bot} & (?A)^{\bot} & = & !A^{\bot} \end{array}$$

$$A^{\bot\bot} = A$$

(Note that we only need one of the two columns above because the other column can be deduced from the equation at the bottom). The function space type is defined as

$$A \multimap B = A^{\bot} \parallel B$$

We first give an intuitive explanation of the various types:

- If A is any type, $A^{\bot}$ (read "A nil") is the type of *acceptors* for values of type A. Such an acceptor should completely consume the value and produce *no output*. (Acceptors generalize continuations as discussed below).

- 1 is the unit type containing a single value denoted by (). There is a single acceptor for 1 which is the only value of type $\bot$. (See below).

- The type $\bot$ (read "perp") has no values worth speaking of. Conceptually, $\bot$ is the type of *commands*, terms which carry out actions without producing any output. In the empty type context (i.e., in the absence of any free variables), the only command is the "empty" command denoted by *noop*. But, in a nontrivial type context, a command consumes all the inputs in the context without producing any output.

  For technical reasons, it is useful to treat *suspensions* of commands as values. Such suspensions are values of type $\bot$. As noted, in the empty type context, *noop* is the only command. So, a suspension of *noop* is the only value of type $\bot$.[7] There is a single acceptor for $\bot$ (which forces the given suspension). We identify it with (). Dually, a suspension of *noop* is the only acceptor for ().

- The values of a primitive type β presumably have representations as constants in the language.

- A value of type $A \times B$ (read "A times B") is a pair consisting of A and B components. The two components are produced using disjoint sets of inputs. Hence, we say that the components are *independent*. An acceptor for $A \times B$ is, by convention, a parallel pair of acceptors $A^{\bot}$ and $B^{\bot}$. (See below). Note that 1 is the identity of $\times$: $1 \times A \cong A \cong A \times 1$.

[7] Thus $\bot$ is also a "unit type". It is a curious aspect of linear logic that it has two kinds of unit types, 1 and $\bot$.

- A value of type $A \parallel B$ (read "A par B") is also a pair, but the components are interpreted as "parallel possibilities". The "excluded middle" $A \parallel A^{\perp}$ is the canonical parallel pair and is always inhabited. All other parallel pairs are produced using this. To consume a value of type $A \parallel B$, both its components must be consumed independently. Thus, an acceptor for $A \parallel B$ is a pair of independent acceptors of types $A^{\perp}$ and $B^{\perp}$. The perp type is the identity for $\parallel$: $\perp \parallel A \cong A \cong A \parallel \perp$.

- A value of type $A \,\&\, B$ (read "A with B") is also a pair, but both its components are produced using the same set of inputs. Thus, only one component is usable. (It is advisable to think of $\&$ as denoting the conjunction of "potentials" of producing A and B). Since only one of the two components is eventually used, the evaluation of the components must be delayed until the point of use. So, we refer to the values of $A \,\&\, B$ as *lazy pairs*. An acceptor for $A \,\&\, B$ must be either an acceptor of the A component or an acceptor of the B component—which is an element of the sum type $A + B$.

- A value of type $A + B$ (read "A plus B") is either a value of type A or a value of type B, tagged to indicate which is the case. An acceptor for $A + B$ must be able to accept both the A and B possibilities (hence, a pair of acceptors), but only one of the two acceptors would ever be used (hence, a lazy pair).

- A value of type $!A$ (read "of course A") is a *persistent* version of an A-typed value. It can be used any number of times (including none at all). A useful view of $!A$ is as the following recursive data type:[8]

$$!A = 1 \,\&\, A \,\&\, (!A \times !A)$$

  An acceptor for $!A$ is a "possibility" of an acceptor for an A-typed value. (See below).

- A value of type $?A$ (read "why not A") is a *possibility* of A-typed values (or an *indeterminate* value). It may denote the possibility of no A-typed value or the possibility of multiple values. Thus, $?A$ may be viewed as the recursive type

$$?A = \perp + A + (?A \parallel ?A)$$

  An acceptor for $?A$ is a persistent acceptor for A. (It must accept each value that the indeterminate term evaluates to).

- A value of type $A \multimap B$ (read "A to B") is a linear function from A-typed values to B-typed values which uses its input exactly once. Such a function can also be thought of as a parallel pair consisting an acceptor $A^{\perp}$ and a value of type B. (Parallel, because the input-acceptor and the output value are produced from the same context). An acceptor for $A \multimap B$ is, therefore, an independent pair with an A-typed value (for the input of the function) and an acceptor for B (to consume the output of the function).

- Given that $A^{\perp}$ is the type of acceptors for A-typed values, an acceptor for $A^{\perp}$ is a value of type A. To accept an acceptor, we must satisfy its request for input.

[8] There is some controversy as to how to interpret this equation. See [10, 24, 40].

Let us say a few words about commands and acceptors. Suppose we have a command θ in a context that includes an input $x : A$. We can "internalize" the input with the command to produce an acceptor of type $A^{\perp}$. This acceptor is written as $\epsilon x.\, \theta$ using a binding operator ϵ.[9] Such an acceptor denotes a "debt" or "deficit" of A, i.e., it requires an A in order to produce "nothing". If a is a term of type A, then the debt is fulfilled by writing the command $(\epsilon x.\theta) := a$. This command is reducible to $\theta[a/x]$ which is what we would have had if we originally provided the term a in place of x.

Acceptors are rather like continuations. In fact, they reduce to continuations in the absence of the $\parallel$ connective. The use of negative types to express continuations has been previously studied by Griffin [17], Filinski [10] and Murthy [30]. Here, $\perp$ is interpreted as the end of an entire computation and, so, the deficit of a value is a continuation. In contrast, the $\parallel$ connective gives rise to the new isomorphism $\perp \parallel \perp \cong \perp$. Intuitively, this means that a parallel combination of two no-value possibilities is the same as a no-value. Thus, $\perp$ just means the end of a local computation which can be combined with other similar local computations. Analogously, our acceptors are local deficits rather than entire continuations.

3 Term language

The context-free syntax of preterms is as follows:

$$
\begin{array}{rcl}
t & \in & \text{Term} \\
\theta & \in & \text{Command} \\
P & \in & \text{Pattern} \\
T & \in & \text{General Term}
\end{array}
$$

$$
\begin{array}{rcl}
t & ::= & x \mid \epsilon x{:}\, A.\, \theta \mid \lambda x{:}\, A.t \mid t_1 t_2 \mid () \mid (t_1, t_2) \mid \texttt{let}\ t\ \texttt{be}\ P\ \texttt{in}\ t \\
 & & \mid parl(t;\ x \Rightarrow t;\ y \Rightarrow \{\theta_2\}) \mid parr(t;\ x \Rightarrow \{\theta_1\};\ y \Rightarrow t) \\
 & & \mid \langle t_1, t_2 \rangle \mid inl(t) \mid inr(t) \mid (\texttt{case}\ t\ \texttt{of}\ inl(x) \Rightarrow t_1 \mid inr(y) \Rightarrow t_2) \\
 & & \mid ![\bar{t}; \bar{x}]t \\
\theta & ::= & noop \mid do(t) \mid t_1 := t_2 \mid \texttt{let}\ t\ \texttt{be}\ P\ \texttt{in}\ \theta \mid par(t;\ x \Rightarrow \theta_1;\ y \Rightarrow \theta_2) \\
 & & \mid (\texttt{case}\ t\ \texttt{of}\ inl(x) \Rightarrow \theta_1 \mid inr(y) \Rightarrow \theta_2) \mid ![\bar{t}; \bar{x}]\theta \mid choose[\bar{t}; \bar{x}](x \Rightarrow \theta) \\
P & ::= & x \mid () \mid (P_1, P_2) \mid \langle P, _ \rangle \mid \langle _, P \rangle \mid !P \mid _ \mid P_1 @ P_2 \\
T & ::= & \theta \mid t
\end{array}
$$

Commands are essentially proofs of $\perp$ as discussed in Section 2. For pedagogical purposes, we define general terms to subsume both terms and commands. We define general terms

The syntax given above is for the full language. Many of the constructs are actually defined as syntactic sugar. The core language and sugar are formally distinguished in the following description of context-sensitive syntax. In particular, the core language only contains elementary patterns whose components are all variables. Matching of complex patterns is defined in terms of matching elementary patterns, *e.g.*,

$$\texttt{let}\ t\ \texttt{be}\ (P_1, P_2)\ \texttt{in}\ T \quad \equiv \quad \texttt{let}\ t\ \texttt{be}\ (x_1, x_2)\ \texttt{in}\ \texttt{let}\ x_1\ \texttt{be}\ P_1\ \texttt{in}\ \texttt{let}\ x_2\ \texttt{be}\ P_2\ \texttt{in}\ T$$

[9] The use of ϵ for this binding construct is inspired by the Hilbert's ϵ operator [18]. It is somewhat loosely related to the latter in terms of functional logic programming [4], but this relationship is yet to be explored fully.

Similarly, we also extend the pattern notation for other binding constructs as follows:

$$\begin{array}{lcl}
\lambda P\colon A.\, t & \equiv & \lambda x\colon A.\, \texttt{let}\ x\ \texttt{be}\ P\ \texttt{in}\ t \\
\epsilon P\colon A.\, \theta & \equiv & \epsilon x\colon A.\, \texttt{let}\ x\ \texttt{be}\ P\ \texttt{in}\ \theta \\
inl(P) \Rightarrow T & \equiv & inl(x) \Rightarrow \texttt{let}\ x\ \texttt{be}\ P\ \texttt{in}\ T \\
inr(P) \Rightarrow T & \equiv & inr(x) \Rightarrow \texttt{let}\ x\ \texttt{be}\ P\ \texttt{in}\ T \\
P \Rightarrow T & \equiv & x \Rightarrow \texttt{let}\ x\ \texttt{be}\ P\ \texttt{in}\ T \\
![t;\ P]T & \equiv & ![t;\ x]\texttt{let}\ x\ \texttt{be}\ P\ \texttt{in}\ T
\end{array}$$

Other syntactic liberties are that we often omit the type annotations in $\epsilon x\colon A.\, \theta$, $\lambda x\colon A.\, t$ and $t_1 :=_A t_2$.

Since the language is quite large, we describe it in various fragments. The *multiplicative* fragment involves the types $1, \bot, A_1 \times A_2, A_1 \parallel A_2$ and $A_1 \multimap A_2$. The *additive* fragment involves the connectives & and + while the *exponential* fragment involves the "modalities" ! and ?.

Our type inference system is a sequent calculus for sequents of the form

$$\begin{array}{ll}
\Gamma \vdash \{\theta\} & \text{(command sequents)} \\
\Gamma \vdash t : A & \text{(term sequents)}
\end{array}$$

Here Γ is a *set* of type assumptions of the form $x_i : A_i$, θ is a command, t is a simple term and A is a type. We often leave out θ in a term sequent, if it is *noop*.

The rules in a sequent calculus are classified into left-rules ($\mathcal{L}$) and right-rules ($\mathcal{R}$) which introduce a connective to the left or right of $\vdash$ respectively. The only exceptions are the following structural rules:

$$\begin{array}{lll}
\text{Axiom} & \dfrac{}{\vdash \{noop\}} & \dfrac{}{x : A \vdash x : A} \\[2ex]
\text{Cut} & \dfrac{\Gamma \vdash t : A \quad \Delta, x : A \vdash \{\phi\}}{\Gamma, \Delta \vdash \{\phi[t/x]\}} & \dfrac{\Gamma \vdash t : A \quad \Delta, x : A \vdash u : B}{\Gamma, \Delta \vdash u[t/x] : B}
\end{array}$$

Here, *noop* is the empty command.

A comparison of the two cut rules shows that the rules for command-sequents can be obtained by leaving out the term part of the term-sequents. In the following, we often leave out the rules for command-sequents which can be similarly obtained.

3.1 Multiplicative fragment

We have the following right- and left-rules for $(\cdot)^\bot$, 1 and $\times$:

$$(\cdot)^\bot\mathcal{R}\ \ \frac{\Gamma, x : A \vdash \{\theta\}}{\Gamma \vdash \epsilon x\colon A.\, \theta : A^\bot} \qquad (\cdot)^\bot\mathcal{L}\ \ \frac{\Gamma \vdash t : A}{\Gamma, x : A^\bot \vdash x :=_A t}$$

$$1\,\mathcal{R}\ \ \frac{}{\vdash () : 1} \qquad 1\,\mathcal{L}\ \ \frac{\Gamma \vdash t : C}{\Gamma, z : 1 \vdash \texttt{let}\ z\ \texttt{be}\ ()\ \texttt{in}\ t : C}$$

$$\times\mathcal{R}\ \ \frac{\Gamma \vdash t : A \quad \Delta \vdash u : B}{\Gamma, \Delta \vdash (t, u) : A \times B} \qquad \times\mathcal{L}\ \ \frac{\Gamma, x : A, y : B \vdash t : C}{\Gamma, z : A \times B \vdash \texttt{let}\ z\ \texttt{be}\ (x, y)\ \texttt{in}\ t : C}$$

The rules $1\mathcal{L}$ and $\times\mathcal{L}$ have their counterparts for command sequents.

The term $\epsilon x\colon A.\, \theta$ denotes an acceptor which, given an $x : A$, executes the command θ. The command $a :=_A t$ supplies the value t to the A-acceptor a. The remaining constructs are self-explanatory.

The operational semantics is given by the following reductions:

$$
\begin{array}{rcll}
(\epsilon x\colon A.\, \theta) :=_A t & \to & \theta[t/x] & (1)\\
t :=_{A^\perp} (\epsilon x\colon A.\, \theta) & \to & \theta[t/x] & (2)\\
\texttt{let}\ ()\ \texttt{be}\ ()\ \texttt{in}\ T & \to & \theta & (3)\\
\texttt{let}\ (t, u)\ \texttt{be}\ (x, y)\ \texttt{in}\ T & \to & T[t/x, u/y] & (4)\\
\epsilon x\colon A^\perp.\, (x :=_A t) & \to & t & (5)\\
\epsilon x\colon A^\perp.\, (t :=_{A^\perp} x) & \to & t & (6)
\end{array}
$$

By using T, we indicate that (3) and (4) apply to both commands and terms. Note that an assignment $t :=_A u$ is treated the same way as $u :=_{A^\perp} t$.

Derived rules In the above, we did not give any rules for $\perp$ and $\parallel$ and $\multimap$. Type rules can be derived for them using the definitions mentioned in Section 2. The equality $\perp = 1^\perp$ gives

$$
\dfrac{\dfrac{\Gamma \vdash \{\theta\}}{\Gamma, z : 1 \vdash \{\texttt{let}\ z\ \texttt{be}\ ()\ \texttt{in}\ \theta\}}\, 1\,\mathcal{L}}{\Gamma \vdash \epsilon().\, \theta : \perp}\, (\cdot)^\perp\mathcal{R}
\qquad
\dfrac{\dfrac{}{\vdash () : 1}\, 1\mathcal{R}}{z : \perp \vdash \{z :=_1 ()\}}\, (\cdot)^\perp\mathcal{L}
$$

The term $\epsilon().\, \theta$ denotes a suspension of the command θ (Cf. Section 2). Assigning $()$ to it forces the execution of the command. We introduce the syntactic sugar $do(t)$ for such forcing.

$$
do(t) \quad \equiv \quad t :=_1 ()
$$

The equality $A \parallel B = (A^\perp \times B^\perp)^\perp$ gives the derived rules:

$$
\dfrac{\Gamma, x : A^\perp, y : B^\perp \vdash \{\theta\}}{\Gamma \vdash \epsilon(x, y).\, \theta : A \parallel B}
\qquad
\dfrac{\Gamma, x : A \vdash \{\phi\} \quad \Delta, y : B \vdash \{\theta\}}{\Gamma, \Delta, z : A \parallel B \vdash \{z :=_{A^\perp \times B^\perp} (\epsilon x.\, \phi, \epsilon y.\, \theta)\}}
$$

This motivates the following syntactic sugar:

$$
\begin{array}{rcll}
par(t;\ x \Rightarrow \phi;\ y \Rightarrow \theta) & \equiv & t :=_{A^\perp \times B^\perp} (\epsilon x.\, \phi, \epsilon y.\, \theta) & \\
parl(t;\ x \Rightarrow u;\ y \Rightarrow \{\theta\}) & \equiv & \epsilon z : C^\perp.\, t :=_{A^\perp \times B^\perp} (\epsilon x.\, (z :=_C u),\ \epsilon y.\, \theta) & : \ C\\
parr(t;\ x \Rightarrow \{\phi\};\ y \Rightarrow u) & \equiv & \epsilon z : C^\perp.\, t :=_{A^\perp \times B^\perp} (\epsilon x.\, \phi,\ \epsilon y.\, (z :=_C u)) & : \ C
\end{array}
$$

The command $par(t;\ x \Rightarrow \phi;\ y \Rightarrow \theta)$ decomposes the parallel pair t and consumes both the components. The operators $parl$ and $parr$ are the counterparts of par for terms.

Finally, the equality $A \multimap B = A^\perp \parallel B = (A \times B^\perp)^\perp$ gives the familiar derived rules for functions, but with rather unconventional term assignments:

$$
\dfrac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \epsilon(x, y).\, (y :=_B t) : A \multimap B}
\qquad
\dfrac{\Gamma \vdash t : A \quad \Delta, y : B \vdash \{\theta\}}{\Gamma, \Delta, f : A \multimap B \vdash \{f :=_{A \times B^\perp} (t, \epsilon y.\, \theta)\}}
$$

These rules motivate the following syntactic sugar:

$$\begin{array}{rcll} \lambda x\colon A.\, t & \equiv & \epsilon(x,y)\colon A \times B^{\perp}.\, (y :=_B t) & : \quad A \multimap B \\ t_1\, t_2 & \equiv & \epsilon y\colon B^{\perp}.\, \{t_1 :=_{A\times B^{\perp}} (t_2, y)\} & : \quad B \end{array}$$

A function $\lambda x\colon A.\, t$ is equivalent to an acceptor which accepts, in addition to the A-typed value x, a B-acceptor y and supplies the output t to y. Applying such a function involves providing an A-typed input and a B-acceptor.

Note that these reductions imply the following derived reductions for the sugared constructs:

$$\begin{array}{rcl} do(\epsilon().\, \theta) & \rightarrow & \theta \\ par(\epsilon(x',y')\colon A^{\perp} \times B^{\perp}.\, \psi;\ x \Rightarrow \phi;\ y \Rightarrow \theta) & \rightarrow & \psi[\epsilon x\colon A.\, \phi/x', \epsilon y\colon B.\, \theta/y'] \\ parl(\epsilon(x',y')\colon A^{\perp} \times B^{\perp}.\, \psi;\ x \Rightarrow t;\ y \Rightarrow \{\theta\}) & \rightarrow & t[(\epsilon x'\colon A^{\perp}.\, \psi[\epsilon y\colon B.\, \theta/y'])/x] \\ parr(\epsilon(x',y')\colon A^{\perp} \times B^{\perp}.\, \psi;\ x \Rightarrow \{\phi\};\ y \Rightarrow t) & \rightarrow & t[(\epsilon y'\colon B^{\perp}.\, \psi[\epsilon x\colon A.\, \phi/x'])/y] \\ (\lambda x\colon A.\, t)u & \rightarrow & t[u/x] \end{array}$$

Example 1 Let us express the difference list example using this syntax:

$$\begin{array}{lcl} DList & = & List \parallel List^{\perp} \\ new & : & 1 \multimap DList \\ addl & : & Int \multimap DList \multimap DList \\ addr & : & DList \multimap Int \multimap DList \\ append & : & DList \multimap DList \multimap DList \\ out & : & DList \multimap List \\ new\ () & = & \epsilon(l,t).\, l := t \\ addl\ x\ d & = & \epsilon(l',t').\, par(d;\ l \Rightarrow l' := x :: l;\ t \Rightarrow t := t') \\ addr\ d\ x & = & \epsilon(l',t').\, par(d;\ l \Rightarrow l' := l;\ t \Rightarrow t := x :: t') \\ append\ d_1\ d_2 & = & \epsilon(l',t').\, par(d_1;\ l_1 \Rightarrow l' := l_1;\ t_1 \Rightarrow par(d_2;\ l_2 \Rightarrow t_1 := l_2;\ t_2 \Rightarrow t_2 := t')) \\ out\ d & = & parl(d;\ l \Rightarrow l;\ t \Rightarrow \{t := nil\}) \end{array}$$

The type *DList* is defined as the parallel combination of a list and a list variable. Using the parallel combination allows both the components to be defined within the same context, thus sharing variables. For the function *new*, (l,t) is an output, but t is an input embedded within the output. The type of (l,t) is $(List \parallel List^{\perp})^{\perp} = List^{\perp} \times List$. From the two components, a contradiction is produced by assigning t to l. Of the remaining functions, *addl* and *addr* add an element to the left and right of a difference list respectively, and *append* appends two difference lists. Finally, the function *out* converts a difference list into a conventional list. Note that all the functions take constant time—something hard to achieve in a traditional functional language. This is made possible by the fact that we can create acceptors but delay assigning to them until such time as information becomes available. (Note that an acceptor t created in the *new* operation is eventually assigned a value in the *out* operation). However, the type system guarantees that every variable is eventually assigned.

3.2 Additive fragment

We have the following type rules for &:

$$\&\mathcal{R} \qquad \frac{\Gamma \vdash t : A \quad \Gamma \vdash u : B}{\Gamma \vdash \langle t, u\rangle : A \,\&\, B}$$

$$\&\mathcal{L} \qquad \frac{\Gamma, x : A \vdash t : C}{\Gamma, z : A \,\&\, B \vdash \texttt{let}\ z\ \texttt{be}\ \langle x, _\rangle\ \texttt{in}\ t : C} \qquad \frac{\Gamma, y : B \vdash t : C}{\Gamma, z : A \,\&\, B \vdash \texttt{let}\ z\ \texttt{be}\ \langle _, y\rangle\ \texttt{in}\ t : C}$$

Pairs of the form $\langle t_1, t_2\rangle$ are called *lazy pairs* because the components cannot be computed in advance. The decomposition construct chooses precisely one of the two components. It is only at this time that the computation of the chosen component be carried out.

These terms satisfy the following reductions:

$$\texttt{let}\ \langle t, u\rangle\ \texttt{be}\ \langle x, _\rangle\ \texttt{in}\ T \ \rightarrow\ T[t/x] \tag{7}$$

$$\texttt{let}\ \langle t, u\rangle\ \texttt{be}\ \langle _, y\rangle\ \texttt{in}\ T \ \rightarrow\ T[u/y] \tag{8}$$

As usual, right- and left-rules for + can be derived by duality:

$$+\mathcal{R} \qquad \frac{\Gamma \vdash t : A}{\Gamma \vdash \epsilon\langle x, _\rangle.\ (x :=_A t) : A + B} \qquad \frac{\Gamma \vdash u : B}{\Gamma \vdash \epsilon\langle _, y\rangle.\ (y :=_B u) : A + B}$$

$$+\mathcal{L} \qquad \frac{\Gamma, x : A \vdash t : C \quad \Gamma, y : B \vdash u : C}{\Gamma, z : A + B \vdash \epsilon c.\ (z :=_{A^\perp \,\&\, B^\perp} \langle \epsilon x.\ (c :=_A t),\ \epsilon y.\ (c :=_B u)\rangle) : C}$$

This motivates the following syntactic sugar:

$$\begin{array}{rcll} inl(t) & \equiv & \epsilon\langle x, _\rangle.\ x :=_A t & : \ A + B \\ inr(u) & \equiv & \epsilon\langle _, y\rangle.\ y :=_B u & : \ A + B \\ \texttt{case}\ v\ \texttt{of}\ inl(x) \Rightarrow t \mid inr(y) \Rightarrow u & = & \epsilon c.\ (v :=_{A^\perp \,\&\, B^\perp} \langle \epsilon x.\ (c :=_A t),\ \epsilon y.\ (c :=_B u)\rangle) & : \ C \end{array}$$

3.3 Exponential fragment

The exponential fragment involves the constructors ! and ?. The type $!A$ consists of *persistent* versions of A-typed values. Such values can be discarded or used multiple times. The term constructions for $!A$ are somewhat complex owing to the need for explicit discarding and copying of all the inputs. In a programming language, it would be best to leave such coercions implicit. If the coercions were left implicit, the type rules for ! would be the following:

$$!\mathcal{R} \quad \frac{!\Gamma \vdash t : A}{!\Gamma \vdash t : !A} \qquad !\mathcal{L} \quad \frac{\Gamma, x : A \vdash t : B}{\Gamma, z : !A \vdash t[z/x] : B}$$

$$!\mathcal{W} \quad \frac{\Gamma \vdash t : C}{\Gamma, z : !A \vdash t : C} \qquad !\mathcal{C} \quad \frac{\Gamma, x : !A, y : !A \vdash t : C}{\Gamma, z : !A \vdash t[z/x, z/y] : C}$$

In the $!\mathcal{R}$ rule, $!\Gamma$ stands for a type context all of whose types are ! types. The rule says that an A-typed value can be considered persistent if all the inputs it uses are also persistent. The $!\mathcal{L}$ rule allows a persistent value to be treated as an ordinary (linear) value. The weakening

rule ($!\mathcal{W}$) allows a persistent value to be discarded. Finally, the contraction rule ($!\mathcal{C}$) allows it to be copied (used multiple times).

For conceptual clarity, as well as for better support for ? types to be introduced shortly, we make all the coercions explicit. This gives the following type rules:

$$!\mathcal{R}\ \frac{\bar{x} : \bar{B} \vdash t : A}{\bar{y} : \bar{B} \vdash ![\bar{y}; \bar{x}]t : !A} \qquad !\mathcal{L}\ \frac{\Gamma, x : A \vdash t : B}{\Gamma, z : !A \vdash \texttt{let}\ z\ \texttt{be}\ !x\ \texttt{in}\ t : B}$$

$$!\mathcal{W}\ \frac{\Gamma \vdash t : C}{\Gamma, z : !A \vdash \texttt{let}\ z\ \texttt{be}\ _\ \texttt{in}\ t : C} \qquad !\mathcal{C}\ \frac{\Gamma, x : !A, y : !A \vdash t : C}{\Gamma, z : !A \vdash \texttt{let}\ z\ \texttt{be}\ x@y\ \texttt{in}\ t : C}$$

The term $![\bar{u}; \bar{x}]t$ denotes a persistent version of the value denoted by t with the use of the persistent inputs $\bar{u}$. The need for keeping track of the inputs is demonstrated by the following reduction rules for the coercions:

$$\texttt{let}\ ![\bar{u}; \bar{x}]t\ \texttt{be}\ !y\ \texttt{in}\ T \ \rightarrow\ T[t[\bar{u}/\bar{x}]/y] \tag{9}$$

$$\texttt{let}\ ![\bar{u}; \bar{x}]t\ \texttt{be}\ _\ \texttt{in}\ T \ \rightarrow\ \texttt{let}\ \bar{u}\ \texttt{be}\ _\ \texttt{in}\ T \tag{10}$$

$$\begin{aligned}\texttt{let}\ ![\bar{u}; \bar{x}]t\ \texttt{be}\ y_1@y_2\ \texttt{in}\ T \ \rightarrow\ & \texttt{let}\ \bar{u}\ \texttt{be}\ \bar{x}_1@\bar{x}_2 \\ & \texttt{in}\ T[![\bar{x}_1; \bar{x}]t/y_1,\ ![\bar{x}_2; \bar{x}]t/y_2]\end{aligned} \tag{11}$$

The reduction (10) treats the one-time use of a persistent value. In this case, the inputs $\bar{u}$ are used and they are substituted for the parameters $\bar{x}$. The reduction (11) treats the discarding of a persistent value. In this case, the inputs $\bar{u}$ must also be discarded. The notation $\texttt{let}\ \bar{u}\ \texttt{be}\ _\ \texttt{in}\ v$ means $\texttt{let}\ u_1\ \texttt{be}\ _\ \texttt{in}\ \ldots \texttt{let}\ u_k\ \texttt{be}\ _\ \texttt{in}\ v$. Finally, the reduction (12) treats the copying of persistent values. In this case, the inputs $\bar{u}$ are copied as well. In practice, the evaluation of the inputs $\bar{u}$ would be delayed until (10) is used. The reduction (12) would involve setting up additional pointers to the inputs.

Example 2 Linear logic satisfies the following isomorphism:

$$sep_pair : !(A \,\&\, B) \cong !A \times !B : merge_pair$$

A persistent version of a lazy pair is equivalent to an independent pair of persistent components. The isomorphisms can be defined using the bivalent functional language as follows:

$$\begin{aligned} sep_pair\ v_1@v_2 &= (![v_1; !\langle a, _\rangle]a,\ ![v_2; !\langle _, b\rangle]b) \\ merge_pair\ (v_1, v_2) &= ![v_1, v_2;\ !a, !b]\langle a, b\rangle \end{aligned}$$

It can be shown that the two functions are inverses of each other using secondary equations (which are omitted from this summary).

Let us now look at the ? type constructor. As usual, we can produce derived type rules using duality.

$$?\mathcal{R}\ \frac{\Gamma \vdash t : A}{\Gamma \vdash \epsilon!x.\ (x :=_A t) : ?A} \qquad ?\mathcal{L}\ \frac{\bar{x} : !\bar{B}, y : A \vdash \{\phi\}}{\bar{x}' : !\bar{B}, z : ?A \vdash \{z :=_{!A} ![\bar{x}'; \bar{x}]\epsilon y.\ \phi\}}$$

$$?\mathcal{W}\ \frac{\Gamma \vdash \{\theta\}}{\Gamma \vdash \epsilon_.\ \theta : ?A} \qquad ?\mathcal{C}\ \frac{\Gamma, x : !A^{\perp}, y : !A^{\perp} \vdash \{\theta\}}{\Gamma \vdash \epsilon x@y.\ \theta : ?A}$$

We introduce the following syntactic sugar for the term assignment of $?\mathcal{L}$.

$$choose[\bar{u};\bar{x}](t; y \Rightarrow \phi) \quad \equiv \quad t :=_{!A} ![\bar{u};\bar{x}]\ \epsilon y.\ \phi$$

This command, given an indeterminate term t, binds y to each possible value of t and executes the command ϕ in that context. Since all the inputs $\bar{x}$ used in ϕ are persistent, they can be used and reused for each value of t. The following derived reductions for ? types make these computations clear:

$$\begin{array}{rcl}
choose[\bar{u};\bar{x}](\epsilon !x.\ (x := t);\ y \Rightarrow \phi) & \rightarrow & \phi[\bar{u}/\bar{x}][t/y] \\
choose[\bar{u};\bar{x}](\epsilon _.\ \theta;\ y \Rightarrow \phi) & \rightarrow & \texttt{let}\ \bar{u}\ \texttt{be}\ _\ \texttt{in}\ \theta \\
choose[\bar{u};\bar{x}](\epsilon z_1@z_2.\ \theta;\ y \Rightarrow \phi) & \rightarrow & \texttt{let}\ \bar{u}\ \texttt{be}\ \bar{x}_1@\bar{x}_2 \\
 & & \texttt{in}\ choose[\bar{x}_1;\bar{x}](\epsilon z_1.\ choose[\bar{x}_2;\bar{x}](\epsilon z_2.\ \theta;\ y \Rightarrow \phi); \\
 & & \qquad\qquad y \Rightarrow \phi)
\end{array}$$

Example 3 The following example illustrates the indeterminacy available with ? types. Consider splitting a list into two parts:

$$\begin{array}{lcl}
type\ List & = & nil + (!Int \times !List) \\
split & : & !List \rightarrow ?(!List \otimes !List) \\
split\ !l_1@!l_2 & = & \epsilon\ !r_1@rs.\ (r_1 := (!nil, !l_1), \\
 & & \qquad \texttt{case}\ l_2\ \texttt{of} \\
 & & \qquad\quad nil \Rightarrow \texttt{let}\ rs\ \texttt{be}\ _\ \texttt{in}\ noop \\
 & & \qquad \mid\ h :: t \Rightarrow choose[rs, h;\ rs, h] \\
 & & \qquad\qquad\qquad (split\ t;\ (s_1, s_2) \Rightarrow \texttt{let}\ rs\ \texttt{be}\ !r_2 \\
 & & \qquad\qquad\qquad\qquad\qquad \texttt{in}\ r_2 := (!(h :: s_1), s_2)))
\end{array}$$

Indeterminate results are produced by accepting a persistent acceptor of list pairs and giving each copy of the acceptor a different split of the input list.

4 Reduction properties

In this section, we show the two important properties of the reduction semantics: strong normalization and confluence. The former property means that every reduction sequence is finite and the latter (also called Church-Rosser property) means that any two reduction sequences from a term t have a common term t'.

4.1 Strong normalization

We give a proof similar to that of Prawitz [33]. First, we consider a type system slightly from that of Section 2. It has the type constructors 1, $\times$, $\&$, ! and $(\cdot)^\perp$, where the last constructor is used only for types which are not of the form $A^\perp$, and an additional type *cmd*. (Essentially, this type system covers only the negation normal forms of the original system). For each type A, we define a set of well-typed terms $[\![A]\!]$ by induction on the structure of A. (Treat *cmd* as being smaller than every other type). The definition uses an auxiliary set $\mathrm{Cv}(A)$ for "convergence" of A:

$$\begin{array}{lcl}
\mathrm{Cv}(cmd) & = & \{noop\} \\
\mathrm{Cv}(1) & = & \{()\} \\
\mathrm{Cv}(A \times B) & = & \{(u, v) \mid u \in [\![A]\!] \wedge v \in [\![B]\!]\} \\
\mathrm{Cv}(A \,\&\, B) & = & \{\langle u, v\rangle \mid u \in [\![A]\!] \wedge v \in [\![B]\!]\} \\
\mathrm{Cv}(!A) & = & \{![\bar{u}; \bar{x}]v \mid (\forall i.\, u_i \in [\![!B_i]\!]) \wedge v[\bar{u}/\bar{x}] \in [\![A]\!]\} \\
\mathrm{Cv}(A^\perp) & = & \{\epsilon x\colon A.\, \theta \mid \forall u.\, u \in [\![A]\!] \Rightarrow \theta[u/x] \in [\![cmd]\!]\}
\end{array}$$

where in the last equation A is not of the form $B^\perp$. Call a term "neutral" if it is a variable, constant, `let` t `be` P `in` u or $t := u$. Now, for each type A, $[\![A]\!]$ is the least set satisfying the equation:

$$[\![A]\!] \quad = \quad \mathrm{Cv}(A) \cup \{t \mid t \text{ neutral} \wedge \forall t'.\, t \to t' \Rightarrow t' \in [\![A]\!]\}$$

$[\![A]\!]$ is essentially the set of terms which reduce to a normal form or a term in $\mathrm{Cv}(A)$ along every reduction sequence. Note that a neutral term of type A that is normal is included in $[\![A]\!]$ (by vacuous universal quantification). It is also easy to see by induction on type terms that

Lemma 4 For all types A, $[\![A]\!] \neq \emptyset$.

The outline of the proof is as follows: First, we show that all terms belonging to type sets are strongly normalizable (Theorem 6). Next, we use this fact in showing that neutral terms belong to type sets (Lemmas 8 and 9). It then follows that all terms belong to type sets (Theorem 10) and, hence, are normalizable. Next, we consider the full type system. This involves the other form of neutral terms $\epsilon x\colon A^\perp.\, \theta$. We show that they belong to the type set $[\![A]\!]$. This concludes the proof.

Lemma 5 If $t \in [\![A]\!]$ and $t \to t'$ then $t' \in [\![A]\!]$.

Proof: By induction on the definition of $[\![A]\!]$.

Theorem 6 (strong normalization) All terms $t \in [\![A]\!]$ are is strongly normalizable.

Proof: By induction on the definition of $[\![A]\!]$. If $t \in \mathrm{Cv}(A)$, use the inductive hypothesis for the smaller types. In the case of $A^{\perp}$, use Lemma 4 as well. If t is neutral, every t' that t reduces to in one step is strongly normalizable by inductive hypothesis. ∎

Even though this result is stated for closed terms, using the fact that type sets are nonempty, it applies to open terms as well.

Corollary 7 If $t[\bar{u}/\bar{x}] \in [\![A]\!]$ whenever $\bar{u}$ belong to type sets, then t is strongly normalizable.

Lemma 8 Assume $v \in [\![C]\!]$. Then, the following statements hold whenever the terms involved are well-typed:

1. $t \in [\![1]\!] \Rightarrow (\texttt{let}\ t\ \texttt{be}\ ()\ \texttt{in}\ v) \in [\![C]\!]$.
2. $t \in [\![A \times B]\!] \wedge (\forall u_1 \in [\![A]\!], u_2 \in [\![B]\!].\ v[u_1/x, u_2/y] \in [\![C]\!]) \Rightarrow (\texttt{let}\ t\ \texttt{be}\ (x, y)\ \texttt{in}\ v) \in [\![C]\!]$.
3. $t \in [\![A \,\&\, B]\!] \wedge (\forall u \in [\![A]\!].\ v[u/x] \in [\![C]\!]) \Rightarrow (\texttt{let}\ t\ \texttt{be}\ \langle x, _\rangle\ \texttt{in}\ v[u/x]) \in [\![C]\!]$.
4. $t \in [\![A \,\&\, B]\!] \wedge (\forall u \in [\![B]\!].\ v[u/y] \in [\![C]\!]) \Rightarrow (\texttt{let}\ t\ \texttt{be}\ \langle _, y\rangle\ \texttt{in}\ v[u/y]) \in [\![C]\!]$.
5. $t \in [\![!A]\!] \wedge (\forall u \in [\![A]\!].\ v[u/x] \in [\![C]\!]) \Rightarrow (\texttt{let}\ t\ \texttt{be}\ !x\ \texttt{in}\ v) \in [\![C]\!]$.
6. $t \in [\![!A]\!] \Rightarrow (\texttt{let}\ t\ \texttt{be}\ _\ \texttt{in}\ v) \in [\![C]\!]$.
7. $t \in [\![!A]\!] \wedge (\forall u_1, u_2 \in [\![!A]\!].\ v[u_1/y_1, u_2/y_2] \in [\![C]\!]) \Rightarrow (\texttt{let}\ t\ \texttt{be}\ y_1@y_2\ \texttt{in}\ v) \in [\![C]\!]$.

Proof: Since t and v belong to type sets, they are strongly normalizable by Theorem 6. The proof is by induction on $\langle t, v\rangle$ compared by lexicographic order. The first component is compared by the relational composition[10] $\rightarrow^* \rhd$ and the second component by $\rightarrow^*$.

A reduction of $\texttt{let}\ t\ \texttt{be}\ P\ \texttt{in}\ v$ is either a reduction of t, a reduction of v or a reduction at top. In the first two cases, we have that the subterms belong to type sets by Lemma 5. Since the pair $\langle t, v\rangle$ is smaller for the resulting term, we use the inductive hypothesis.

If the reduction is at top, the additional hypothesis about v can be used to conclude that the resulting term is in $[\![C]\!]$. (We show this below for selected cases). Since every reduction of the term gives a term in $[\![C]\!]$, the term is in $[\![C]\!]$ by definition.

For 1, the reduction at top gives v which is assumed to be in $[\![C]\!]$.

For 2, the reduction at top gives $v[u_1/x, u_2/x]$ where $t = (u_1, u_2)$. By the hypothesis for t, we have $u_1 \in [\![A]\!]$ and $u_2 \in [\![B]\!]$. The additional hypothesis for v shows that the term is in $[\![C]\!]$.

For 5, the reduction at top gives $v[u[\bar{u}/\bar{x}]/x]$ where $t = ![\bar{u}; \bar{x}]u$. By the hypothesis for t, $u[\bar{u}/\bar{x}] \in [\![A]\!]$. The additional hypothesis for v gives the result.

For 6, the reduction at top gives $\texttt{let}\ \bar{u}\ \texttt{be}\ _\ \texttt{in}\ v$ where $t = ![\bar{u}/\bar{x}]u$. Since $t \in [\![!A]\!]$, each u_i is in some $[\![!B_i]\!]$. Moreover, u_i is a subterm of t. Hence, use the inductive hypothesis to infer that $\texttt{let}\ \bar{u}\ \texttt{be}\ _\ \texttt{in}\ v$ is in $[\![C]\!]$.

For 7, the reduction at top gives

$$\texttt{let}\ \bar{u}\ \texttt{be}\ \bar{x}_1@\bar{x}_2\ \texttt{in}\ v[![\bar{x}_1; \bar{x}]u/y_1, ![\bar{x}_2; \bar{x}]u/y_2]$$

where $t = ![\bar{u}/\bar{x}]u$. Use the same reasoning as for 6, noting that the additional hypothesis for v shows that the new body of `let` is in $[\![C]\!]$. ∎

[10] $t \rhd t'$ if t' is a subterm of t. It is easy to see that $\rightarrow^* \rhd$ is a well-founded order. See [36] for an argument.

Lemma 9 For nonnegative A, if $t \in [\![A^\perp]\!]$ and $u \in [\![A]\!]$ then $(t :=_A u) \in [\![cmd]\!]$.

Proof: By induction on $\langle t, u\rangle$ compared by the lexicographic combination of $\rightarrow^*$. If $t := u$ is reduced by reducing t or u, use the inductive hypothesis. If it reduced at the top, then one of the reduction rules (1) and (2) is used. Suppose it is (1). Then, t is of the form $\epsilon x\colon A.\,\theta$ and the command reduces to $\theta[t/x]$ which is in $[\![C]\!]$ by definition of $[\![A^\perp]\!]$. The other case is similar. ■

An environment η is a map from variables to $\bigcup_A [\![A]\!]$. Since $[\![A]\!]$'s are made of terms, η is essentially a substitution. An environment is said to *satisfy* a type context $\bar{x} : \bar{A}$ (written $\eta \models \bar{x} : \bar{A}$) if $dom(\eta) = \bar{x}$ and $\eta(x_i) \in [\![A_i]\!]$ for all i. We say that $\Gamma \models t : A$ if for all $\eta \models \Gamma$, $t\eta \in [\![A]\!]$. We have the following result:

Theorem 10 (realizability) $\Gamma \vdash t : A \Rightarrow \Gamma \models t : A$, and $\Gamma \vdash \{\theta\} \Rightarrow \Gamma \models \theta : cmd$.

Proof: By induction on the derivation of $\Gamma \vdash t : A$. Consider the last inference.

- If it is one of the two axioms, $noop \in [\![cmd]\!]$ and $x : A \models x : A$ by definition. Similarly, if it is an instance of cut, the result follows by definition and the inductive hypothesis for the premises.
- If it is an instance of a $\mathcal{R}$ rule, other than $(\cdot)^R$, then t is easily seen to be in $\mathrm{Cv}(A)$. If it is $(\cdot)^\perp\mathcal{R}$ and A is not negative, use the fact that $x : A$ is neutral and normal. [[Is this needed?]] If $A = B^\perp$ and B is not negative, by inductive hypothesis, $\theta\eta \in [\![cmd]\!]$ whenever $\eta \models \Gamma, x : B^\perp$. ...
- If it is an instance of a $\mathcal{L}$ rule or a structural rule, the result follows from Lemmas 8 and 9.

Next, we consider the full type system including double negated types. Make the following observation:

Lemma 11 Converses of Lemmas 8 and 9 hold.

Proof: There is exactly one derivation for each $\Gamma \vdash t : A$. Thus, if the conclusion of a type rule is derivable, the premises are derivable as well. The converses then follow from Theorem 10. ■

Lemma 12 For any nonnegative type A, if $t \in [\![A]\!] \Rightarrow \theta[t/x] \in [\![cmd]\!]$ then $\epsilon x\colon A^\perp.\,\theta \in [\![A]\!]$.

Proof: By induction on θ compared by $\rightarrow^*$. If $\epsilon x\colon A^\perp.\,\theta$ is reduced by reducing θ, use the inductive hypothesis. Otherwise, one of the reduction rules (5) and (6) is used. Suppose (5) is used. Then, θ is of the form $x :=_A u$. By hypothesis, $t :=_A u \in [\![cmd]\!]$ whenever $t \in [\![A]\!]$. Since $[\![A]\!]$ is nonempty, using the converse of Lemma 9, $u \in [\![A]\!]$. Since every immediate reduction leads to a term in $[\![A]\!]$, $\epsilon x\colon A^\perp.\,\theta \in [\![A]\!]$. ■

Corollary 13 For all types A, $[\![A^{\perp\perp}]\!] \subseteq [\![A]\!]$.

Proof: Follows from Lemma 12 using induction on the structure of A. Note also that $[\![(\cdot)^\perp]\!]$ is monotonic from the definition. ■

Putting all the pieces together, we have

Theorem 14 If $\Gamma \vdash t : A$ is derivable then t is strongly normalizable. That is, all well-typed terms are strongly normalizable.

Proof: By Theorem 10, we have $\Gamma \models t : A$. Since type sets are nonempty, there exists $\eta \models \Gamma$. So, $t\eta \in [\![A]\!]$ and, by Theorem 6 $t\eta$ is strongly normalizable. But, then t must be strongly normalizable as well. ■

4.2 Confluence

Since well-typed terms are strongly normalizable, the reduction relation $\rightarrow^*$ is Noetherian. By classical results (see, for example, [21]), we only need to show that all critical pairs are joinable. At this time, we are only able to show this for closed terms.

The only critical overlaps are, on the one hand, between (1) and (2), and, on the other, between each of (1) and (2) and each of (5) and (6). Note that all these overlaps form commands. Since there are no command constants, a closed command has only one possible normal form: *noop*. Hence, by strong normalization, each term of a critical pair must reduce to *noop*. Hence, we have

Theorem 15 The reduction relation $\rightarrow^*$ is confluent on closed terms.

5 Conclusion

We have shown that classical linear logic has significant applications to functional programming which go considerably beyond the applications of linearity considered in previous work. In particular, it allows the design of typed functional languages which incorporate novel programming concepts such as first-class acceptor values. These values systematize and generalize single assignment variables, on the one hand, and continuations on the other. It also allows a systematic use of nondeterminism in functional programs.

For space limitations, we did not have adequate opportunity to comment on implementation aspects. Essentially, data flow languages, logic programming languages and continuations suggest three different kinds of implementation paradigms with varying degrees of overheads and concurrency. Perhaps, the most efficient paradigm for sequential implementations is the model of logic programming where acceptors are implemented by unbound cells and are eventually filled by values.

Our main motivation in this study has been to better understand the issues of imperative concepts in functional languages [43, 42]. Single assignment variables represented by negative types are a particularly simple form of references. The insights obtained from this study should also eventually lead to a better understanding of references.

Acknowledgments

I am indebted to Dick Kieburtz and Boris Agapiev for introducing me to the fascinating world of linear logic. Thanks are also due to Sam Kamin and Vipin Swarup for several discussions.

References

[1] S. Abramsky. *Computational Interpretations of Linear Logic.* Research Report DOC 90/20, Imperial College, London, Oct 1990.

[2] S. Abramsky and S. Vickers. *Quantales, Observational Logic and Process Semantics.* Research Report DOC 90/1, Imperial College, LOndon, Jan 1990.

[3] A. Asperti. A logic for concurrency. Unpublished manuscript, Nov. 1987.

[4] F. Bronsard and U. S. Reddy. An axiomatization of a functional logic language. In H. Kirchner and W. Wechler, editors, *Algebraic and Logic Programming*, pages 101–116, Springer-Verlag, 1990. (LNCS Vol. 463).

[5] C. Brown. *Relating Petri Nets to Formulae of Linear Logic.* Technical Report ECS-LFCS-89-87, University of Edinburgh, June 1989.

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

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

[8] R. L. Constable, *et. al. Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, New York, 1986.

[9] M. Felleisen, D. P. Friedman, E. Kohlbecker, and B. Duba. A syntactic theory of sequential control. *Theoretical Comp. Science*, 52(3):205–237, 1987. (to appear).

[10] A. Filinski. Linear continuations. In *ACM Symp. on Princ. of Programming Languages*, page (to appear), ACM, Alburquerque, New Mexico, Jan 1992.

[11] V. Gehlot and C. Gunter. Normal process representatives. In *Symp. on Logic in Computer Science*, pages 200–207, IEEE, Philadelphia, Pa., June 1990.

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

[13] J.-Y. Girard. Linear logic and parallelism. In M. V. Zilli, editor, *Mathematical Models for the Semantics of Parallelism*, pages 166–182, Springer-Verlag, Berlin, 1987. (LNCS Vol. 280).

[14] 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, American Mathematical Society, Boulder, Colorado, June 1987. (Contemporary Mathematics, Vol. 92).

[15] J-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types. Cambridge Tracts in Theor. Comp. Science*, Cambridge Univ. Press, Cambridge, 1989.

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

[17] T. G. Griffin. A formulae-as-types notion of control. In *ACM Symp. on Princ. of Programming Languages*, pages 47–58, ACM, San Francisco, Jan 1990.

[18] D. Hilbert and W. Ackermann. *Principles of Mathematical Logic.* Chelsea Pub. Co., New York, 1950. translated by L. M. Hammond, G. G. Leckie, and F. Steinhardt.

[19] S. Holmstrom. A linear functional language. Draft paper, Chalmers University of Technology, 1988.

[20] 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.

[21] G. Huet. Confluent reductions: abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, October 1980. (Previous version in *Proc. Symp. Foundations of Computer Science*, Oct 1977).

[22] R. Jagadeesan, P. Panangaden, and K. Pingali. A fully abstract semantics for a functional language with logic variables. In *Fourth Ann. Symp. on Logic in Computer Science*, page , IEEE, 1989.

[23] Y. Lafont. Interaction nets. In *Seventeenth Ann. ACM Symp. on Princ. of Programming Languages*, pages 95–108, ACM, San Francisco, Ca., Jan 1990.

[24] Y. Lafont. The linear abstract machine. *Theoretical Comp. Science*, 59:157–180, 1988.

[25] G. Lindstrom. Functional programming and the logical variable. In *ACM Symp. on Princ. of Programming Languages*, 1985.

[26] D. MacQueen. Modules for Standard ML. In *ACM Symp. on LISP and Functional Programming*, pages 198–207, 1984. (Revised version in *Polymorphism Newsletter*, II, 2, Oct 85.

[27] 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, Amsterdam, 1982.

[28] N. Marti-Oliet and J. Meseguer. From Petri nets to linear logic. *Mathematical Structures in Computer Science*, 1:69–101, 1991.

[29] J. C. Mitchell and G. D. Plotkin. Abstract types have existential types. In *ACM Symp. on Princ. of Programming Languages*, pages 37–51, January 1985.

[30] C. Murthy. An evaluation semantics for classical proofs. In *Sixth Ann. Symp. on Logic in Computer Science*, pages 96–109, IEEE, Amsterdam, Jul 1991.

[31] C. Murthy. *Extracting Constructive Content from Classical Proofs.* PhD thesis, Cornell University, Aug 1990.

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

[33] D. Prawitz. Ideas and results in proof theory. In *Proc. Second Scandinavian Logic Symposium*, Springer-Verlag, 1970.

[34] U. S. Reddy. Narrowing as the operational semantics of functional languages. In *Symp. on Logic Programming*, pages 138–151, IEEE, Boston, 1985.

[35] 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.

[36] U. S. Reddy. Term rewriting induction. In M. Stickel, editor, *10th International Conference on Automated Deduction*, pages 162–177, Springer-Verlag, 1990. (Lecture Notes in Artificial Intelligence, Vol. 449).

[37] U. S. Reddy. Transformation of logic programs into functional programs. In *Intern. Symp. on Logic Programming*, pages 187–197, IEEE, 1984.

[38] J. C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372, North-Holland, 1981.

[39] J. C. Reynolds. *Preliminary Design of the Programming Language Forsythe.* Technical Report CMU-CS-88-159, Carnegie-Mellon University, June 1988.

[40] R. A. G. Seely. Linear logic, *-autonomous categories and cofree coalgebras. In J. W. Gray and A. Scedrov, editors, *Proc. AMS-IMS-SIAM Joint Conf. on Categories in Computer Science and Logic*, pages 371–382, American Mathematical Society, Boulder, Colorado, 1989. (Contemporary Mathematics, Vol. 92).

[41] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, Cambridge, Mass., 1986.

[42] V. Swarup and U. S. Reddy. A logical view of assignments. In M. J. O'Donnell, editor, *Constructivity in Computer Science*, pages 127–141, Trinity University, San Antonio, Texas, June 1991.

[43] V. Swarup, U. S. Reddy, and E. Ireland. Assignments for applicative languages. In R. J. M. Hughes, editor, *Conf. on Functional Program. Lang. and Comput. Arch.*, pages 192–214, Springer-Verlag, Berlin, 1991. (Earlier versions: University of Illinois at Urbana-Champaign, July 1990, Revised Nov 1990, Feb 1991).

[44] S. Thompson. *Type Theory and Functional Programming.* Addison-Wesley, Wokingham, England, 1991.

[45] P. Wadler. Is there a use for linear logic? In *Proc. ACM SIGPLAN Conf. on Partial Evaluation and Semantics-Based Program Manipulation*, page , ACM, New York, 1991. (SIGPLAN Notices, to appear).

[46] P. Wadler. Linear types can change the world. In M. Broy and C. B. Jones, editors, *Programming Concepts and Methods*, page , North-Holland, Amsterdam, 1990. (Proc. IFIP TC 2 Working Conf., Sea of Galilee, Israel).

[47] D. Wakeling and C. Runciman. Linearity and laziness. In R. J. M. Hughes, editor, *Conf. on Functional Program. Lang. and Comput. Arch.*, pages 215–240, Springer-Verlag, Berlin, 1991. (LNCS Vol. 523).

[48] D. H. D. Warren, L. M. Pereira, and F. Pereira. Prolog - the language and its implementation compared with Lisp. *SIGPLAN Notices*, 12(8), 1977.