Computational Interpretations of Linear Logic

Samson Abramsky

Department of Computing
Imperial College of Science, Technology and Medicine
180 Queen's Gate, London SW7 2BZ, England
sa@doc.ic.ac.uk

To appear in *Theoretical Computer Science*

February 27, 1992

Abstract

We study Girard's Linear Logic from the point of view of giving a concrete computational interpretation of the logic, based on the Curry-Howard isomorphism. In the case of Intuitionistic Linear Logic, this leads to a refinement of the lambda calculus, giving finer control over order of evaluation and storage allocation, while maintaining the logical content of programs as proofs, and computation as cut-elimination. In the classical case, it leads to a concurrent process paradigm with an operational semantics in the style of Berry and Boudol's Chemical Abstract Machine. This opens up a promising new approach to the parallel implementation of functional programming languages; and offers the prospect of typed concurrent programming in which correctness is guaranteed by the typing.

1 Introduction

Since its inception, Linear Logic [Gir87] has offered great promise, as a formalism particularly well-suited to serve at the interface between Logic and Computer Science.

- From the logical side, Linear Logic combines the symmetries of Classical Logic, as made manifest in Gentzen's sequent calculus, with the constructive content of Intuitionistic Logic.
- From the computational side, Linear Logic offers a *logical perspective* on computational issues such as control of resources and order of evaluation. By contrast, extant declarative languages either adulterate their logical content in the search for efficiency, or require an elaborate infrastructure of implementation techniques, which do not themselves draw inspiration from the mathematical structure of the language.

The paradigm followed by Girard in seeking to apply Linear Logic to computation is the "Curry-Howard" isomorphism (see e.g. [GLT89]), in which propositions (or logical formulae) are interpreted as *types*, proofs as *programs*, and the process of normalization or cut-elimination as *computation*. This paradigm has been a cornerstone of much recent work on the connections between intuitionistic logic, functional programming and category theory (see e.g. [Hue90]). It has by now firmly established itself as a major component of the logical foundations of programming. In the case of Intuitionistic Logic, it relates typed λ-calculus (*i.e.* typed functional programs in a canonical syntax) to proofs in Intuitionistic Logic; and reduction of terms to normal form (*i.e.* program execution) to normalization of proofs. What is particularly satisfying about this correspondence in the case of Intuitionistic Logic is that the formalism on the computational side is immediately recognisable as an attractive programming paradigm, which has already been extensively developed and enthusiastically advocated by a significant community of software practitioners [Tur90, BW88, FH88, PJ87].

What has been lacking to date from the development of Linear Logic is a comparably attractive computational interpretation. Such an interpretation would have two main ingredients:

- An interpretation of *proofs* as *programs*, *i.e.* well-formed expressions in some programming notation.
- An *operational semantics* for these programs, embodying a clear conception of program execution. Such a semantics should be formulated at a suitable level of abstraction, unencumbered by implementation details, so that it can serve as a reference *specification* of the language.

On this basis, Linear Logic in its computational aspect could be studied in the general framework of programming language semantics, as developed with considerable success over the last 30 years. The computational *intuitions* which have been proposed in connection with Linear Logic could be made precise, and its actual advantages as a computational formalism assessed in relation to the claims made on its behalf.

It should be said that this programme runs somewhat counter to that advocated by Girard. He has adopted the methodological principle of avoiding the "bureaucracy" of syntax [Gir89b], aiming instead for a "geometrical" view of computation, exemplified by the "Geometry of Interaction" [Gir89a], which interprets cut-elimination in Linear Logic by iterations of operators in C^*-algebras. From that perspective, what we are seeking to do here might be seen as a retrogressive step.

However, we see our work as complementary to Girard's. By giving a simple, concrete computational interpretation of Linear Logic, which makes connections with other computational formalisms apparent, and is immediately meaningful in programming terms, we hope to make Linear Logic much more accessible to Computer Scientists, and to provide the basis for some substantial applications. At the same time, we hope to establish connections with Girard's geometrical approach, although that must be left to future work.

The further contents of this paper are as follows. In Section 2, we review the connections between Intuitionistic Logic and typed λ-calculus (functional programming), and describe the operational semantics of λ-calculus in a style inspired by Martin-Löf [ML84], and currently (as "Natural" or "Relational" Semantics) widely used in programming language specification [Kah87, MTH90]. We give the correspondence for both the Natural Deduction and Sequent Calculus presentations of Intuitionistic Logic. The former is standardly used as a "type inference" system for functional programming, while the latter forms the basis for the refinement of Intuitionistic Logic into Linear Logic.

Then is Section 3 we present Intuitionistic Linear Logic (the fragment containing $\otimes$, $\multimap$, $\&$, $\oplus$, $!$). This fragment gives rise to a refined version of functional programming. We present a term calculus of notations for proofs in this fragment, essentially a refinement of the λ-calculus which allows greater control over computational behaviour, while preserving the strict correspondence between terms and proofs. We give an operational semantics for this language in the Natural Semantics style, thus achieving our basic aim of giving a computational interpretation of Intuitionistic Linear Logic. We study various aspects of this intepretation in the next two sections. In Section 4, we sketch some of the possible applications to static program analysis and optimization, and give a detailed description of a (sequential) implementation of the linear term calculus, in terms of a variant of the SECD machine [Lan64]. Then in Section 5 we establish some of the basic theoretical properties of the calculus, in its second-order propositional version.

We turn to Classical Linear Logic in sections 6–8. This should be regarded as the main contribution of the present paper. Although the material on Intuitionistic Linear logic is interesting in its own right, it has been included mainly for expository purposes. Classical Linear Logic requires a much more radical departure from the functional framework, so going by way of Intuitionistic Linear Logic helps to cushion the shock. In Section 6 we introduce *proof expressions* as a notation for proofs in Classical Linear Logic, and present a concurrent operational semantics in the style of Berry and Boudol's Chemical Abstract Machine [BB90]. The basic theoretical properties of the proof-expression calculus are established in Section 7. Finally, a parallel implementation is sketched in Section 8.

Acknowledgements

It is a great pleasure to acknowledge the enormous amount I have learnt from the writings, lectures and conversation of Jean-Yves Girard, the originator of Linear Logic. I have also learnt a great deal from Yves Lafont, who introduced me to Linear Logic during his "tour of duty" at Imperial. I have tried to give a self-contained account of Linear Logic in this paper, reflecting my own understanding and intuitions, but of course this is based on what I learnt from their presentations. The computational interpretation of Intuitionistic Linear Logic described in Section 3 clearly owes much to previous work by Yves Lafont [Laf88] and Sören Holmström [Hol88]. However, the key ingredients seem to me to be either new, or clarified and simplified in an essential fashion by comparison with these previous works. Similar remarks apply to the discussion of pragmatics and implementation in Section 4. The material in Section 5 is new, but should be regarded as a straightforward adaptation of similar results for other calculi. The material on Classical Linear Logic in Sections 6–8 is new; connections with proof nets are discussed at the end of Section 6.

My thanks to Phil Wadler, Phil Scott and Andre Scedrov for stimulating discussions. I would particularly like to thank Christian Retoré and Steve Vickers for many very helpful discussions on these matters; Mike Mislove for inviting me to give a talk at the MFPS Workshop at Kingston in May 1990, where I presented my preliminary work on this topic; Paul Taylor for the use of his Linear Logic font and diagram macros; and the U.K. Science and Engineering Research Council and ESPRIT Basic Research Action 3003 (the "CLICS" project) for financial support.

Since the preliminary version of this paper was released as a technical report [Abr90], I have benefited from comments and corrections from a number of people, particularly Phil

Wadler, Anne Troelstra, Ian Mackie, and especially Yves Lafont. My thanks also to Ugo Solitro and Silvio Valentini for bringing their papers [Sol89, Val89] to my attention.

2 Intuitionistic Logic and Functional Programming

This section essentially reviews standard material, although the style of presentation of the operational semantics, and the assignment of terms to proofs in the sequent calculus, are neither as widely known as they deserve to be. A good reference for general background is [GLT89].

2.1 Natural Deduction

We present Natural Deduction for Intuitionistic Logic (strictly speaking for *Minimal Logic*, but we shall not be pedantic on this point). For ease of comparison with the Sequent Calculus, we present "Natural Deduction in sequent form", in which the objects being derived are sequents

$$A_1, \ldots, A_n \vdash A.$$

(We use Γ, Δ to range over sequences of formulas, including the empty sequence; and write Γ, Δ for concatenation of sequences.) What distinguishes the system as Natural Deduction is the form of the rules for each connective: these are structured into *introduction rules* and *elimination rules*.

Axiom:

$$(\mathrm{Id}) \quad \frac{}{A \vdash A}$$

Structural Rules:

$$(\mathrm{Exchange}) \quad \frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, B, A, \Delta \vdash C}$$

$$(\mathrm{Contraction}) \quad \frac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B} \qquad (\mathrm{Weakening}) \quad \frac{\Gamma \vdash B}{\Gamma, A \vdash B}$$

Logical Rules:

$$(\wedge - I) \quad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \qquad (\wedge - E) \quad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \quad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}$$

$$(\supset - I) \quad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \supset B} \qquad (\supset - E) \quad \frac{\Gamma \vdash A \supset B \quad \Gamma \vdash A}{\Gamma \vdash B}$$

$$(\vee - I) \quad \frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \quad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \qquad (\vee - E) \quad \frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}$$

Note that all the "action" in the Natural Deduction-style logical rules is on the right hand side of the turnstile. Also notice the asymmetry between the rules for conjunction and disjunction.

Using the structural rules, it is easy to derive the following variant of $(\supset - E)$:

$$\frac{\Gamma \vdash A \supset B \quad \Delta \vdash A}{\Gamma, \Delta \vdash B}$$

and hence the Cut Rule

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

which is *not* a basic rule of Natural Deduction.

2.2 Term Assignment for Natural Deduction

We now assign terms of the typed λ-calculus to Natural Deduction proofs. From the proof-theoretic point of view, the significance of this is to give a "functional interpretation" of intuitionistic proofs; this is an embodiment of the Heyting semantics for intuitionistic logic, in which formulas (or "propositions") are interpreted by means of their proofs: a proof of a conjunction is a *pair* of proofs of the conjuncts; a proof of an implication $A \supset B$ is a (constructive) *function* mapping proofs of A to proofs of B; a proof of a disjunction $A \vee B$ is *either* a proof of A *or* a proof of B, together with the information as to which disjunct was actually proved. Thus propositions are viewed as *data types*:

$$\begin{aligned} A \wedge B &= A \times B, \quad \text{(Cartesian product)}, \\ A \supset B &= A \Rightarrow B, \quad \text{(Function space)}, \\ A \vee B &= A + B \quad \text{(Disjoint union)}. \end{aligned}$$

From the functional programming point of view, the programs (terms) have a primary interest of their own. From this perspective, what we have is a type inference system for functional programs, which assigns types to terms, rather than a logical system assigning terms to proofs. Of course, the advantage of an isomorphism is that both views can coexist harmoniously.

We present the term assigment as a version of Natural Deduction in which the objects being derived now have the form

$$x_1 : A_1, \ldots, x_n : A_n \vdash t : A$$

where the x_i are distinct variables, and t is a term. We present this system (and all others in this paper) in the style of Curry rather than Church [BH90]; *i.e.* terms have no embedded types, and can have many types assigned to them. This choice is made for technical convenience rather than necessity, and certainly does not reflect any ideological commitment.

Axiom:

$$\text{(Id)} \quad \frac{}{x : A \vdash x : A}$$

Structural Rules:

$$\text{(Exchange)} \quad \frac{\Gamma, x : A, y : B, \Delta \vdash t : C}{\Gamma, y : B, x : A, \Delta \vdash t : C}$$

$$\text{(Contraction)} \quad \frac{\Gamma, x : A, y : A \vdash t : B}{\Gamma, z : A \vdash t[z/x, z/y] : B} \qquad \text{(Weakening)} \quad \frac{\Gamma \vdash t : B}{\Gamma, z : A \vdash t : B}$$

Logical Rules:

$$(\wedge - I)\ \frac{\Gamma \vdash t : A \quad \Gamma \vdash u : B}{\Gamma \vdash \langle t, u\rangle : A \wedge B} \qquad (\wedge - E)\ \frac{\Gamma \vdash t : A \wedge B}{\Gamma \vdash \mathtt{fst}(t) : A} \quad \frac{\Gamma \vdash t : A \wedge B}{\Gamma \vdash \mathtt{snd}(t) : B}$$

$$(\supset - I)\ \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \supset B} \qquad (\supset - E)\ \frac{\Gamma \vdash t : A \supset B \quad \Gamma \vdash u : A}{\Gamma \vdash tu : B}$$

$$(\vee - I)\ \frac{\Gamma \vdash t : A}{\Gamma \vdash \mathtt{inl}(t) : A \vee B} \qquad \frac{\Gamma \vdash u : B}{\Gamma \vdash \mathtt{inr}(u) : A \vee B}$$

$$(\vee - E)\ \frac{\Gamma \vdash t : A \vee B \quad \Gamma, x : A \vdash u : C \quad \Gamma, y : B \vdash v : C}{\Gamma \vdash \mathtt{case}\ t\ \mathtt{of}\ \mathtt{inl}(x) \ \Rightarrow\ u \mid \mathtt{inr}(y) \ \Rightarrow\ v : C}$$

2.3 Operational Semantics

From the proof-theoretic point of view, the next step would be to set up an equational theory for terms, reflecting the intended notion of equivalence of proofs; and to use this to translate normalization of proofs into reduction of terms to normal forms. However, we shall proceed in a different fashion, following a method of presenting operational semantics inspired by Martin-Löf [ML84], and currently widely used under the name of Natural or Relational Semantics [Kah87, MTH90]. There are a number of reasons for this choice:

- This style of formalization of operational semantics is much better suited to specifying realistic programming languages, in which the evaluation strategy is an intrinsic part of the language, than an equational theory or term-rewriting system.
- The style is also more robust, since it extends smoothly to languages incorporating such features as lazy evaluation and general recursion, in which it is no longer the case that every program has a normal form.
- There are technical advantages, as witnessed by our work in Sections 5 and 8. The main results are considerably easier to prove. While it may be objected that they are weaker than the corresponding results for reduction to normal form, they have a wider range of applicability, to situations where the stronger results may actually fail.
- The most telling point is that the Martin-Löf style of operational semantics has what the theory of reductions for proofs significantly lacks: the evaluation rules are formally inevitable, and write themselves. By contrast, the "commutative conversions" e.g. for disjunction are unmemorable and awkward. Moreover, the evaluation rules capture what is actually done in a computation.

Before presenting the operational semantics for typed λ-calculus, we shall explain the general concepts underlying this approach. Firstly, there is a classification of constructions on terms into two groups: *constructors* (corresponding to introduction rules) and *destructors* (corresponding to elimination rules). (This classification of course goes right back to the pioneers McCarthy [McC63] and Landin [Lan64], although they lacked the proof-theoretic perspective.) Constructors *produce* information (pieces of structured data); destructors *consume* it. The basic unit of computation (reduction step) is when a destructor meets a corresponding constructor; the author has found it suggestive to think of particles of information and anti-information colliding and annihilating each other, possibly generating some new particles—a communication event. Note that the operational significance of type checking is precisely to ensure that a constructor of one type never collides with a destructor of a different type—*i.e.* that the consumer can always plug in to the producer, and communication occur.

We think of computation as applying only to *programs*, *i.e.* closed terms. The overall effect of a computation is to reduce a program to a *canonical form*, in which some quantity of information has been made explicit, by being put into constructor form. At this point, a bifurcation occurs, between Lazy (including call-by-name) Evaluation—the principle of producing as *little* information as possible at each stage of evaluation—and Eager (including call-by-value) Evaluation, in which *as much* as possible is produced. Each of these determines an evaluation strategy, and hence an operational semantics. The proof system in itself does *not* enforce a strategy on us. This is not too surprising, since *extensional* differences between these strategies only show up in the λ-calculus in the presence of non-terminating programs (see e.g. [Plo75]), and under the strict correspondence of typed programs with proofs, we have Strong Normalization, so that all programs—and indeed all strategies—terminate.

(However, it is one of the most notable features of Linear Logic that a clear perspective on Lazy vs. Eager Evaluation *is* provided there, even at the pure logical level, and in the absence of divergence. See Section 3.)

For each of the Lazy and Eager strategies we shall specify a set of canonical forms, and present the operational semantics in terms of an *evaluation relation* $t \Downarrow c$, to be read as "t evaluates (or converges) to canonical form c".

Lazy Evaluation

Here the canonical forms are all programs (closed terms) with a constructor at the top.

Canonical forms:

$$\lambda x.t \quad \langle t, u \rangle \quad \mathtt{inl}(t) \quad \mathtt{inr}(u)$$

Evaluation relation:

This is defined inductively, as the least satisfying the following clauses:

$$\frac{}{\langle t, u \rangle \Downarrow \langle t, u \rangle} \qquad \frac{t \Downarrow \langle u, v \rangle \quad u \Downarrow c}{\mathtt{fst}(t) \Downarrow c} \qquad \frac{t \Downarrow \langle u, v \rangle \quad v \Downarrow c}{\mathtt{snd}(t) \Downarrow c}$$

$$\frac{}{\lambda x.t \Downarrow \lambda x.t} \qquad \frac{t \Downarrow \lambda x.v \quad v[u/x] \Downarrow c}{tu \Downarrow c}$$

$$\frac{}{\mathtt{inl}(t) \Downarrow \mathtt{inl}(t)} \qquad \frac{}{\mathtt{inr}(u) \Downarrow \mathtt{inr}(u)}$$

$$\frac{t \Downarrow \mathtt{inl}(w) \quad u[w/x] \Downarrow c}{\mathtt{case}\, t\, \mathtt{of}\, \mathtt{inl}(x) \Rightarrow u \mid \mathtt{inr}(y) \Rightarrow v \Downarrow c} \qquad \frac{t \Downarrow \mathtt{inr}(w) \quad v[w/y] \Downarrow c}{\mathtt{case}\, t\, \mathtt{of}\, \mathtt{inl}(x) \Rightarrow u \mid \mathtt{inr}(y) \Rightarrow v \Downarrow c}$$

Eager Evaluation

Canonical forms:

$$\lambda x.t \quad \langle c, d\rangle \quad \mathtt{inl}(c) \quad \mathtt{inr}(d)$$

where c, d are canonical forms.

Note that *any* abstraction $\lambda x.t$ is canonical, since we only evaluate programs, and t may not be closed. This exactly mirrors what is done in actual eager evaluation languages, e.g. ML [MTH90].

Evaluation relation:

$$\frac{t \Downarrow c \quad u \Downarrow d}{\langle t, u\rangle \Downarrow \langle c, d\rangle} \qquad \frac{t \Downarrow \langle c, d\rangle}{\mathtt{fst}(t) \Downarrow c} \qquad \frac{t \Downarrow \langle c, d\rangle}{\mathtt{snd}(t) \Downarrow d}$$

$$\frac{}{\lambda x.t \Downarrow \lambda x.t} \qquad \frac{t \Downarrow \lambda x.v \quad u \Downarrow c \quad v[c/x] \Downarrow d}{tu \Downarrow d}$$

$$\frac{t \Downarrow c}{\mathtt{inl}(t) \Downarrow \mathtt{inl}(c)} \qquad \frac{u \Downarrow d}{\mathtt{inr}(u) \Downarrow \mathtt{inr}(d)}$$

$$\frac{t \Downarrow \mathtt{inl}(c) \quad u[c/x] \Downarrow d}{\mathtt{case}\ t\ \mathtt{of}\ \mathtt{inl}(x) \Rightarrow u \mid \mathtt{inr}(y) \Rightarrow v \Downarrow d} \qquad \frac{t \Downarrow \mathtt{inr}(c) \quad v[c/y] \Downarrow d}{\mathtt{case}\ t\ \mathtt{of}\ \mathtt{inl}(x) \Rightarrow u \mid \mathtt{inr}(y) \Rightarrow v \Downarrow d}$$

2.4 Sequent Calculus

We now review the Sequent Calculus presentation of Intuitionistic Logic. The objects derived in this calculus are exactly the same sequents $\Gamma \vdash A$ as in (our version of) Natural Deduction. The difference appears in the form of the rules. Firstly, the Cut Rule

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

is taken as primitive in Sequent Calculus. The Axiom and Structural Rules are as before. The logical rules are different: they are structured into *left* and *right* rules rather than introduction and elimination rules. The right rules are the same as the introduction rules of Natural Deduction. The left rules introduce the principal connective on the left of the turnstile.

Logical Rules:

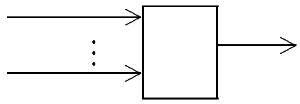
$$(\wedge - R) \quad \frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \wedge B} \qquad (\wedge - L) \quad \frac{\Gamma, A \vdash C}{\Gamma, A \wedge B \vdash C} \quad \frac{\Gamma, B \vdash C}{\Gamma, A \wedge B \vdash C}$$

$$(\supset - R) \quad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \supset B} \qquad (\supset - L) \quad \frac{\Gamma \vdash A \quad B, \Delta \vdash C}{\Gamma, A \supset B, \Delta \vdash C}$$

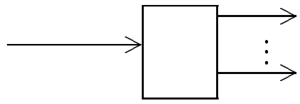
$$(\vee - R) \quad \frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \quad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \qquad (\vee - L) \quad \frac{\Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma, A \vee B \vdash C}$$

The importance of Sequent Calculus and the symmetries it brings to light has been forcefully argued by Girard [GLT89]. These symmetries are only partially on view in the intuitionistic sequent calculus, which incorporates an important *asymmetry*, in that only a single formula can appear on the right hand side of the turnstile. This is intimately linked with the possibility of a functional interpretation for intuitionistic logic, since it corresponds to the asymmetric nature of functions with respect to inputs (premises) and outputs (conclusions).

However, the symmetry that does exist between the left and right rules can be nicely related to our earlier discussion of constructors and destructors. Constructors (generated by right rules) build structure on the output



while destructors (generated by left rules) decompose structure on the input



The most familiar instance of this latter pattern is the *conditional*, which is generalized to the **case** statement in typed λ-calculus; more generally yet, destructors correspond to *pattern-matching*, which has become an important feature of functional programming languages [BW88, FH88, Tur90]. For function types, the destructor is application, which decomposes a function into its graph.

Term Assignment for Sequent Calculus

We now show how terms can be assigned to proofs in the Sequent Calculus. The terms being assigned and the form of the sequents are of course exactly the same as for Natural Deduction. The point is to show the actual assignments corresponding to the Sequent Calculus rules. Although in principle these follow automatically from the known translations of Sequent Calculus into Natural Deduction (see e.g. [GLT89]), they do not seem to be as well known as they might be.

Cut Rule:

$$\frac{\Gamma \vdash t : A \quad x : A, \Delta \vdash u : B}{\Gamma, \Delta \vdash u[t/x] : B}$$

Logical Rules:

$$(\wedge - R) \quad \frac{\Gamma \vdash t : A \quad \Delta \vdash u : B}{\Gamma, \Delta \vdash \langle t, u \rangle : A \wedge B}$$

$$(\wedge - L) \quad \frac{\Gamma, x : A \vdash t : C}{\Gamma, z : A \wedge B \vdash t[\mathtt{fst}(z)/x] : C} \quad \frac{\Gamma, y : B \vdash u : C}{\Gamma, z : A \wedge B \vdash u[\mathtt{snd}(z)/y] : C}$$

$$(\supset - R) \quad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \supset B} \qquad (\supset - L) \quad \frac{\Gamma \vdash t : A \quad x : B, \Delta \vdash u : C}{\Gamma, f : A \supset B, \Delta \vdash u[(ft)/x] : C}$$

$$(\vee - R) \quad \frac{\Gamma \vdash t : A}{\Gamma \vdash \mathtt{inl}(t) : A \vee B} \quad \frac{\Gamma \vdash u : B}{\Gamma \vdash \mathtt{inr}(u) : A \vee B}$$

$$(\vee - L) \quad \frac{\Gamma, x : A \vdash u : C \quad \Gamma, y : B \vdash v : C}{\Gamma, z : A \vee B \vdash \texttt{case}\ z\ \texttt{of}\ \texttt{inl}(x) \Rightarrow u \mid \texttt{inr}(y) \Rightarrow v : C}$$

Note that terms generated by cut-free proofs are in *normal form*; in particular, terms generated by the left rules have variables in the head position, so no redexes are created. Redexes only arise as a result of the substitutions performed by applications of the Cut rule. Thus all computation is concentrated into the process of cut elimination.

3 Intuitionistic Linear Logic

The basic idea of Linear Logic [Gir87] is to control the use of resources. In the functional framework, a resource may be taken to be a piece of information—of data—supplied as an input to a computation. The structural rules of Intuitionistic Logic (excluding the trivial Exchange rule) allow us to *copy* resources (Contraction) and to *discard* them (Weakening):

$$\text{(Contraction)} \quad \frac{\Gamma, x : A, y : A \vdash t : B}{\Gamma, z : A \vdash t[z/x, z/y] : B} \qquad \text{(Weakening)} \quad \frac{\Gamma \vdash t : B}{\Gamma, z : A \vdash t : B}$$

Specifically, Contraction allows multiple occurrences of a variable to appear in the proof term, while weakening allows variables to be introduced as premises which do not appear in the proof term at all.

Linear Logic arises by dropping these two structural rules. This means that each input must be used *exactly once* in producing the output. This has immediate implications for the interpretation of the logical connectives. Firstly, we find that two distinct interpretations of conjunction, or in programming terms a type of *pairs* of values, arise:

- If we wish to use *both* components of the pair, on the unique occasion when we use the input, then we lose the ability to project. This leads to the mutiplicative version of conjunction, the tensor product $A \otimes B$.

- If we wish to project, then on the unique occasion when we use the input, we must choose to take *either* the first *or* the second projection, and this is the only part of the input we will ever see. So the additive conjunction $A \,\&\, B$ appears as a kind of *choice*—an "external" choice in the terminology of CSP [Hoa85], since it is made by the consumer of the datum.

- The disjoint sum (additive disjunction) $A \oplus B$ appears as an *internal* choice, since it is at the discretion of the producer of the datum whether the choice is made from A—a value of the form $\texttt{inl}(t)$, or from B—a value of the form $\texttt{inr}(u)$.

- The linear implication $A \multimap B$ is the type of functions which use their argument exactly once, internalizing linear inference.

These connectives by themselves are far too weak to provide useful expressive power. This is regained by reintroducing weakening and contraction in a controlled form, not as omnipresent structural rules, but reflected into a datatype, the exponential $!A$ ("Of course A"). The effect is to make as many copies of a value of type A available as may be needed.

That we have recovered adequate expressive power is witnessed by the fact that Intuitionistic Logic can be interpreted in Linear Logic with the above connectives. In particular, intuitionistic implication is recovered by

$$A \supset B = !A \multimap B.$$

This decomposition of implication (in programming terms, of the function type) is one of the most interesting aspects of Linear Logic.

We now flesh out these intuitive ideas by giving the Sequent Calculus formalization of Intuitionistic Linear Logic.

Axiom:

$$(\mathrm{Id}) \quad \frac{}{A \vdash A}$$

Structural Rule:

$$(\mathrm{Exchange}) \quad \frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, B, A, \Delta \vdash C}$$

Cut Rule:

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

Logical Rules:

$$(\mathbf{1} - R) \quad \frac{}{\vdash \mathbf{1}} \qquad (\mathbf{1} - L) \quad \frac{\Gamma \vdash A}{\Gamma, \mathbf{1} \vdash A}$$

$$(\otimes - R) \quad \frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B} \qquad (\otimes - L) \quad \frac{\Gamma, A, B \vdash C}{\Gamma, A \otimes B \vdash C}$$

$$(\multimap - R) \quad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B} \qquad (\multimap - L) \quad \frac{\Gamma \vdash A \quad B, \Delta \vdash C}{\Gamma, A \multimap B, \Delta \vdash C}$$

$$(\& - R) \quad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \& B} \qquad (\& - L) \quad \frac{\Gamma, A \vdash C}{\Gamma, A \& B \vdash C} \quad \frac{\Gamma, B \vdash C}{\Gamma, A \& B \vdash C}$$

$$(\oplus - R) \quad \frac{\Gamma \vdash A}{\Gamma \vdash A \oplus B} \quad \frac{\Gamma \vdash B}{\Gamma \vdash A \oplus B} \qquad (\oplus - L) \quad \frac{\Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma, A \oplus B \vdash C}$$

$$(\,! - R) \quad \frac{!\Gamma \vdash A}{!\Gamma \vdash !A} \qquad (\mathrm{Dereliction}) \quad \frac{\Gamma, A \vdash B}{\Gamma, !A \vdash B}$$

($!\Gamma$ means a sequence of the form $!A_1, \ldots, !A_k$.)

$$(\mathrm{Contraction}) \quad \frac{\Gamma, !A, !A \vdash B}{\Gamma, !A \vdash B} \qquad (\mathrm{Weakening}) \quad \frac{\Gamma \vdash B}{\Gamma, !A \vdash B}$$

The essence of the distinction between the additive and multiplicative connectives is conveyed by the fact that *different* contexts are combined, without interaction, in $(\otimes - R)$, $(\multimap - L)$, while the *same* contexts are used in $(\& - R)$ and $(\oplus - L)$. This reflects the fact that in the additives a choice is made, between one of the components of a pair for $\&$, or one of the guards in a `case` for $\oplus$; so the *same* inputs must be used in *both* alternatives, to ensure that each input is used exactly once in producing the output. Note also the key rôle of the left rules in defining the !-type; the proper treatment of these rules is the technical crux in our computational interpretation of Intuitionistic Linear Logic.

The Linear Term Calculus

We now give an assigment of terms to proofs in Intuitionistic Linear Logic. The calculus from which these terms are drawn is a refinement of the λ-calculus. A key rôle is played by pattern-matching constructs, corresponding to left rules of the logic. One—minor—complication is that the syntax of terms is not context-free, but must reflect *linearity constraints*, *i.e.* syntactic constraints on occurrences of variables corresponding to the semantic constraint that inputs are used exactly once.

To formalize these notions, it is convenient to use an auxiliary syntactic category of patterns. We use X, Y, Z to range over finite sets of variables. Now $\mathcal{P}_X$, the set of patterns with variables in X, is defined as follows:

$$*, \_ \in \mathcal{P}_\varnothing \qquad \langle x, \_\rangle, \langle \_, x\rangle, !x \in \mathcal{P}_{\{x\}} \qquad x \otimes y, x@y \in \mathcal{P}_{\{x,y\}}$$

We can now define $\mathcal{T}_X$, the linear terms with free variables in X, inductively as follows:

- $x \in \mathcal{T}_{\{x\}}$
- $* \in \mathcal{T}_\varnothing$
- $t \in \mathcal{T}_X, u \in \mathcal{T}_Y, X \cap Y = \varnothing \implies t \otimes u, tu \in \mathcal{T}_{X \cup Y}$
- $t, u \in \mathcal{T}_X \implies \langle t, u\rangle \in \mathcal{T}_X$
- $t \in \mathcal{T}_X \implies \mathtt{inl}(t), \mathtt{inr}(t), !t \in \mathcal{T}_X$
- $t \in \mathcal{T}_{X \cup \{x\}}, x \notin X \implies \lambda x.t \in \mathcal{T}_X$
- $t \in \mathcal{T}_X, p \in \mathcal{P}_Y, u \in \mathcal{T}_{Y \cup Z}, X \cap Z = Y \cap Z = \varnothing$
  $\implies \mathtt{let}\ t\ \mathtt{be}\ p\ \mathtt{in}\ u \in \mathcal{T}_{X \cup Z}$
- $t \in \mathcal{T}_X, u \in \mathcal{T}_{Z \cup \{x\}}, v \in \mathcal{T}_{Z \cup \{y\}}, X \cap Z = \{x, y\} \cap Z = \varnothing$
  $\implies \mathtt{case}\ t\ \mathtt{of}\ \mathtt{inl}(x) \Rightarrow u \mid \mathtt{inr}(y) \Rightarrow v \in \mathcal{T}_{X \cup Z}$

We now present the assignment of linear terms to proofs in Intuitionistic Linear Logic, in the same style as the term assigment for Sequent Calculus given in the previous section. Our sequents now have the form

$$x_1 : A_1, \ldots, x_k : A_k \vdash t : A$$

where the A_i are linear formulae (built from the connectives $\mathbf{1}$, $\otimes$, $\multimap$, $\&$, $\oplus$, $!$), the x_i are distinct variables, and $t \in \mathcal{T}_X$, $X = \{x_1, \ldots, x_k\}$. Note that the rules presented below are subject to the implicit constraint that the linearity conditions for well-formedness of terms are satisfied. This constraint can always be met, e.g. by using distinct variables for all instances of the Axiom.

Axiom:

$$(\mathrm{Id}) \quad \frac{}{x : A \vdash x : A}$$

Structural Rule:

$$(\mathrm{Exchange}) \quad \frac{\Gamma, x : A, y : B, \Delta \vdash t : C}{\Gamma, y : B, x : A, \Delta \vdash t : C}$$

Cut Rule:

$$\frac{\Gamma \vdash t : A \quad x : A, \Delta \vdash u : B}{\Gamma, \Delta \vdash u[t/x] : B}$$

Logical Rules:

$$(\mathbf{1}-R)\ \frac{}{\vdash * : \mathbf{1}} \qquad (\mathbf{1}-L)\ \frac{\Gamma \vdash t : A}{\Gamma, z : \mathbf{1} \vdash \mathtt{let}\ z\ \mathtt{be}\ *\ \mathtt{in}\ t : A}$$

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

$$(\multimap - R)\ \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B} \qquad (\multimap - L)\ \frac{\Gamma \vdash t : A \quad x : B, \Delta \vdash u : C}{\Gamma, f : A \multimap B, \Delta \vdash u[(ft)/x] : C}$$

$$(\& - R)\ \frac{\Gamma \vdash t : A \quad \Gamma \vdash u : B}{\Gamma \vdash \langle t, u \rangle : A \& B}$$

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

$$(\oplus - R)\ \frac{\Gamma \vdash t : A}{\Gamma \vdash \mathtt{inl}(t) : A \oplus B} \qquad \frac{\Gamma \vdash u : B}{\Gamma \vdash \mathtt{inr}(u) : A \oplus B}$$

$$(\oplus - L)\ \frac{\Gamma, x : A \vdash u : C \quad \Gamma, y : B \vdash v : C}{\Gamma, z : A \oplus B \vdash \mathtt{case}\ z\ \mathtt{of}\ \mathtt{inl}(x)\ \Rightarrow\ u \mid \mathtt{inr}(y)\ \Rightarrow\ v : C}$$

$$(\,!-R)\ \frac{!\Gamma \vdash t : A}{!\Gamma \vdash !t : !A}$$

$$(\text{Dereliction})\ \frac{\Gamma, x : A \vdash t : B}{\Gamma, z : !A \vdash \mathtt{let}\ z\ \mathtt{be}\ !x\ \mathtt{in}\ t : B}$$

$$(\text{Contraction})\ \frac{\Gamma, x : !A, y : !A \vdash t : B}{\Gamma, z : !A \vdash \mathtt{let}\ z\ \mathtt{be}\ x@y\ \mathtt{in}\ t : B}$$

$$(\text{Weakening})\ \frac{\Gamma \vdash t : B}{\Gamma, z : !A \vdash \mathtt{let}\ z\ \mathtt{be}\ \_\ \mathtt{in}\ t : B}$$

Operational Semantics

We now give an operational semantics for the linear term calculus in exactly the same style, and with the same supporting intuitions as the semantics of the λ-calculus given in the previous section. However, one notable difference emerges, as immediate evidence of the more refined computational content of the Linear types. Whereas Intuitionistic Logic was perfectly neutral as to which evaluation strategy to adopt for the λ-calculus, in Linear Logic the logical structure of the types gives a clear indication as to which form of evaluation to employ:

- For a term of tensor type $A \otimes B$ we know that any consumer (e.g. a destructor context `let` $[\cdot]$ `be` $x \otimes y$ `in` u) will evaluate this term to a pair, and *use both components*. This clearly indicates eager evaluation. Similarly, a term of type $A \multimap B$ must evaluate to an abstraction, which when applied to any argument will evaluate it exactly once. Evaluation of the argument exactly once is the slogan of call-by-value [Myc80]. Finally, any consumer of a term of type $A \oplus B$ will evaluate it to a term of the form $\mathtt{inl}(t)$ or $\mathtt{inr}(t)$, and then use t in evaluating the appropriate arm of a `case` statement; so once again, eager evaluation is indicated.
- On the other hand, a value of type $A \& B$ will evaluate to a pair, *exactly one component of which* will be used in any given context. Since we cannot predict which component will be used, it is clear that evaluating either component in advance of their actual use will lead in general to redundant computation. Thus lazy evaluation is indicated here. Again, a value of type $!A$ may be discarded altogether, so evaluation in advance of actual use may lead to redundant computation, and lazy evaluation is indicated.

Thus we get a classification:

- $\otimes, \multimap, \oplus$ Eager Evaluation
- $\&, !$ Lazy Evaluation

What is particularly interesting is that when we interpret the intuitionistic types

$$\begin{aligned} A \times B &= A \& B \\ A \Rightarrow B &= !A \multimap B \end{aligned}$$

we see that the intuitionistic function type will be operationally *call-by-name* (Lazy), since its argument is "frozen" by the lazy evaluation of the $!A$ type. So the mixed evaluation strategy of the Linear types, incorporating a high degree of eager evaluation, supports lazy evaluation at the higher level of the intuitionistic types. One might say that this gives a rational reconstruction, in logical terms, of the standard method for implementing lazy evaluation on top of an eager evaluation strategy, as introduced by Landin [Lan65], and used in the SECD and CAM machines [Hen80, Hue90]. This idea is standardly modelled in denotational semantics by *lifting* [Plo85], *i.e.*

$$A \Rightarrow B = A_{\perp} \rightharpoonup B$$

where $A \rightharpoonup B$ is the type of *partial* (or alternatively *strict*) functions. This account *requires* the presence of divergent programs; the Linear decomposition

$$A \Rightarrow B = !A \multimap B$$

does not.

With these motivating remarks, we now present the operational semantics.

Canonical forms

$$\langle t, u \rangle \quad !t$$

$$* \quad c \otimes d \quad \lambda x.t \quad \mathtt{inl}(c) \quad \mathtt{inr}(d)$$

where c, d are canonical.

Evaluation relation

$$\frac{}{*\Downarrow *} \qquad \frac{t\Downarrow * \quad u\Downarrow c}{\mathtt{let}\ t\ \mathtt{be}\ *\ \mathtt{in}\ u\Downarrow c}$$

$$\frac{t\Downarrow c \quad u\Downarrow d}{t\otimes u\Downarrow c\otimes d} \qquad \frac{t\Downarrow c\otimes d \quad u[c/x, d/y]\Downarrow e}{\mathtt{let}\ t\ \mathtt{be}\ x\otimes y\ \mathtt{in}\ u\Downarrow e}$$

$$\frac{}{\lambda x.t\Downarrow \lambda x.t} \qquad \frac{t\Downarrow \lambda x.v \quad u\Downarrow c \quad v[c/x]\Downarrow d}{tu\Downarrow d}$$

$$\frac{}{\langle t, u\rangle\Downarrow\langle t, u\rangle} \qquad \frac{t\Downarrow\langle v, w\rangle \quad v\Downarrow c \quad u[c/x]\Downarrow d}{\mathtt{let}\ t\ \mathtt{be}\ \langle x, \_\rangle\ \mathtt{in}\ u\Downarrow d} \qquad \frac{t\Downarrow\langle v, w\rangle \quad w\Downarrow c \quad u[c/y]\Downarrow d}{\mathtt{let}\ t\ \mathtt{be}\ \langle \_, y\rangle\ \mathtt{in}\ u\Downarrow d}$$

$$\frac{t\Downarrow c}{\mathtt{inl}(t)\Downarrow\mathtt{inl}(c)} \qquad \frac{u\Downarrow d}{\mathtt{inr}(u)\Downarrow\mathtt{inr}(d)}$$

$$\frac{t\Downarrow\mathtt{inl}(c) \quad u[c/x]\Downarrow d}{\mathtt{case}\ t\ \mathtt{of}\ \mathtt{inl}(x)\ \Rightarrow\ u \mid \mathtt{inr}(y)\ \Rightarrow\ v\Downarrow d} \qquad \frac{t\Downarrow\mathtt{inr}(c) \quad v[c/y]\Downarrow d}{\mathtt{case}\ t\ \mathtt{of}\ \mathtt{inl}(x)\ \Rightarrow\ u \mid \mathtt{inr}(y)\ \Rightarrow\ v\Downarrow d}$$

$$\frac{}{!t\Downarrow !t} \qquad \frac{t\Downarrow !v \quad v\Downarrow c \quad u[c/x]\Downarrow d}{\mathtt{let}\ t\ \mathtt{be}\ !x\ \mathtt{in}\ u\Downarrow d}$$

$$\frac{t\Downarrow !v \quad u\Downarrow c}{\mathtt{let}\ t\ \mathtt{be}\ \_\ \mathtt{in}\ u\Downarrow c} \qquad \frac{t\Downarrow !v \quad u[!v/x, !v/y]\Downarrow c}{\mathtt{let}\ t\ \mathtt{be}\ x@y\ \mathtt{in}\ u\Downarrow c}$$

These rules codify the previous discussion in a very direct fashion. Note that the rules corresponding to Contraction and Weakening respectively copy and discard their inputs.

4 Pragmatics and Implementation

In this section, we sketch some of the promising applications to program analysis and optimization opened up by the computational interpretation of Intuitionistic Linear Logic presented in the previous section. We also describe an SECD machine implementation of the Linear calculus.

Logical Compilation

We have already mentioned the translation of Intuitionistic Logic into Intuitionistic Linear Logic. The full translation of formulas is as follows:

$$\begin{aligned}
(A \supset B)^\circ &= !A^\circ \multimap B^\circ \\
(A \wedge B)^\circ &= A^\circ \mathbin{\&} B^\circ \\
(A \vee B)^\circ &= !A^\circ \oplus !B^\circ
\end{aligned}$$

It is important to note that the translation works not just at the level of theorems, or even provable sequents, but of *proofs*. That is, every proof of a sequent $\Gamma \vdash A$ in Intuitionistic Sequent Calculus can be translated into a proof of $!\Gamma^\circ \vdash A^\circ$ in Intuitionistic Linear Logic. This in turn induces a translation from λ-terms into linear terms. This throws up an interesting aspect of the Curry-Howard isomorphism which I have not seen discussed in the literature: an interpretation of one logic L_1 into another L_2 at the level of proofs induces a *compiler* from the programs corresponding to proofs in L_1 to programs of L_2. Of course, the correctness of this compiler should follow directly from logical properties of the interpretation. In particular, it seems appropriate to speak of compiling λ-terms into linear terms, since the linear types are finer-grained, and "lower-level" in the programming sense.

We will not describe the translation explicitly here; see [Gir87] for details. The main points will be sufficiently clear from the following examples.

Examples. Consider the $\mathbf{S}$ and $\mathbf{K}$ combinators:

$$\mathbf{S} = \lambda f, g, x.(fx)(gx), \qquad \mathbf{K} = \lambda x, y.x.$$

Neither of these terms is linear: $\mathbf{S}$ uses x twice, $\mathbf{K}$ discards y. However, consider the following linear terms:

$$\mathbf{S}' = \lambda f, g, z.\mathtt{let}\ z\ \mathtt{be}\ x@y\ \mathtt{in}\ (fx)(gy), \qquad \mathbf{K}' = \lambda x, z.\mathtt{let}\ z\ \mathtt{be}\ \_\ \mathtt{in}\ x.$$

We can derive the following typing judgements:

$$\vdash \mathbf{S}' : (\,!\alpha \multimap \beta \multimap \gamma) \multimap (\,!\alpha \multimap \beta) \multimap !\alpha \multimap \gamma$$
$$\vdash \mathbf{K}' : \alpha \multimap !\beta \multimap \alpha$$

Logic-based Program Analysis

The above examples illustrate a further point: the typings obtained are the most general for the given terms. They "optimize" the types which would be obtained by the uniform translation of $\mathbf{S}$ and $\mathbf{K}$ into Linear Logic, *i.e.*

$$\begin{aligned} ((\alpha \supset \beta \supset \gamma) \supset (\alpha \supset \beta) \supset \alpha \supset \gamma)^\circ &= !(\,!\alpha \multimap !\beta \multimap \gamma) \multimap (\,!\alpha \multimap \beta) \multimap !\alpha \multimap \gamma \\ (\alpha \supset (\beta \supset \alpha))^\circ &= !\alpha \multimap !\beta \multimap \alpha. \end{aligned}$$

By introducing fewer !-types, we increase the possibilites of eager (possibly parallel) evaluation; of course, the justification for these "improved" typings is that e.g. in $\mathbf{K}'$, x is actually used. But this is exactly the kind of information that *strictness analysis* tries to extract [AH87].

There are other uses for the information made explicit by the Linear types. If we know that a value is never *shared* (as it will not be *unless* it is of a !-type) then we can safely update it in place on the (necessarily unique) occasion when we access it; so we get a handle on "in-place update analysis" [AH87]. We can also consider refinements of the !-type, in which instances of ! are indexed by expressions which describe patterns of usage in more precise ways. This has been done with reference to complexity theory in [GSS90], where a system of Bounded Linear Logic is described, in which ! is graded by "resource polynomials". This leads to a term assignment in which exactly the polynomial-time computable functions are typable. Again, there should be connections with "complexity analysis" as in [San89].

The general framework suggested by these ideas might be called "Logic-based Program Analysis", by analogy with the already well-established subject of Semantics-Based Program Analysis [AH87, GJ86]. (Of course, the two approaches should be complementary.) The real applicability of this approach remains to be demonstrated, but it looks genuinely promising, and a number of researchers have already made preliminary investigations along these lines [Laf88, Hol88, Wad90]. Our contribution is to suggest that the linear term calculus introduced in the previous section may form a good medium for performing static analysis and optimization. One may start from a standard functional program, translate it by the uniform method into the linear calculus, and then try to "linearize" it, *i.e.* to minimize the usage of the exponential types. This should provide a sound basis for performing many useful optimizations.

The Linear SECD Machine

We will now describe an implementation of the linear term calculus by a variant of the SECD machine [Lan64]. Although some possibilities for parallel evaluation of the calculus do exist, the implementation we shall describe is purely sequential. (In any case, the potential for parallel execution is much greater for Classical Linear Logic, which is treated in Sections 6–8.)

We will follow the very lucid exposition of the (standard) SECD machine given in [Hen80] fairly closely. The machine is based on a list-structured store. We shall use Turner's notation [Tur85] for list operations:

$[]$	for the empty list
$x : l$	for infix cons
$[x_1, \ldots, x_n]$	for $x_1 : \cdots : x_n : []$.

The objects manipulated by the machine are inductively defined as follows:

- A *code* is a list of instructions.
- An *instruction* is one of the forms

PUSHENV	HD	TL
RET	PUSH	POP
MAKEFCL(c)	AP	
UNIT UNUNIT	PAIR	UNPAIR
MAKECCL(c_1, c_2)	FST	SND
INL	INR	CASE(c_1, c_2)
MAKEOCL(c)	READ	DUP

  where c, c_1, c_2 are codes.

- A *value* is one of the forms

$$* \quad (v_1, v_2) \quad \texttt{inl}(v) \quad \texttt{inr}(v) \quad \texttt{fcl}(c, e) \quad \texttt{ccl}(c_1, c_2, e) \quad \texttt{ocl}(c, e)$$

  where v, v_1, v_2 are values, c, c_1, c_2 are codes, and e is an environment. (Values of the form $\texttt{fcl}(c, e)$, $\texttt{ccl}(c_1, c_2, e)$, $\texttt{ocl}(c, e)$ are called *function*, *choice* and *of course* closures respectively.)

- An *environment* is a list of values.

The state of the machine is determined by four registers, s, e, c, d:

- s is the current expression evaluation stack; evaluation terminates with the resulting value at the top of the stack.
- e is the environment giving the values of the free variables of the current expression.
- c is the code corresponding to the current expression.
- d is a *dump*, *i.e.* a stack of suspended procedure activations, represented as $[s, e, c]$ triples.

The operation of the machine is described by transition rules specifying the effect of each instruction; see Figure 1. Note that each instruction can obviously be implemented in constant time on a conventional machine.

We can now define a compiler from linear terms to SECD codes. More precisely, we define a function $t \star l$ which, for each linear term $t \in \mathcal{T}_X$ and list of variables l such that every variable in X occurs in l, yields a code for the Linear SECD machine.

The definition is by induction on the structure of t. (Notation: we use infix $\mid$ for list concatenation.)

$$x \star l = [\text{PUSHENV}] \mid \underbrace{[\text{TL}, \ldots, \text{TL}]}_{n} \mid [\text{HD}] \tag{1}$$

where n is the index of the first occurence of x in l (starting from 0).

$$* \star l = [\text{UNIT}] \tag{2}$$

$$\texttt{let}\ t\ \texttt{be}\ *\ \texttt{in}\ u \star l = t \star l \mid [\text{UNUNIT}] \mid u \star l \tag{3}$$

$$t \otimes u \star l = t \star l \mid u \star l \mid [\text{PAIR}] \tag{4}$$

$$\texttt{let}\ t\ \texttt{be}\ x \otimes y\ \texttt{in}\ u \star l = t \star l \mid [\text{UNPAIR}, \text{PUSH}, \text{PUSH}] \mid u \star x : y : l \mid [\text{POP}, \text{POP}] \tag{5}$$

$$\lambda x.t \star l = [\text{MAKEFCL}(t \star x : l \mid [\text{POP}, \text{RET}])] \tag{6}$$

$$tu \star l = u \star l \mid [\text{PUSH}] \mid t \star l \mid [\text{AP}] \tag{7}$$

$$
\begin{array}{lcl}
s, e, \text{PUSHENV} : c, d & \longrightarrow & e : s, e, c, d \\
(v : l) : s, e, \text{HD} : c, d & \longrightarrow & v : s, e, c, d \\
(v : l) : s, e, \text{TL} : c, d & \longrightarrow & l : s, e, c, d \\
v : s, e, \text{RET} : c, [s', e', c'] : d & \longrightarrow & v : s', e', c', d \\
v : s, e, \text{PUSH} : c, d & \longrightarrow & s, v : e, c, d \\
s, v : e, \text{POP} : c, d & \longrightarrow & s, e, c, d \\
s, e, \text{MAKEFCL}(c') : c, d & \longrightarrow & \mathtt{fcl}(c', e) : s, e, c, d \\
s, e, \text{UNIT} : c, d & \longrightarrow & * : s, e, c, d \\
* : s, e, \text{UNUNIT} : c, d & \longrightarrow & s, e, c, d \\
v : w : s, e, \text{PAIR} : c, d & \longrightarrow & (v, w) : s, e, c, d \\
(v, w) : s, e, \text{UNPAIR} : c, d & \longrightarrow & v : w : s, e, c, d \\
\mathtt{fcl}(c', e') : s, v : e, \text{AP} : c, d & \longrightarrow & [], v : e', c', [s, e, c] : d \\
s, e, \text{MAKECCL}(c_1, c_2) : c, d & \longrightarrow & \mathtt{ccl}(c_1, c_2, e) : s, e, c, d \\
\mathtt{ccl}(c_1, c_2, e') : s, e, \text{FST} : c, d & \longrightarrow & [], e', c_1, [s, e, c] : d \\
\mathtt{ccl}(c_1, c_2, e') : s, e, \text{SND} : c, d & \longrightarrow & [], e', c_2, [s, e, c] : d \\
v : s, e, \text{INL} : c, d & \longrightarrow & \mathtt{inl}(v) : s, e, c, d \\
v : s, e, \text{INR} : c, d & \longrightarrow & \mathtt{inr}(v) : s, e, c, d \\
\mathtt{inl}(v) : s, e, \text{CASE}(c_1, c_2) : c, d & \longrightarrow & [], v : e, c_1, [s, e, c] : d \\
\mathtt{inr}(v) : s, e, \text{CASE}(c_1, c_2) : c, d & \longrightarrow & [], v : e, c_2, [s, e, c] : d \\
s, e, \text{MAKEOCL}(c') : c, d & \longrightarrow & \mathtt{ocl}(c', e) : s, e, c, d \\
\mathtt{ocl}(c', e') : s, e, \text{READ} : c, d & \longrightarrow & [], e', c', [s, e, c] : d \\
v : s, e, \text{DUP} : c, d & \longrightarrow & v : v : s, e, c, d
\end{array}
$$

Figure 1: Linear SECD machine transitions

$$\langle t, u\rangle \star l = [\textsc{makeccl}(t \star l \mid [\textsc{ret}]], u \star l \mid [\textsc{ret}]) \tag{8}$$

$$\texttt{let}\ t\ \texttt{be}\ \langle x, \_\rangle\ \texttt{in}\ u \star l = t \star l \mid [\textsc{fst}, \textsc{push}] \mid u \star x : l \mid [\textsc{pop}] \tag{9}$$

$$\texttt{let}\ t\ \texttt{be}\ \langle \_, y\rangle\ \texttt{in}\ u \star l = t \star l \mid [\textsc{snd}, \textsc{push}] \mid u \star y : l \mid [\textsc{pop}] \tag{10}$$

$$\texttt{inl}(t) \star l = t \star l \mid [\textsc{inl}] \tag{11}$$

$$\texttt{inr}(t) \star l = t \star l \mid [\textsc{inr}] \tag{12}$$

$$\texttt{case}\ t\ \texttt{of}\ \texttt{inl}(x)\ \Rightarrow\ u \mid \texttt{inr}(y)\ \Rightarrow\ v \star l = t \star l \mid [\textsc{case}(c_1, c_2)] \tag{13}$$

where

$$\begin{aligned} c_1 &= u \star x : l \mid [\textsc{pop}, \textsc{ret}] \\ c_2 &= v \star y : l \mid [\textsc{pop}, \textsc{ret}] \end{aligned}$$

$$!t \star l = [\textsc{makeocl}(t \star l \mid [\textsc{ret}])] \tag{14}$$

$$\texttt{let}\ t\ \texttt{be}\ !x\ \texttt{in}\ u \star l = t \star l \mid [\textsc{read}, \textsc{push}] \mid u \star x : l \mid [\textsc{pop}] \tag{15}$$

$$\texttt{let}\ t\ \texttt{be}\ \_\ \texttt{in}\ u \star l = u \star l \tag{16}$$

$$\texttt{let}\ t\ \texttt{be}\ x@y\ \texttt{in}\ u \star l = t \star l \mid [\textsc{dup}, \textsc{push}, \textsc{push}] \mid u \star x : y : l \mid [\textsc{pop}, \textsc{pop}] \tag{17}$$

The *correctness* of the implementation with respect to the operational semantics can be stated as follows. Write $t\star$ for $t \star []$, t closed; also, if c is a code and v a value, write $c{\downarrow}v$ for

$$[], [], c, [] \longrightarrow^* [v], [], [], [].$$

Correctness of the implementation
For all typable programs t,

$$t{\Downarrow}c \implies \exists! v.(t\star{\downarrow}v \ \&\ c\star{\downarrow}v).$$

We will not attempt to prove correctness here. The main point is that we have an implementation-independent reference semantics with respect to which correctness can be formulated.

Yves Lafont has described an implementation of Intuitionistic Linear Logic in terms of a Linear Abstract Machine [Laf88], which is related to our machine in much the same way that the Categorical Abstract machine is related to the standard SECD machine (see e.g. [Hue90]). The reader is referred to [Laf88] for his interesting discussion of the implications of an implementation of this kind for storage allocation, in particular for the elimination of garbage collection.

However, our implementation is by no means committed to complete avoidance of sharing. Our DUP instruction, interpreted in the usual way on a list-structured memory, creates a copy of the *pointer* at the top of the stack, thus implementing the copying of the !-type by sharing. With a little additional work, we can ensure that the standard function type $A \Rightarrow B = !A \multimap B$ is implemented by the standard call-by-need technique [PJ87]. We shall briefly describe how this can be done. Firstly, we introduce a new instruction UPD, and a new form of value $\mathtt{ocv}(v)$, representing the "consolidation" of a value into a !-closure. The compilation of $!t$ is changed by replacing RET by UPD. The transition for READ is replaced by the following two transitions:

$$\begin{array}{lcl} l :: \mathtt{ocl}(c', e') : s, e, \mathrm{READ} : c, d & \longrightarrow & [], e', c', l : [s, e, c] : d \\ \mathtt{ocv}(v) : s, e, \mathrm{READ} : c, d & \longrightarrow & v : s, e, c, d \end{array}$$

Here the notation $l :: \mathtt{ocl}(c', e')$ means that l is the location of the cell representing the !-closure. The transition for UPD is:

$$\begin{array}{lcl} v : s, e, \mathrm{UPD} : c, l : [s', e', c'] : d & \longrightarrow & v : s', e', c', d \\ & & \text{where } l := \mathtt{ocv}(v) \end{array}$$

Thus in this transition the location of the !-closure is over-written by the consolidated value.

In this approach, just the values of the non-exponential types would be implemented without sharing. This seems to offer a better balance for sequential implementations than Lafont's approach, and is consistent with the idea of using Linear types to increase efficiency as described in the previous subsection. By contrast, the situation is rather different in a parallel implementation, where the avoidance of sharing has potentially much greater benefits. Our concurrent operational semantics for Classical Linear Logic, as presented in Section 6, does avoid sharing in a thorough-going fashion; parallel implementation is discussed in Section 8.

5 Basic Theory of the Linear Calculus

The systems we have considered so far have been very weak in expressive power, corresponding in logical terms to the Intuitionistic propositional calculus, and in programming terms to the simple typed λ-calculus, together with their Linear refinements. We shall now make a deceptively simple-looking extension, which in fact generates an enormous increase in expressive power. This is the addition of second-order propositional quantifiers, or in programming terms of (impredicative) quantification over types, enabling the definition of polymorphic functions. In the implicit typing approach we are using here, this extension looks particularly simple, since it does not appear at the term level at all, following the philosophy that types are compile-time constraints, and are not used in the actual computation. (This reflects current practice in languages such as ML [MTH90], Miranda [Tur85] and Haskell [HW88].)

The syntax of formulas is extended with propositional variables α, β, γ and the universal quantifier $\forall \alpha.A$. The system ILL (*i.e.* the sequent formulation of Linear Logic as given in Section 3) is extended to its second-order version ILL_2 by the rules

$$(\forall - R) \quad \frac{\Gamma \vdash A}{\Gamma \vdash \forall \alpha.A} \quad (*) \qquad (\forall - L) \quad \frac{\Gamma, A[B/\alpha] \vdash C}{\Gamma, \forall \alpha.A \vdash C}$$

where the right rule is subject to the side-condition $(*)$ that α does not appear free in Γ.

The linear term calculus is unchanged, and the term assignment extended to ILL_2 as follows:

$$(\forall - R)\quad \frac{\Gamma \vdash t : A}{\Gamma \vdash t : \forall\alpha.A}\quad (*) \qquad\qquad (\forall - L)\quad \frac{\Gamma, x : A[B/\alpha] \vdash t : C}{\Gamma, x : \forall\alpha.A \vdash t : C}$$

Of course, the operational semantics is left unchanged.

This system can be seen as a refinement of System F [GLT89], *i.e.* the second-order intuitionistic propositional calculus with its term assignment (in the implicit typing version), and we accordingly name it System LF ("Linear System F"). System F can be interpreted into LF by the obvious extension of the translation from Intuitionistic Logic into ILL we have already mentioned. The significance of this is that System F can represent all provably total recursive functions of second-order arithmetic [GLT89]. Moreover, inductive types such as lists and trees can be encoded in System F. For good surveys of programming in System F, see [GLT89, Rey85, Hue90].

Determinacy and Convergence

We now turn to the basic metatheory of System LF. The two major results for System F are undoubtedly that it satisfies the Church-Rosser and Strong Normalization properties [GLT89]. These results concern *reduction*, and apply to all strategies. The Church-Rosser property implies that all reduction strategies lead to the same result (normal form) when they terminate, while Strong Normalization says that all strategies do in fact terminate. What analogous properties can be formulated in terms of our style of operational semantics? Corresponding to the Church-Rosser property, we have:

Determinacy: For all closed t, $t \Downarrow c$ and $t \Downarrow d$ implies $c = d$; while corresponding to Strong Normalization, we have:

Convergence: For all closed t typable in System LF (*i.e.* for which

$$\vdash t : A$$

can be derived in LF for some type A), $t \Downarrow c$ for some c.

These two properties together say that evaluation, which *prima facie* is just a binary relation between programs and canonical forms, is in fact a *total function* on typable programs.

Since these properties are clearly weaker than CR and SN, why study them?

- Firstly, these results extend smoothly to situations where the stronger properties actually *fail*. For example, in Section 7 we will prove corresponding results for our computational interpretation of Classical Linear Logic, while the Church-Rosser property *fails* for the theory of reduction, as applied to sequent proofs or proof nets [Gir87]. Again, if we extend the calculus with general recursion, Strong Normalization will definitely be lost, while Convergence can be refined into Semantic Soundness [DM82] plus Computational Adequacy [MC88].

- As already explained, the evaluation relation reflects the intrinsic computational content of the Linear types, and so is the natural object of study.

- The proofs of Determinacy and Convergence are considerably simpler and less technical than the proofs of CR and SN.

Firstly, we have:

Theorem 5.1 *System LF satisfies Determinacy.*

Proof. By induction on the length of the inference that $t \Downarrow c$. If t has any form other than a `case`, at most one clause in the inductive definition of the evaluation relation is applicable to it. Otherwise, by the induction hypothesis at most one of the two rules corresponding to $(\oplus - L)$ is applicable. ∎

We now turn to Convergence. Our proof is a simplified and suitably modified version of Girard's original proof of SN for System F [Gir72]; see [Gal90] for a good exposition.

The idea is to use the evaluation relation to give a realizability semantics for types. We take a *semantic type* to be a set of closed linear terms, *i.e.* a subset of $\mathcal{T} = \mathcal{T}_{\varnothing}$. We interpret the Linear connectives over $\wp(\mathcal{T})$ as follows:

$$\begin{array}{rcl} \mathbf{1} & = & \{t \in \mathcal{T} \mid t \Downarrow * \} \\ U \otimes V & = & \{t \in \mathcal{T} \mid t \Downarrow c \otimes d \ \& \ c \in U, d \in V\} \\ U \multimap V & = & \{t \in \mathcal{T} \mid t \Downarrow \lambda x.v \ \& \ \forall u \in U.\, (tu \in V)\} \\ U \mathbin{\&} V & = & \{t \in \mathcal{T} \mid t \Downarrow \langle u, v\rangle \ \& \ u \in U, v \in V\} \\ U \oplus V & = & \{t \in \mathcal{T} \mid (t \Downarrow \mathtt{inl}(c) \ \& \ c \in U) \text{ or } (t \Downarrow \mathtt{inr}(d) \ \& \ d \in V)\} \\ !U & = & \{t \in \mathcal{T} \mid t \Downarrow !u \ \& \ u \in U\} \end{array}$$

while for $F : \wp(\mathcal{T}) \to \wp(\mathcal{T})$, we define

$$\forall(F) = \bigcap \{F(U) \mid U \in \wp(\mathcal{T})\}.$$

These definitions induce a semantic function

$$[\![\cdot]\!] : \mathsf{TExp} \to \mathsf{TEnv} \to \wp(\mathcal{T})$$

where TExp is the set of linear type expressions, and $\mathsf{TEnv} = \mathsf{TVar} \to \wp(\mathcal{T})$ is the set of type environments mapping type (or propositional) variables to semantic types.

We can now give a realizability interpretation for sequents:

$$\bar{x} : \Gamma \models t : A \iff \forall \eta \in \mathsf{TEnv}, \bar{u} \in [\![\Gamma]\!]\eta.\, (t[\bar{u}/\bar{x}] \in [\![A]\!]\eta),$$

and state the basic result:

Theorem 5.2 (Realizability) *If $\Gamma \vdash t : A$ is derivable in System LF, then $\Gamma \models t : A$.*

Proof. By induction on the derivation of $\Gamma \vdash t : A$ in LF. Three cases will illustrate the argument quite sufficiently.

(1).

$$(\multimap - L) \quad \frac{\Gamma \vdash t : A \quad x : A, \Delta \vdash u : C}{\Gamma, f : A \multimap B, \Delta \vdash u[(ft)/x] : C}$$

Let η, $\bar{u}$, $\bar{v}$, w be given with $\bar{u} \in [\![\Gamma]\!]\eta$, $\bar{v} \in [\![\Delta]\!]\eta$, $w \in [\![A \multimap B]\!]\eta$. By induction hypothesis $\bar{x} : \Gamma \models t : A$, and so $t[\bar{u}/\bar{x}] \in [\![A]\!]\eta$; together with $w \in [\![A \multimap B]\!]\eta$, this implies $wt[\bar{u}/\bar{x}] \in [\![B]\!]\eta$. By induction hypothesis again, $x : A, \bar{y} : \Delta \models u : C$, and so $u[wt[\bar{u}/\bar{x}]/x, \bar{v}/\bar{y}] \in [\![C]\!]\eta$. But (using linearity)

$$u[wt[\bar{u}/\bar{x}]/x, \bar{v}/\bar{y}] = u[(ft)/x][\bar{u}/\bar{x}, w/f, \bar{v}/\bar{y}].$$

So $\Gamma, f : A \multimap B, \Delta \models u[(ft)/x] : C$, as required.

(2).

$$(\text{Contraction}) \quad \frac{\Gamma, x : !A, y : !A \vdash t : B}{\Gamma, z : !A \vdash \texttt{let}\ z\ \texttt{be}\ x@y\ \texttt{in}\ t : B}$$

Let η, $\bar{u}$, u be given with $\bar{u}, u \in [\![\Gamma, !A]\!]\eta$. By induction hypothesis, $\Gamma, x : !A, y : !A \models t : B$, so $t[\bar{u}, u/x, u/y]\Downarrow c \in [\![B]\!]\eta$. Applying the definition of the evaluation relation, we can conclude that $\texttt{let}\ u\ \texttt{be}\ x@y\ \texttt{in}\ t[\bar{u}]\Downarrow c \in [\![B]\!]\eta$, as required.

(3).

$$(\forall - R) \quad \frac{\Gamma \vdash t : A}{\Gamma \vdash t : \forall\alpha.A}$$

By induction hypothesis, $\Gamma \models t : A$, *i.e.*

$$\forall\eta.(\forall\bar{u} \in [\![\Gamma]\!]\eta.\, t[\bar{u}] \in [\![A]\!]\eta).$$

Because α does not occur free in Γ, this is equivalent to

$$\forall\eta.(\forall\bar{u} \in [\![\Gamma]\!]\eta.\, \forall U.\, t[\bar{u}] \in [\![A]\!]\eta[\alpha \mapsto U]),$$

i.e. to $\Gamma \models t : \forall\alpha.A$. ∎

As a simple consequence of the Realizability Theorem:

Theorem 5.3 *System LF satisfies Convergence.*

PROOF. Suppose that $\vdash t : A$ is derivable in System LF. We can apply $(\forall - R)$ freely here, so without loss of generality we can assume that A is closed. By the Realizability Theorem, $\models t : A$, *i.e.* $t \in [\![A]\!]$. Write A as $\forall\bar{\alpha}.\,B$, where B does not have a quantifier outermost. If $B = \alpha_i$, then $[\![A]\!] = \varnothing$, contradicting $t \in [\![A]\!]$. In any other case, the realizability semantics of the outermost connective in B immediately implies that $t\Downarrow$. ∎

6 Classical Linear Logic

Intuitionistic Linear Logic is essentially a refinement of ordinary Intuitionistic Logic, and its computational interpretation is a refinement of the λ-calculus. The full system of Linear Logic, which for emphasis we refer to as Classical Linear Logic (CLL), represents a much more radical departure from the tradition of constructive logic, and its computational interpretation requires a corresponding departure from the functional framework.

The basic step in the extension from Intuitionistic to Classical Linear logic is the introduction of the *Linear negation* $A^{\perp}$. The idea is that this will obey the same kind of laws as classical negation, while constructive content is retained through linearity. This requires the introduction of a number of new connectives, as duals to the existing ones: $\perp$ as dual to $\mathbf{1}$, $\parr$ ("par") as dual to $\otimes$, ? ("why not") as dual to !, and $\exists$ as dual to $\forall$. (The two additive constructs $\&$ and $\oplus$ will be dual to each other in CLL.) Linear negation is then characterized by the following laws:

$$
\begin{aligned}
A^{\perp\perp} &= A \\
\mathbf{1}^{\perp} &= \perp \\
(A \otimes B)^{\perp} &= A^{\perp} \invamp B^{\perp} \\
(A \mathbin{\&} B)^{\perp} &= A^{\perp} \oplus B^{\perp} \\
(!A)^{\perp} &= ?A^{\perp} \\
(\forall \alpha.\, A)^{\perp} &= \exists \alpha.\, A^{\perp} \\
A \multimap B &= A^{\perp} \invamp B.
\end{aligned}
\tag{18}
$$

The syntax of Linear formulas in CLL is then defined as follows. Formulas are built from propositional variables α, β, γ and their Linear negations $\alpha^{\perp}$, $\beta^{\perp}$, $\gamma^{\perp}$ by the following connectives and quantifiers:

Units	$\mathbf{1}$	$\perp$
Multiplicatives	$\otimes$	$\invamp$
Additives	$\&$	$\oplus$
Exponentials	$!$	$?$
Quantifiers	$\forall$	$\exists$.

Linear negation is *definitionally extended* to general formulas by the equations (18), while Linear implication is treated as a derived operator, defined by the last equation in (18).

The proof system for CLL is a fully symmetric sequent calculus, in which sequents have the form

$$\Gamma \vdash \Delta,$$

with the intended meaning that the formula $\bigotimes \Gamma \multimap \invamp \Delta$ is valid. However, a considerable economy is made by observing that a sequent $\Gamma \vdash \Delta$ is equivalent by (18) to the sequent $\vdash \Gamma^{\perp}, \Delta$; so it is sufficient to consider right-sided sequents only. The sequent calculus presentation of CLL can then be given as follows.

$$
\text{Axiom}\ \frac{}{\vdash A^{\perp}, A} \qquad \text{Exchange}\ \frac{\vdash \Gamma, A, B, \Delta}{\vdash \Gamma, B, A, \Delta} \qquad \text{Cut}\frac{\vdash \Gamma, A \quad \vdash \Delta, A^{\perp}}{\vdash \Gamma, \Delta}
$$

$$
\text{Unit}\ \frac{}{\vdash \mathbf{1}} \qquad \text{Perp}\ \frac{\vdash \Gamma}{\vdash \Gamma, \perp}
$$

$$
\text{Times}\ \frac{\vdash \Gamma, A \quad \vdash \Delta, B}{\vdash \Gamma, \Delta, A \otimes B} \qquad \text{Par}\ \frac{\vdash \Gamma, A, B}{\vdash \Gamma, A \invamp B}
$$

$$
\text{With}\ \frac{\vdash \Gamma, A \quad \vdash \Gamma, B}{\vdash \Gamma, A \mathbin{\&} B} \qquad \text{Plus (i)}\ \frac{\vdash \Gamma, A}{\vdash \Gamma, A \oplus B} \qquad \text{Plus (ii)}\ \frac{\vdash \Gamma, B}{\vdash \Gamma, A \oplus B}
$$

$$\text{Dereliction} \quad \frac{\vdash \Gamma, A}{\vdash \Gamma,\, ?A} \qquad \text{Of Course} \quad \frac{\vdash ?\Gamma, A}{\vdash ?\Gamma,\, !A}$$

$$\text{Weakening} \quad \frac{\vdash \Gamma}{\vdash \Gamma,\, ?A} \qquad \text{Contraction} \quad \frac{\vdash \Gamma,\, ?A,\, ?A}{\vdash \Gamma,\, ?A}$$

$$\text{All} \quad \frac{\vdash \Gamma, A}{\vdash \Gamma, \forall\alpha.A} \quad (*) \qquad \text{Exists} \quad \frac{\vdash \Gamma, A[B/\alpha]}{\vdash \Gamma, \exists\alpha.A}$$

Note that these rules can be obtained from the rules of ILL by using the equations (18) and shifting the premises to the right of the turnstile. For example, both $(\& - R)$ and $(\oplus - L)$ translate into instances of the With rule, while $(\multimap - R)$ translates into the Par rule, and $(\multimap - L)$ into the Times rule. In particular, this shows that ILL can be interpreted as a subsystem of CLL; and hence that Intuitionistic logic can be interpreted in CLL.

The question now arises, how to give a computational interpretation for CLL, which is not merely an extension of ILL, but embodies a radical change of perspective: the asymmetry between inputs and outputs has been abolished, apparently by formal fiat. But what does this actually *mean* in computational terms?

To sharpen our ideas, let us focus on the Cut rule. In ILL, this appears as:

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

while in CLL one has:

$$\frac{\vdash \Gamma, A \quad \vdash \Delta, A^{\perp}}{\vdash \Gamma, \Delta}$$

There is an important formal difference between these two versions of the Cut rule. The Intuitionistic version is asymmetric; the left premise is distinguished from the right by the fact that the cut formula appears in the output position (*i.e.* as a conclusion) in one, and in input position (*i.e.* as a premise) in the other. This is reflected on the programming level by the fact that Cut is interpreted in System LF by the non-commutative operation of *function composition* (expressed syntactically as substitution).

By contrast, the Cut rule in CLL is fully symmetric; in CLL we have the *equality* $A^{\perp\perp} = A$, so we could equally well write

$$\frac{\vdash \Delta, A^{\perp} \quad \vdash \Gamma, A}{\vdash \Delta, \Gamma}$$

(By the Exchange rule, the different order of the formulas in the resulting sequent is not significant.) So on the programming level, we should expect to interpret the CLL Cut rule by a *commutative* operation. At this point it becomes very natural to invoke concurrency theory, as offering just the kind of generalization we need. As Milner has emphasized throughout his work on concurrency [Mil80, Mil90], communicating processes can be thought of as a generalization of functions, while the key operation of *parallel composition* is a commutative operation which generalizes function application or composition. So we may attempt to replace expression evaluation by concurrent process execution as the underlying computational paradigm, and to interpret Cut by a suitable form of parallel composition. Of course, these ideas are entirely in line with Girard's emphatic hints that Classical Linear Logic opens the way to a logical view of parallel computation [Gir87, Gir89b].

We shall now present a computational interpretation of CLL which seeks to embody these ideas in a simple and elegant form, in the general framework we have established in the preceding sections. Thus we will define a syntax for *proof expressions*, and give an assigment of proof expressions to sequent proofs in CLL, and an operational semantics for proof expressions.

Syntax of Proof Expressions

Firstly, a point of terminology: we shall use *list* to mean finite sequence. We define a number of syntactic categories:

- A set $\mathcal{N}$ of *names*, ranged over by x, y, z. We use $\bar{x}$, $\bar{y}$, $\bar{z}$ to range over lists of names.
- *Terms* have one of the forms

$$\begin{array}{cccc} x & & & \\ * & \circledast & & \\ t \otimes u & t \parr u & & \\ \mathtt{inl}(t) & \mathtt{inr}(u) & \bar{x}(P \mathbin{[\!]} Q) & \\ ?t & \_ & t@u & \bar{x}(P) \end{array}$$

  where t, u are terms, and P, Q are proof expressions.

- *Coequations* have the form $t \perp u$, where t, u are terms. We use Θ, Ξ to range over lists of coequations.
- *Proof expressions* have the form $\Theta; \bar{t}$, where Θ is a list of coequations, and $\bar{t}$ is a list of terms. We use P, Q to range over proof expressions.

Notation. The occurrences of $x_1, \ldots, x_k$ in a term of the form $x_1, \ldots, x_k(P \mathbin{[\!]} Q)$ or $x_1, \ldots, x_k(P)$ are said to be *passive*; all other occurrences are *active*. If e is some syntactic expression (term, coequation, proof expression etc.) we write $\mathcal{N}(e)$ for the set of names occurring in e, and $\mathrm{A}\mathcal{N}(e)$ $(\mathrm{P}\mathcal{N}(e))$ for the set of names occurring actively (passively) in e.

We shall now define an assignment of proof expressions to sequent proofs in CLL. The idea is that, to each proof Π of a sequent $\vdash A_1, \ldots, A_k$, we will assign a proof expression $\Theta; t_1, \ldots, t_k$, where Θ corresponds to the uses of the Cut rule in Π.

To ensure that suitable linearity constraints are satisfied, we shall adopt the following Name Convention (cf. the Variable Convention in [Bar84]): *different* names are introduced for each instance of the Axiom, With and Of Course rules.

Proof Expression Assignment for CLL

$$\text{Axiom} \quad \frac{}{\vdash ; x : A^{\perp}, x : A} \qquad \text{Exchange} \quad \frac{\vdash \Theta; \Gamma, t : A, u : B, \Delta}{\vdash \Theta; \Gamma, u : B, t : A, \Delta}$$

$$\text{Cut} \quad \frac{\vdash \Theta; \Gamma, t : A \qquad \vdash \Xi; \Delta, u : A^{\perp}}{\vdash \Theta, \Xi, t \perp u; \Gamma, \Delta}$$

$$\text{Unit} \quad \frac{}{\vdash ; * : \mathbf{1}} \qquad \text{Perp} \quad \frac{\vdash \Theta; \Gamma}{\vdash \Theta; \Gamma, \circledast : \bot}$$

$$\text{Times} \quad \frac{\vdash \Theta; \Gamma, t : A \quad \vdash \Xi; \Delta, u : B}{\vdash \Theta, \Xi; \Gamma, \Delta, t \otimes u : A \otimes B} \qquad \text{Par} \quad \frac{\vdash \Theta; \Gamma, t : A, u : B}{\vdash \Theta; \Gamma, t \,⅋\, u : A \,⅋\, B}$$

$$\text{With} \quad \frac{\vdash \Theta; \bar{t} : \Gamma, t : A \quad \vdash \Xi; \bar{u} : \Gamma, u : B}{\vdash ; \bar{x} : \Gamma, \bar{x}(\Theta; \bar{t}, t \mathbin{[\!]} \Xi; \bar{u}, u) : A \,\&\, B}$$

$$\text{Plus (i)} \quad \frac{\vdash \Theta; \Gamma, t : A}{\vdash \Theta; \Gamma, \mathtt{inl}(t) : A \oplus B} \qquad \text{Plus (ii)} \quad \frac{\vdash \Theta; \Gamma, u : B}{\vdash \Theta; \Gamma, \mathtt{inr}(u) : A \oplus B}$$

$$\text{Dereliction} \quad \frac{\vdash \Theta; \Gamma, t : A}{\vdash \Theta; \Gamma, ?t : ?A} \qquad \text{Weakening} \quad \frac{\vdash \Theta; \Gamma}{\vdash \Theta; \Gamma, \_ : ?A}$$

$$\text{Contraction} \quad \frac{\vdash \Theta; \Gamma, t : ?A, u : ?A}{\vdash \Theta; \Gamma, t@u : ?A}$$

$$\text{Of Course} \quad \frac{\vdash \Theta; \bar{t} : ?\Gamma, t : A}{\vdash ; \bar{x} : ?\Gamma, \bar{x}(\Theta; \bar{t}, t) : !A}$$

$$\text{All} \quad \frac{\vdash \Theta; \Gamma, t : A}{\vdash \Theta; \Gamma, t : \forall \alpha . A} \quad (*) \qquad \text{Exists} \quad \frac{\vdash \Theta; \Gamma, t : A[B/\alpha]}{\vdash \Theta; \Gamma, t : \exists \alpha . A}$$

Operational Semantics: the Linear CHAM

We now complete our computational interpretation of Classical Linear Logic by giving an operational semantics for proof expressions. Rather than directly defining the relation of evaluation to canonical form, we shall define a one-step transition relation on proof expressions, and define canonical forms as certain *normal forms* with respect to this relation. This is because the notion of computation for proof expressions is inherently *parallel*; the model is that the coequations form a pool of concurrent processes. In fact, our presentation of the operational semantics fits very nicely into the framework of the Chemical Abstract Machine proposed recently by Berry and Boudol [BB90] as a paradigm for concurrent abstract machines. They describe the basic ideas thus:

> Most available concurrency models are based on architectural concepts, e.g. networks of processes communicating by means of ports or channels. Such concepts convey a rigid geometrical vision of concurrency. Our chemical abstract machine model is based on a radically different paradigm ... where the concurrent components are freely "moving" in the system and communicate when they come into contact. ...
>
> Intuitively, the state of a system is like a *chemical solution* in which floating *molecules* can interact with each other according to *reaction rules*; a *magical mechanism* stirs the solution, allowing for possible contacts between molecules—in chemistry, this is the result of Brownian motion, but we don't insist on any particular mechanism, this being an implementation matter. The solution transformation process is obviously truly parallel; any number of reactions can be performed in parallel, provided that they involve disjoint sets of molecules.

The "molecules" of the Linear CHAM are the coequations. We refer to Θ in $\Theta;\bar{t}$ as the "solution", and to $\bar{t}$ as the "main body". The idea is that the computation is done in the solution, with the result recorded in the main body. One can think of each coequation either as a single sequential process, or as a tightly-coupled synchronous parallel composition of two processes, proceeding in lockstep. (So coequations could be modelled by "membranes" in Berry and Boudol's terminology; but we shall not pursue this idea.)

We distinguish between two kinds of rule for the CHAM (cf. [Mil90]): *structural rules*, which describe the "magical mixing" of the solution; and *reaction rules*, which describe the actual computation steps.

Structural Rules

There are two basic structural rules:

- $t \perp u \rightleftharpoons u \perp t$
- $t \perp u, t' \perp u' \rightleftharpoons t' \perp u', t \perp u$

The first says that each coequation can be regarded as a multiset of exactly two terms, the second that lists of coequations can be regarded as multisets.

These rules can be applied in any context:

$$\frac{\Theta \rightleftharpoons \Xi}{C[\Theta] \rightleftharpoons C[\Xi]}$$

The basic metarule for the CHAM refers to the transition relation $\longrightarrow$ to be defined below.

Magical Mixing Rule:

$$\frac{P \rightleftharpoons^* P' \quad P' \longrightarrow Q' \quad Q' \rightleftharpoons^* Q}{P \longrightarrow Q}$$

We regard this as a metarule, since it is really part of the specification of the machine, rather than a description of an actual computation step.

Notational Interlude: Variants

We shall need to consider *variants* of terms t occurring in a proof expression P, *i.e.* copies of t in which all names have been replaced by "fresh" names not already occurring in P. In order to implement this global condition in a local way, we need a little extra structure. We fix a bijection $\mathcal{N} \cong N_0 \times \{l, r\}^*$, and extend the Name Convention so that when a name $x \leftrightarrow \langle x_0, s\rangle$ is introduced in a proof expression, the x_0 component is distinct from that of any name already occurring in the expression. Now given a term t, we define t^l, t^r to be the result of replacing each occurrence of a name $x \leftrightarrow \langle x_0, s\rangle$ in t by $y \leftrightarrow \langle x_0, sl\rangle$, $z \leftrightarrow \langle x_0, sr\rangle$ respectively. The idea is that the following invariant is established by the proof expression assignment and maintained by the transition relation to be defined below:

For all distinct names $x \leftrightarrow \langle x_0, s\rangle$, $y \leftrightarrow \langle y_0, t\rangle$ occurring in P, $x_0 = y_0$ implies that s is incompatible with t (*i.e.* they have no upper bound in the prefix ordering).

Reaction Rules

These rules describe how lists of adjacent coequations react, giving rise to new lists.

Notation: given $\bar{x} = x_1, \ldots, x_k$, $\bar{t} = t_1, \ldots, t_k$, we write $\bar{x} \perp \bar{t}$ to denote the list $x_1 \perp t_1, \ldots, x_k \perp t_k$.

Communication:

$$t \perp x, x \perp u \longrightarrow t \perp u$$

Unit:

$$* \perp \circledast \longrightarrow$$

Pair:

$$t \otimes u \perp t' \mathbin{⅋} u' \longrightarrow t \perp t', u \perp u'$$

Case Left:

$$\bar{x}(\Theta; \bar{t}, t \mathbin{[\!]} \Xi; \bar{u}, u) \perp \mathtt{inl}(v) \longrightarrow \Theta, \bar{x} \perp \bar{t}, t \perp v$$

Case Right:

$$\bar{x}(\Theta; \bar{t}, t \mathbin{[\!]} \Xi; \bar{u}, u) \perp \mathtt{inr}(v) \longrightarrow \Xi, \bar{x} \perp \bar{u}, u \perp v$$

Read:

$$\bar{x}(\Theta; \bar{t}, t) \perp ?u \longrightarrow \Theta, \bar{x} \perp \bar{t}, t \perp u$$

Discard:

$$\bar{x}(P) \perp \_ \longrightarrow x_1 \perp \_, \ldots, x_k \perp \_$$

Copy:

$$\bar{x}(P) \perp u@v \longrightarrow \bar{x} \perp (\bar{x}^l @ \bar{x}^r), \bar{x}(P)^l \perp u, \bar{x}(P)^r \perp v$$

The reaction rules contribute to the global transition relation on proof expressions via the following metarule:

Reaction Context Rule:

$$\frac{\Theta \longrightarrow \Xi}{\Theta_1, \Theta, \Theta_2; \bar{t} \longrightarrow \Theta_1, \Xi, \Theta_2; \bar{t}}$$

Cleanup Rule

Finally we have a rule which tidies up a computation by consolidating information back into the main body $\bar{t}$ of a proof expression $\Theta; \bar{t}$. This is somewhat analogous to collecting the answer substitution from a PROLOG computation.

Cleanup:

$$x \perp t, \Theta; \bar{t} \longrightarrow \Theta; \bar{t}[t/x] \quad (x \in \mathrm{A}\mathcal{N}(\bar{t}))$$

We can now define the *result* of a computation. A proof expression $P = \Theta; \bar{t}$ is *canonical* if it is a $\longrightarrow$-normal form, and each coequation in Θ has the form $x \perp t$ or $t \perp x$ for some name x. P is *cut-free* if Θ is empty.

Definition 6.1 *We define $P \Downarrow Q$—P evaluates to canonical form Q—by:*

$$P \Downarrow Q \stackrel{\text{def}}{\Longleftrightarrow} P \longrightarrow^* Q, \quad Q \textit{ canonical.}$$

Discussion

Firstly, we consider the computational intuitions behind these rules. The key rule is Communication, which is the only one which involves interaction between coequations. In ILL, as in λ-calculus, variables are place-holders for substitution. In CLL, the two occurrences of a name can be thought of as the two ends of a channel; the Communication rule uses this channel to connect two processes (terms) together. Linearity amounts to the restriction that channels are used only once.

From the computational aspect, the most interesting rules are those for the additives and exponentials. In both cases we have lazy types—$\&$ and !—which in the concurrent framework must be implemented by some form of explicit *synchronization*. This is the rôle of the forms $\bar{x}(P \,[\!]\, Q)$ and $\bar{x}(P)$. In both cases, proof expressions are suspended from execution, and only resumed when sufficient information is available (or, in more computational terms, when sufficient demand has been generated). In the case of the additives, the With rule (which under the classical dualities is equivalent to the Intuitionistic rule $(\oplus - L)$) corresponds to a `case` statement, *i.e.* a choice between two alternatives. Clearly, we only want to evaluate that expression corresponding to the alternative actually chosen; so we must *wait* until the choice is made. This is done when the term $\bar{x}(P \,[\!]\, Q)$ is cut against a term denoting a proof of the dual $(A \& B)^{\perp} = A^{\perp} \oplus B^{\perp}$, of the form $\texttt{inl}(t)$, where t is a proof of $A^{\perp}$, or $\texttt{inr}(u)$, where u is a proof of $B^{\perp}$. Hence the Case Left and Case Right rules. So we must defer any evaluation of the proofs of the side formulas Γ of the With rule until this choice is made. (Indeed, we don't even know till then whether these proofs should be taken as $\bar{t}$ or $\bar{u}$.) This is accomplished by replacing the proof terms by the names $\bar{x}$. These can in turn be embedded in complex proof terms and cut against other terms. However, when one of these names "rises to the surface" in a coequation $x_i \perp w$, the computation with that coequation will not be able to proceed until the choice associated with the With rule which generated the name x_i is resolved, by the application of a Case Left or Case Right rule. Suppose the Case Left rule is applied. At that point, the coequation $x_i \perp t_i$ is released into the solution, and by the Communication rule, this can "bond" with $x_i \perp w$ to form the coequation $t_i \perp w$, which can now proceed.

Similar considerations apply to the rules for the exponentials. The idea here is that the term of type $?A^{\perp}$ specifies how many copies of the term of type $!A$ are required; each of the terms for the side formulas $?\Gamma$ of the Of Course rule which generated the $!A$ term $\bar{x}(P)$ must then be directed to ask for a corresponding multiple of copies from its "input".

From the logical side, the reaction rules correspond *exactly* to the key steps in Cut Elimination, or more precisely of evaluation to canonical form. This is spelled out in detail in the proof of the Realizability Theorem 7.17, and the reader is strongly encouraged to work out some of the transitions described there in detail. So each rule has a clear logical content.

Finally, some brief remarks about the relationship between our proof-expressions and Girard's proof nets [Gir87]. (A detailed comparison must be left to future work.) Roughly speaking, proof expressions correspond to proof nets, the lazy forms $\bar{x}(P \,[\!]\, Q)$ and $\bar{x}(P)$ to proof boxes, and the reaction rules to the symmetric contractions, as described in [Gir87]. A more precise comparison would require some care; for example, the use of channels for both axiom contraction and the synchronization associated with the lazy types in our calculus does not appear in the proof net formalism. The author's impression is that both representations have their merits and uses:

- Proof nets are visually appealing and support geometric insights into the structure of proofs. They work very well for the multiplicative fragment, but the use of boxes is cumbersome and negates many of their advantages.

- Proof expressions are an efficient syntactic vehicle for making precise definitions and carrying out detailed proofs; and also as a linear notation for writing down Linear proofs!

The reader must be left to form his or her own opinion of the relative merits of proof expressions *vs.* proof nets as a syntactic medium, and, more importantly, whether we have succeeded in making the computational reading of Linear Logic, and particularly the connections with concurrent computation, more substantial and convincing. (A full evaluation should include the material to be presented in Sections 7 and 8.) We will briefly indicate a significant difference in the present approach as compared to Girard's, that should not be overlooked. This is that, in keeping with the general philosophy on operational semantics set out in Section 2, our operational semantics is based on evaluation to *canonical form* rather than *normal form*. We feel that this choice is amply justified by the general arguments given in Section 2, the evidence of our definitions in this Section, and the detailed results in Section 7. To recapitulate:

- Our operational definitions are much more compact, elegant, and memorable than the calculus presented in [Gir87].
- They correspond much better to what would actually be done in an implementation. (See Section 8.)
- There are considerable technical benefits, the main one being that Determinacy is preserved. See Theorem 7.9, and Lemma 7.10. At the same time, not too much is lost: see Theorem 7.20. (Also, our realizability interpretation of the Linear types is essentially the same as Girard's, yet our computation rules suffice to prove the Realizability Theorem 7.17.)

7 Basic Theory of PE_2

We name the formal system of second-order propositional CLL with its assignment of proof expressions PE_2, by analogy with Girard's PN2. We shall now study the basic properties of this system.

Notation. We write $\mathsf{PE}_2 \vdash \Theta; \bar{t} : \Gamma$ if the sequent $\vdash \Theta; \bar{t} : \Gamma$ is derivable in PE_2.

Firstly, we consider two important structural conditions on proof expressions.

1. **Linearity**. A proof expression P is *linear* if:
   - Each name occurring in P does so exactly *twice*.
   - For each term $\bar{x}(Q)$ or $\bar{x}(Q \,[\!]\, R)$ occurring in P, the proof expressions Q, R are linear.
2. **Acyclicity**. Given a proof expression P, we define a graph $\mathcal{G}(P)$ with two types of arc as follows:
   - The nodes of $\mathcal{G}(P)$ are the set of all occurrences of terms t in P. (We will blur the distinction between terms and their occurrences in our notation, but the reader should be aware of it.)

- There is an arc $t \smile u$ iff one of the following conditions holds:
  - $t \perp u$ or $u \perp t$ occurs in P.
  - $t \otimes u$ or $u \otimes t$ occurs in P.
  - For some v, w, x, y, ($t \perp v$ or $v \perp t$) and ($u \perp w$ or $w \perp u$) occurs in P, $x \in \mathcal{N}(v)$, $y \in \mathcal{N}(w)$, and x, y occur in $\bar{x}$ for some $\bar{x}(Q)$ occurring in P.
- There is an arc $t \frown u$ iff t, u are disjoint occurrences, and $\mathcal{N}(t) \cap \mathcal{N}(u) \neq \varnothing$.

A *cycle* in $\mathcal{G}(P)$ is a sequence

$$t_1 \smile u_1 \frown \cdots \frown t_k \smile u_k \frown t_1 \quad (k \geq 1),$$

in which no occurrence is repeated other than t_1.

A proof expression P is *acyclic* if $\mathcal{G}(P)$ has no cycles.

Acyclicity can be understood as the appropriate hereditary condition to ensure that self-loops $x \perp x$ can never appear in a typable proof expression as we apply transitions. We can read $t \smile u$ as "t should be disjoint from u", and $t \frown u$ as "t is connected to u". Then acyclicity precludes the "contradictory" situation in which nodes that should be disjoint are connected.

Proposition 7.1 *If* $\mathsf{PE}_2 \vdash \Theta; \bar{t} : \Gamma$*, then* $\Theta; \bar{t}$ *is linear and acyclic.*

PROOF. By induction on derivations in PE_2. Linearity is immediate from the form of the rules and the Name Convention. For acyclicity, we consider the case for the Cut rule:

$$\frac{\vdash \Theta; \bar{t} : \Gamma, t : A \qquad \vdash \Xi; \bar{u} : \Delta, u : A^{\perp}}{\vdash \Theta, \Xi, t \perp u; \bar{t} : \Gamma, \bar{u} : \Delta}$$

By the Name Convention, $\mathcal{N}(\Theta; \bar{t}, t) \cap \mathcal{N}(\Xi; \bar{u}, u) = \varnothing$, so there is no $\frown$-link between these two proof expressions. Hence any cycle in $\mathcal{G}(\Theta, \Xi, t \perp u; \bar{t}, \bar{u})$ must in fact lie either in $\mathcal{G}(\Theta; \bar{t}, t)$ or in $\mathcal{G}(\Xi; \bar{u}, u)$, contra hypothesis. ∎

Proposition 7.2 *Suppose* $P \longrightarrow Q$*. If* P *is linear and acyclic, so is* Q*.*

PROOF. Firstly we show that linearity is preserved, by cases on the rule used to derive $P \longrightarrow Q$.

Unit, Pair: trivial.

Communication: $t \perp x, x \perp u \longrightarrow t \perp u$. The net effect is to delete both occurrences of x.

Cleanup: $t \perp x, \Theta; \bar{t} \longrightarrow \Theta; \bar{t}[t/x]$, where $x \in \mathrm{A}\mathcal{N}(\bar{t})$. Since P is linear, the net effect is to delete both occurrences of x.

Case Left: $\bar{x}(R \mathbin{[\!]} S) \perp \mathtt{inl}(v) \longrightarrow \Theta, \bar{x} \perp \bar{t}, t \perp v$, where $R = \Theta; \bar{t}, t$. By definition of linearity, P linear implies R, S linear. Thus every name occurring in S had both its occurrences there; so the net effect of this transition is to delete all occurrences of $\mathcal{N}(S)$.

Case Right: symmetrical to Case Left.

Read: trivial, since this transition has no effect on the number of name occurrences.

Discard: $\bar{x}(R) \perp \_ \longrightarrow \bar{x} \perp \_, \ldots, \_$. Similarly to Case Left, since R is linear the effect is to delete all occurrences of $\mathcal{N}(R)$.

Copy: $\bar{x}(R) \perp t@u \longrightarrow \bar{x} \perp (\bar{x}^l@\bar{x}^r), \bar{x}(R)^l \perp t, \bar{x}(R)^r \perp u$. Since R was linear, and $(\cdot)^l$, $(\cdot)^r$ rename outside P, the net effect is to delete $\mathcal{N}(R)$, and to add two copies of x^l, x^r for each $x \in \mathcal{N}(R) \cup \bar{x}$. Furthermore, R^l, R^r will be linear, since R was.

Now we show that acylicity is preserved; this will require the assumption that P is linear. The general technique is to show that any cycle in Q can be transformed into one in P, contradicting the assumption that P is acyclic. We argue again by cases on how $P \longrightarrow Q$ was derived.

Cleanup: $t \perp x, \Theta; \bar{t} \longrightarrow \Theta; \bar{t}[t/x]$. Any path

$$\cdots \frown t \smile \cdots$$

in Q can be transformed into

$$\cdots \frown t \smile x \frown x \smile \cdots$$

in P.

Communication: $t \perp x, x \perp u \longrightarrow t \perp u$. A cycle in Q can be transformed into one in P by replacing any sub-path

$$\cdots \frown t \smile u \frown \cdots$$

by

$$\cdots \frown t \smile x \frown x \smile u \frown \cdots.$$

Unit: trivial.

Pair: $t \otimes u \perp t' \mathbin{⅋} u' \longrightarrow t \perp t', u \perp u'$. Consider e.g. a link

$$\cdots \frown t \smile t' \frown \cdots$$

in a putative cycle in Q. If $u \frown t$ or $t' \frown u'$ then we can replace this link by

$$\cdots \frown t \smile u \frown \cdots.$$

In any other case, we can replace it by

$$\cdots \frown t \otimes u \smile t' \mathbin{⅋} u' \frown \cdots.$$

Case Left: $\bar{x}(R \mathbin{[\!]} S) \perp \mathtt{inl}(v) \longrightarrow \Theta, \bar{x} \perp \bar{t}, t \perp v$, where $R = \Theta; \bar{t}, t$. Since R is linear by assumption, $\mathcal{N}(R)$ is disjoint from the remainder of P. So there can be no cycle containing links $x_i \smile t_i$ or $t \smile v$; and any cycle containing a link in R must lie entirely in R. But then any cycle in Q must have already occurred in P.

Case Right, Read: similar to Case Left.

Discard: trivial.

Copy: $\bar{x}(R) \perp t@u \longrightarrow \bar{x} \perp (\bar{x}^l@\bar{x}^r), \bar{x}(R)^l \perp t, \bar{x}(R)^r \perp u$. Arguing as in the proof of linearity, we know that R^l is linear, and $\mathcal{N}(R^l)$ is disjoint from the rest of Q. Hence any cycle in Q containing a node in R^l must lie wholly within R^l; but this would imply a cycle in R and hence in P, contra hypothesis. Similarly for R^r. Thus if we had a cycle in Q, by replacing any sub-paths

$$\cdots \frown x_i \smile (x_i^l@x_i^r) \frown \bar{x}(R)^l \smile t \frown \cdots$$

or

$$\cdots \frown x_i \smile (x_i^l @ x_i^r) \frown \bar{x}(R)^r \smile u \frown \cdots$$

by

$$\cdots \frown \bar{x}(R) \smile t @ u \frown \cdots$$

we would obtain a cycle in P, contra hypothesis. The only case in which this cannot be done, because $t \frown u$, yields

$$\cdots \frown v_i \smile w_i \frown x_i \smile (x_i^l @ x_i^r) \frown \bar{x}(R)^l \smile t \frown u \smile \bar{x}(R)^r \frown (x_j^l @ x_j^r) \smile x_j \frown w_j \smile v_j \frown \cdots$$

But then we can replace this by

$$\cdots \frown v_i \smile v_j \frown \cdots$$

since $v_i \smile v_j$ in $\mathcal{G}(P)$. ∎

For the remainder of this section, we will assume that *all proof expressions under consideration are linear and acyclic.*

7.1 Determinacy

We now prove a suitable version of Determinacy for PE_2. To state this properly, we need a definition.

Definition 7.3 *A renaming is a permutation $\rho : \mathcal{N} \cong \mathcal{N}$. This is extended to a substitution on terms, coequations, proof expressions etc. in the usual way. Now we define structural equivalence of proof expressions:*

$$P \equiv Q \quad \stackrel{\text{def}}{\Longleftrightarrow} \quad \exists \rho .\, (\rho(P) \leftrightharpoons^* Q).$$

Structural equivalence merely factors out irrelevant syntactic detail, similarly to α-equivalence in the λ-calculus [Bar84].

We can now state the appropriate form of Determinacy:

$$\textbf{Determinacy}: \quad P {\Downarrow} Q \;\&\; P {\Downarrow} R \implies Q \equiv R.$$

The remainder of this subsection is devoted to proving this property.

Firstly, it will be convenient to decompose the transition relation $\longrightarrow$. We define $P \longrightarrow_r Q$ iff $P \longrightarrow Q$ can be derived using the Reaction rules, and $P \longrightarrow_c Q$ iff $P \longrightarrow Q$ can be derived using the Cleanup rule. Clearly $\longrightarrow \; = \; \longrightarrow_r \cup \longrightarrow_c$.

Proposition 7.4 *If $P \longrightarrow_r Q$ and $P \longrightarrow_r R$, then either $Q \leftrightharpoons^* R$, or for some S, $Q \longrightarrow_r S$ and $R \longrightarrow_r S$.*

Proof. The only critical pair for $\longrightarrow_r$ arises from the Communication rule. Acyclicity precludes the situation $x \perp y, x \perp y$. The only remaining possibility is:

$$\begin{tikzcd}
 & t \perp x, x \perp y, y \perp u \arrow[dl, "r"] \arrow[dr, "r"] & \\
t \perp y, y \perp u \arrow[dr, "r"] & & t \perp x, x \perp u \arrow[dl, "r"] \\
 & t \perp u &
\end{tikzcd}$$

∎

Proposition 7.5 *If $P \longrightarrow_c Q$ and $P \longrightarrow_c R$, then either $Q \equiv R$, or for some S, $Q \longrightarrow_c S$ and $R \longrightarrow_c S$.*

Proof. The only case for a critical pair is

$$\begin{tikzcd}
 & x \perp y, \Theta; \bar{t} \arrow[dl, "c"] \arrow[dr, "c"] & \\
\Theta; \bar{t}[x/y] & \equiv & \Theta; \bar{t}[y/x]
\end{tikzcd}$$

∎

Proposition 7.6 *If $P \longrightarrow_r Q$ and $P \longrightarrow_c R$, then for some S, $Q \longrightarrow_c S$ and $R \longrightarrow_r S$.*

Proof. By linearity, the only case for a critical pair is

$$\begin{tikzcd}
 & x \perp y, y \perp t, \Theta; \bar{t} \arrow[dl, "r"] \arrow[dr, "c"] & \\
x \perp t, \Theta; \bar{t} \arrow[d, "c"] & & y \perp t, \Theta; \bar{t}[y/x] \arrow[d, "r"] \\
\Theta; \bar{t}[t/x] & = & \Theta; \bar{t}[y/x][t/y]
\end{tikzcd}$$

∎

Proposition 7.7 *If $P \longrightarrow_c Q \longrightarrow_r R$, then for some S, $P \longrightarrow_r S \longrightarrow_c R$.*

Proof. If $P = x \perp t, \Theta; \bar{t} \longrightarrow_c \Theta; \bar{t}[t/x] \longrightarrow_r \Xi; \bar{t}[t/x]$, then

$$P \longrightarrow_r x \perp t, \Xi; \bar{t} \longrightarrow_c \Xi; \bar{t}[t/x]. \quad ∎$$

By standard arguments [Bar84], Propositions 7.4–7.7 imply the corresponding properties for $\longrightarrow_r^*$, $\longrightarrow_c^*$.

Now we show that $\longrightarrow^*$ is confluent up to structural equivalence.

Theorem 7.8 *If $P \longrightarrow^* P'$ and $P \longrightarrow^* P''$, then for some $Q' \equiv Q''$, $P' \longrightarrow^* Q'$ and $P'' \longrightarrow^* Q''$.*

Proof. Consider the following diagram:

$$\begin{tikzcd}
P \arrow[r, "*", "r"'] \arrow[d, "*"', "r"] \arrow[dr, phantom, "(1)"] & \cdot \arrow[r, "*", "c"'] \arrow[d, "*"', "r"] \arrow[dr, phantom, "(2)"] & P' \arrow[d, "*"', "r"] \\
\cdot \arrow[r, "*", "r"'] \arrow[d, "*"', "c"] \arrow[dr, phantom, "(3)"] & \cdot \arrow[r, "*", "c"'] \arrow[d, "*"', "c"] \arrow[dr, phantom, "(4)"] & \cdot \arrow[d, "*"', "c"] \\
P'' \arrow[r, "*", "r"'] & \cdot \arrow[r, "*", "c"'] & \cdot
\end{tikzcd}$$

Firstly, by Proposition 7.7, $P \longrightarrow^* P'$ can be written as $P \longrightarrow_r^* \longrightarrow_c^* P'$, and similarly for $P \longrightarrow^* P''$. Next, (1) can be filled in up to $\leftrightharpoons^*$ by Proposition 7.4; then (2) and (3) can be filled in by Propostion 7.6; and finally (4) can be filled in up to $\equiv$ by Proposition 7.5. ∎

As an immediate corollary to confluence, we have:

Theorem 7.9 (Determinacy) $P{\Downarrow}Q \;\&\; P{\Downarrow}R \implies Q \equiv R$.

7.2 Convergence

Our aim in this subsection is to prove:

$$\textbf{Convergence:}\quad \mathsf{PE}_2 \vdash \Theta; \bar{t} : \Gamma \implies \exists Q.\, (\Theta; \bar{t}{\Downarrow}Q).$$

In fact, we shall prove a stronger result: every typable proof expression P is *canonically strongly normalizing* (notation: $\mathsf{CSN}(P)$), *i.e.* every transition sequence

$$P \longrightarrow P_1 \longrightarrow P_2 \longrightarrow \cdots$$

ends in a canonical form.

Firstly, we will prove a very useful lemma, which will play a rôle in our work analogous to Girard's use of the Standardization Theorem in [Gir87].

Lemma 7.10 $P \longrightarrow^* Q \;\&\; \mathsf{CSN}(Q) \implies \mathsf{CSN}(P)$.

Proof. We begin by making a number of reductions of what is to be proved. Firstly, it clearly suffices to prove

$$P \longrightarrow Q \;\&\; \mathsf{CSN}(Q) \implies \mathsf{CSN}(P)$$

since the general case follows immediately by induction. Next, note that it suffices to prove

$$P \longrightarrow Q \;\&\; \mathsf{SN}(Q) \implies \mathsf{SN}(P) \tag{19}$$

Indeed, suppose (19) holds, $P \longrightarrow Q$ and $\mathsf{CSN}(Q)$. By (19), every transition sequence from P ends in a normal form; by Determinacy, this is the same (up to $\equiv$) as the (any) canonical form that Q evaluates to.

Finally, it suffices to prove that:

$$P \longrightarrow Q \;\&\; \mathsf{SN}_r(Q) \implies \mathsf{SN}_r(P) \tag{20}$$

where $\mathsf{SN}_r(P)$ means that every $\longrightarrow_r$-sequence starting from P ends in a $\longrightarrow_r$-normal form. In fact, $\mathsf{SN}(P) \iff \mathsf{SN}_r(P)$. To see this, suppose for a contradiction that $\mathsf{SN}_r(P)$. and there is an infinite sequence

$$P \longrightarrow P_1 \longrightarrow P_2 \longrightarrow \cdots. \tag{21}$$

If there were infinitely many $\longrightarrow_r$-steps in the sequence, then by Proposition 7.7 we could construct an infinite $\longrightarrow_r$-sequence from P, contradictng $\mathsf{SN}_r(P)$. So there are only finitely many $\longrightarrow_r$-steps in (21), hence there is an infinite $\longrightarrow_c$-sequence from some P_n. But this is impossible, as $\longrightarrow_c$ is obviously strongly normalizing.

Finally, we prove (20). Suppose that $P \longrightarrow Q_1$, and that there is an infinite sequence

$$P \longrightarrow_r P_1 \longrightarrow_r P_2 \longrightarrow_r \cdots$$

Using Propositions 7.4 and 7.6, we can proceed as in the following diagram:

$$\begin{tikzcd}
P \arrow[r, "r"'] \arrow[d] & P_1 \arrow[r, "r"'] \arrow[d, dotted] & P_2 \arrow[r, "r"'] \arrow[d, dotted] & \cdots \\
Q_1 \arrow[r, dotted, "r"'] & Q_2 \arrow[r, dotted, "r"'] & Q_3 \arrow[r, dotted, "r"'] & \cdots
\end{tikzcd}$$

Either this diagram can be extended indefinitely, or $Q_n \Leftarrow^* P_n$ for some n. In either case, there is an infinite $\longrightarrow_r$-sequence from Q. So $\mathsf{SN}_r(Q) \Rightarrow \mathsf{SN}_r(P)$, as required. ∎

If we define $P\Downarrow \;\stackrel{\text{def}}{\iff}\; \exists Q.\,(P\Downarrow Q)$, then as an immediate corollary of this Lemma we have

Proposition 7.11 $P\Downarrow \iff \mathsf{CSN}(P)$.

In the light of this Proposition, we write $P\Downarrow$ rather than $\mathsf{CSN}(P)$. Also, we define $P\Uparrow \;\stackrel{\text{def}}{\iff}\; \neg(P\Downarrow)$.

We now proceed with the proof of Convergence, following much the same lines as Girard's proof of Strong Normalization for PN2 in [Gir87], but in the style of Section 5, and with some significant modifications dictated by the differences in our framework.

Firstly, some notation. Given proof expressions

$$P = \Theta; \bar{t}, t, \quad Q = \Xi; \bar{u}, u$$

we define

$$\mathtt{Cut}(P, Q) = \Theta, \Xi, t \perp u; \bar{t}, \bar{u}.$$

More precisely, we choose $P' \equiv P$, $Q' \equiv Q$ such that $\mathcal{N}(P') \cap \mathcal{N}(Q') = \varnothing$, and form $\mathtt{Cut}(P', Q')$ (compare the definition of substitution in [Bar84]); we will generally take this renaming for granted, and not refer to it explicitly. Note that $\mathtt{Cut}(P, Q)$ is only defined for proof expressions with non-empty main body; we will shortly take steps to excise this minor nuisance.

Now we define

$$P \perp Q \stackrel{\text{def}}{\iff} \mathtt{Cut}(P, Q)\Downarrow.$$

(This definition is easily seen to be independent of the choice of P', Q'.) Let $\mathbb{PE}$ be the set of linear, acyclic proof expressions with non-empty main body. Given $U \subseteq \mathbb{PE}$, we can define

$$U^{\perp} = \{P \in \mathbb{PE} \mid \forall Q \in U. (P \perp Q)\}.$$

Now by standard facts about Galois connections [Coh81], we have:

Proposition 7.12 *(i) The operator* $(\cdot)^{\perp\perp}$ *is monotone, inflationary and idempotent.*
(ii) $U^{\perp\perp\perp} = U^{\perp}$.
(iii) $\forall P \in \mathbb{PE}, U \subseteq \mathbb{PE}. (P \perp U \iff P \perp U^{\perp\perp})$.

A *semantic type* is a subset $U \subseteq \mathbb{PE}$ satisfying:

- $U \neq \varnothing$
- $U\Downarrow$ *i.e.* $\forall P \in U. (P\Downarrow)$
- $U = U^{\perp\perp}$

We write $\mathcal{U}$ for the set of all semantic types.

Lemma 7.13 $\mathtt{Cut}(P, Q)\Downarrow \implies P\Downarrow \,\&\, Q\Downarrow$.

Proof. Suppose $P\Uparrow$. There are two possibilities:

(1) $\neg\mathsf{SN}(P)$. By Lemma 7.10, this implies $\neg\mathsf{SN}_r(P)$; but then $\neg\mathsf{SN}_r(\mathtt{Cut}(P, Q))$, and so $\mathtt{Cut}(P, Q)\Uparrow$.

(2) P has a non-canonical normal form. This means that $P \longrightarrow_r^* t \perp u, \Theta; \bar{t}$, where neither t or u are names, and no reaction rule is applicable to $t \perp u$. But then $\mathtt{Cut}(P, Q) \longrightarrow_r^* t \perp u, \Xi; \bar{u}$, and any normal form derivable from this expression will still contain the co-equation $t \perp u$, and hence be non-canonical.

The case when $Q\Uparrow$ is entirely similar. ∎

Lemma 7.14 *For all* $U \subseteq \mathbb{PE}$*:*

$$(i) \quad U \neq \varnothing \implies U^{\perp}\Downarrow$$
$$(ii) \quad U\Downarrow \implies U^{\perp} \neq \varnothing.$$

Proof. (i) If $P \in U$, $Q \in U^{\perp}$, then $\mathtt{Cut}(P, Q)\Downarrow$, so by Lemma 7.13, $P\Downarrow$.

(ii) If $U\Downarrow$, then $; x, x \in U^{\perp}$. To see this, suppose $P \in U$. Then $\mathtt{Cut}(P, ; x, x) \longrightarrow P$, and $P\Downarrow$, so by Lemma 7.10, $\mathtt{Cut}(P, ; x, x)\Downarrow$ as required. ∎

Proposition 7.15 *If* $U \subseteq \mathbb{PE}$ *satisfies* $U \neq \varnothing$ *and* $U\Downarrow$*, then* $U^{\perp} \in \mathcal{U}$.

PROOF. By Lemma 7.14 and Proposition 7.12(ii). ∎

We will now give a realizability interpretation of the Linear types as elements of $\mathcal{U}$.

Firstly, we define some operations on proof expressions, corresponding to the logical rules of PE_2.

$$
\begin{array}{rcl}
\mathtt{Id}_x & = & ;x,x \\
\mathtt{Unit} & = & ;* \\
\mathtt{Perp}(\Theta;\bar{t}) & = & \Theta;\bar{t},\circledast \\
\mathtt{Par}(\Theta;\bar{t},t,u) & = & \Theta;\bar{t},t \parr u \\
\mathtt{Times}(\Theta;\bar{t},t,\Xi;\bar{u},u) & = & \Theta,\Xi;\bar{t},\bar{u},t\otimes u \\
\mathtt{With}_{\bar{x}}(P,Q) & = & \bar{x}(P \,[\!]\, Q) \\
\mathtt{Plusl}(\Theta;\bar{t},t) & = & \Theta;\bar{t},\mathtt{inl}(t) \\
\mathtt{Plusr}(\Theta;\bar{t},t) & = & \Theta;\bar{t},\mathtt{inr}(t) \\
\mathtt{OfCourse}_{\bar{x}}(P) & = & \bar{x}(P) \\
\mathtt{Der}(\Theta;\bar{t},t) & = & \Theta;\bar{t},\ ?t \\
\mathtt{Weak}(\Theta;\bar{t}) & = & \Theta;\bar{t},\_ \\
\mathtt{Con}(\Theta;\bar{t},t,u) & = & \Theta;\bar{t},t@u
\end{array}
$$

The same provisos about renaming which we made for the Cut construct apply to all these operations; this ensures that, except for the Cut, none of these operations create any communication between their arguments. It easily follows that all of the above operations $F(P_1,\ldots,P_n)$ satisfy:

$$P_1\Downarrow \,\&\, \cdots \,\&\, P_n\Downarrow \implies F(P_1,\ldots,P_n)\Downarrow.$$

In a sense, the whole purpose of the realizability semantics is to formulate a sufficiently strong "inductive hypothesis" to allow us to extend this to the Cut.

Now we define:

$$
\begin{array}{rcl}
\bot & = & \{\mathtt{Unit}\}^{\bot} \\
U \parr V & = & \{\mathtt{Times}(P,Q) \mid P\in U^{\bot}, Q\in V^{\bot}\}^{\bot} \\
U \,\&\, V & = & (\{\mathtt{Plusl}(P) \mid P\in U^{\bot}\}\cup\{\mathtt{Plusr}(Q) \mid Q\in V^{\bot}\})^{\bot} \\
?U & = & \{\mathtt{OfCourse}(P) \mid P\in U^{\bot}\}^{\bot}
\end{array}
$$

and for $F:\mathcal{U}\to\mathcal{U}$

$$\forall(F) = (\bigcup\{F(U)^{\bot} \mid U\in\mathcal{U}\})^{\bot}$$

By Propositions 7.15 and 7.12 and the remarks immediately preceding these definitions, they do yield semantic types.

The remaining connectives are defined by duality, so as to force the equations (18) to be satisfied:

$$
\begin{array}{rcl}
\mathbf{1} & = & \perp^{\perp} \\
U \otimes V & = & (U^{\perp} \parr V^{\perp})^{\perp} \\
U \oplus V & = & (U^{\perp} \,\&\, V^{\perp})^{\perp} \\
!U & = & (?U^{\perp})^{\perp} \\
\exists(F) & = & \forall(\lambda U.F(U)^{\perp})^{\perp}
\end{array}
$$

These definitions induce a semantic function

$$[\![\cdot]\!] : \mathsf{TExp} \to \mathsf{TEnv} \to \mathcal{U}$$

where TExp is the set of Linear type expressions, *i.e.* formulae of CLL, and $\mathsf{TEnv} = \mathsf{TVar} \to \mathcal{U}$ is the set of type environments, ranged over by η.

Lemma 7.16 *For all $A \in \mathsf{TExp}$, $\eta \in \mathsf{TEnv}$:*

$$([\![A]\!]\eta)^{\perp} = [\![A^{\perp}]\!]\eta.$$

Proof. Immediate from the fact that the realizability interpretation of the Linear connectives satisfies (18). ∎

We can now give a realizability interpretation of PE_2 sequents. Firstly, given $P = \Theta; t, \bar{t}$, $Q = \Xi; \bar{u}, u$ we define

$$P \cdot Q = \Theta, \Xi, t \perp u; \bar{t}, \bar{u}$$

(with the standard proviso about renaming to ensure $\mathcal{N}(P)$ disjoint from $\mathcal{N}(Q)$); and write $PQ_1 \cdots Q_k$ to abbreviate $(\cdots(P \cdot Q_1) \cdots \cdot Q_k)$.

Now we define

$$\models \Theta; \bar{t} : \Gamma \;\stackrel{\mathrm{def}}{\iff}\; \forall \eta \in \mathsf{TEnv}, \bar{P} \in [\![\Gamma^{\perp}]\!]\eta.\, (P\bar{P}\Downarrow)$$

where $P = \Theta; \bar{t}$, $\Gamma = A_1, \ldots, A_k$, $[\![\Gamma^{\perp}]\!]\eta = [\![{A_1}^{\perp}]\!]\eta, \ldots, [\![{A_k}^{\perp}]\!]\eta$.

As a final preliminary, we define the *shift* operator on $\mathbb{PE}$:

$$\sigma(\Theta; t, \bar{t}) = \Theta; \bar{t}, t.$$

Clearly, $\sigma(P)\Downarrow \iff P\Downarrow$. More generally, if π is a permutation on $\{1, \ldots, k\}$, then

$$\Theta; t_{\pi 1}, \ldots, t_{\pi k}\Downarrow \iff \Theta; t_1, \ldots, t_k\Downarrow.$$

Note the following relationship:

$$\mathtt{Cut}(\sigma(P), Q) = P \cdot Q.$$

We can now prove the basic result on the realizability interpretation.

Theorem 7.17 (Realizability) $\mathsf{PE}_2 \vdash \Theta; \bar{t} : \Gamma \implies \models \Theta; \bar{t} : \Gamma.$

PROOF. By induction on derivations in PE_2.

(1) Axiom:

$$\overline{\vdash; x : A^\perp, x : A}$$

Fix $\eta \in \mathsf{TEnv}$, $P \in [\![A^{\perp\perp}]\!]\eta = ([\![A^\perp]\!]\eta)^\perp$, $Q \in [\![A^\perp]\!]\eta$. We must show that $\mathtt{Id}_x PQ{\Downarrow}$. But $\mathtt{Id}_x PQ \longrightarrow \mathtt{Cut}(P,Q)$, and $\mathtt{Cut}(P,Q){\Downarrow}$, since $P \perp Q$ by assumption. Hence by 7.10, $\mathtt{Id}_x PQ{\Downarrow}$.

(2) Exchange: immediate from the remarks about permutations preceding the Theorem.

(3) Cut:

$$\frac{\vdash \Theta; \bar{t} : \Gamma, t : A \quad \vdash \Xi; \bar{u} : \Delta, u : A^\perp}{\vdash \Theta, \Xi, t \perp u; \bar{t} : \Gamma, \bar{u} : \Delta}$$

Let $P = \Theta; \bar{t}, t$, $Q = \Xi; \bar{u}, u$, and fix $\eta \in \mathsf{TEnv}$, $\bar{P} \in [\![\Gamma^\perp]\!]\eta$, $\bar{Q} \in [\![\Delta^\perp]\!]\eta$. We must show that $\mathtt{Cut}(P,Q)\bar{P}\bar{Q}{\Downarrow}$. By induction hypothesis, for all $R \in [\![A^\perp]\!]\eta$, $P\bar{P}R{\Downarrow}$, and for all $S \in [\![A]\!]\eta$, $Q\bar{Q}S{\Downarrow}$. Hence $\sigma(P\bar{P}) \in ([\![A^\perp]\!]\eta)^\perp = [\![A]\!]\eta$, and $\sigma(Q\bar{Q}) \in ([\![A]\!]\eta)^\perp$, so $\mathtt{Cut}(\sigma(P\bar{P}), \sigma(Q\bar{Q})){\Downarrow}$. But $\mathtt{Cut}(\sigma(P\bar{P}), \sigma(Q\bar{Q})) \Longleftarrow^* \mathtt{Cut}(P,Q)\bar{P}\bar{Q}$, so $\mathtt{Cut}(P,Q)\bar{P}\bar{Q}{\Downarrow}$.

(4) Perp:

$$\frac{\vdash \Theta; \bar{t} : \Gamma}{\vdash \Theta; \bar{t} : \Gamma, \circledast : \perp}$$

Let $P = \Theta; \bar{t}$ and fix $\eta \in \mathsf{TEnv}$, $\bar{Q} \in [\![\Gamma^\perp]\!]\eta$. We must show that for all $Q \in [\![\perp^\perp]\!]\eta$, $\mathtt{Perp}(P)\bar{Q}Q{\Downarrow}$, *i.e.* that $\sigma(\mathtt{Perp}(P)\bar{Q}) \perp \{\mathtt{Unit}\}^{\perp\perp}$. By Proposition 7.12(iii), it suffices to show that $\sigma(\mathtt{Perp}(P)\bar{Q}) \perp \{\mathtt{Unit}\}$, *i.e.* that $\mathtt{Perp}(P)\bar{Q}\mathtt{Unit}{\Downarrow}$. But $\mathtt{Perp}(P)\bar{Q}\mathtt{Unit} \longrightarrow P\bar{Q}$, and by induction hypothesis $P\bar{Q}{\Downarrow}$, so by 7.10, $\mathtt{Perp}(P)\bar{Q}Q{\Downarrow}$.

(5) Unit:

$$\overline{\vdash; * : \mathbf{1}}$$

We must show that for all η, $P \in [\![\mathbf{1}^\perp]\!]\eta$, $\mathtt{Unit}P{\Downarrow}$. By Proposition 7.12(i), $(\cdot)^{\perp\perp}$ is inflationary, so $\mathtt{Unit} \in [\![\mathbf{1}]\!]\eta$, and $\mathtt{Unit}P = \mathtt{Cut}(\mathtt{Unit}, P){\Downarrow}$.

(6) Par:

$$\frac{\vdash \Theta; \bar{t} : \Gamma, t : A, u : B}{\vdash \Theta; \bar{t} : \Gamma, t \parr u : A \parr B}$$

Let $P = \Theta; \bar{t}, t, u$, and fix $\eta \in \mathsf{TEnv}$, $\bar{Q} \in [\![\Gamma^\perp]\!]\eta$. We must show that for all $Q \in [\![(A \parr B)^\perp]\!]\eta$, $\mathtt{Par}(P)\bar{Q}Q{\Downarrow}$, *i.e.* that $\sigma(\mathtt{Par}(P)\bar{Q}) \perp [\![(A \parr B)^\perp]\!]\eta$. Applying Proposition 7.12(iii) to the definition of $[\![(A \parr B)^\perp]\!]\eta$, we see that it is sufficient to consider Q of the form $\mathtt{Times}(R,S)$, where $R \in [\![A^\perp]\!]\eta$, $S \in [\![B^\perp]\!]\eta$. But $\mathtt{Par}(P)\bar{Q}\mathtt{Times}(R,S) \longrightarrow P\bar{Q}RS$, and by the induction hypothesis $P\bar{Q}RS{\Downarrow}$, so by 7.10, $\mathtt{Par}(P)\bar{Q}\mathtt{Times}(R,S){\Downarrow}$.

(7) Times:

$$\frac{\vdash \Theta; \bar{t} : \Gamma, t : A \quad \vdash \Xi; \bar{u} : \Delta, u : B}{\vdash \Theta, \Xi; \bar{t} : \Gamma, \bar{u} : \Delta, t \otimes u : A \otimes B}$$

Let $P = \Theta; \bar{t}, t$, $Q = \Xi; \bar{u}, u$, and fix $\eta \in \mathsf{TEnv}$, $\bar{P} \in [\![\Gamma^\perp]\!]\eta$, $\bar{Q} \in [\![\Delta^\perp]\!]\eta$. We must show that for all $R \in [\![(A \otimes B)^\perp]\!]\eta$, $\mathtt{Times}(P,Q)\bar{P}\bar{Q}R{\Downarrow}$. By induction hypothesis, for all

$S \in [\![A^\perp]\!]\eta$, $P\bar{P}S\Downarrow$, and for all $T \in [\![B^\perp]\!]\eta$, $Q\bar{Q}T\Downarrow$. Hence $\sigma(P\bar{P}) \in ([\![A^\perp]\!]\eta)^\perp = [\![A]\!]\eta$, and $\sigma(Q\bar{Q}) \in ([\![B^\perp]\!]\eta)^\perp$. Applying Proposition 7.12(i) (specifically, the fact that $(\cdot)^{\perp\perp}$ is inflationary) to the definition of $[\![(A \parr B)^\perp]\!]\eta$, we see that

$$\mathtt{Times}(\sigma(P\bar{P}), \sigma(Q\bar{Q})) \in ([\![A^\perp \parr B^\perp]\!]\eta)^\perp,$$

and hence that $\mathtt{Cut}(\mathtt{Times}(\sigma(P\bar{P}), \sigma(Q\bar{Q})), R)\Downarrow$. But

$$\mathtt{Cut}(\mathtt{Times}(\sigma(P\bar{P}), \sigma(Q\bar{Q})), R)\Downarrow \Rightarrow^* \mathtt{Times}(P, Q)\bar{P}\bar{Q}R,$$

so $\mathtt{Times}(P, Q)\bar{P}\bar{Q}R\Downarrow$.

(8) With:

$$\frac{\vdash \Theta; \bar{t} : \Gamma, t : A \quad \vdash \Xi; \bar{u} : \Gamma, u : B}{\vdash; \bar{x} : \Gamma, \bar{x}(P \parallel Q) : A \,\&\, B}$$

where $P = \Theta; \bar{t}, t$, $Q = \Xi; \bar{u}, u$. Fix $\eta \in \mathsf{TEnv}$, $\bar{P} \in [\![\Gamma^\perp]\!]\eta$. We must show that for all $R \in [\![(A \,\&\, B)^\perp]\!]\eta$, $\mathtt{With}(P, Q)\bar{P}R\Downarrow$. Reasoning as in the case for Par, it suffices to consider Q of the form *either* $\mathtt{Plusl}(S)$, $S \in [\![A^\perp]\!]\eta$, *or* $\mathtt{Plusr}(T)$, $T \in [\![B^\perp]\!]\eta$. In the first case, $\mathtt{With}(P, Q)\bar{P}\mathtt{Plusl}(S) \longrightarrow^* P\bar{P}S$, and by induction hypothesis $P\bar{P}S\Downarrow$, so by 7.10, $\mathtt{With}(P, Q)\bar{P}R\Downarrow$. The second case is similar.

(9) Plus Left:

$$\frac{\vdash \Theta; \bar{t} : \Gamma, t : A}{\vdash \Theta; \bar{t} : \Gamma, \mathtt{inl}(t) : A \oplus B}$$

Let $P = \Theta; \bar{t}, t$, and fix $\eta \in \mathsf{TEnv}$, $\bar{Q} \in [\![\Gamma^\perp]\!]\eta$. We must show that for all $Q \in [\![(A \oplus B)^\perp]\!]\eta$, $\mathtt{Plusl}(P)\bar{Q}Q\Downarrow$. By induction hypothesis, for all $R \in [\![A^\perp]\!]\eta$, $P\bar{Q}R\Downarrow$, so $\sigma(P\bar{Q}) \in ([\![A]\!]\eta)^\perp$, and

$$\mathtt{Plusl}(\sigma(P\bar{Q})) \in ([\![A^\perp \,\&\, B^\perp]\!]\eta)^\perp = [\![A \oplus B]\!]\eta,$$

so $\mathtt{Cut}(\mathtt{Plusl}(\sigma(P\bar{Q})), Q)\Downarrow$. But

$$\mathtt{Cut}(\mathtt{Plusl}(\sigma(P\bar{Q})), Q) \Rightarrow^* \mathtt{Plusl}(P)\bar{Q}Q,$$

so $\mathtt{Plusl}(P)\bar{Q}Q\Downarrow$.

(10) Plus Right: similar to Plus Left.

(11) Dereliction:

$$\frac{\vdash \Theta; \bar{t} : \Gamma, t : A}{\vdash \Theta; \bar{t} : \Gamma, \ ?t : ?A}$$

Let $P = \Theta; \bar{t}, t$, and fix $\eta \in \mathsf{TEnv}$, $\bar{Q} \in [\![\Gamma^\perp]\!]\eta$. We must show that for all $Q \in [\![(?A)^\perp]\!]\eta$, $\mathtt{Der}(P)\bar{Q}Q\Downarrow$. Reasoning as in the case for Par, it suffices to consider Q of the form $\mathtt{OfCourse}(R)$, $R \in [\![A^\perp]\!]\eta$. But $\mathtt{Der}(P)\bar{Q}\mathtt{OfCourse}(R) \longrightarrow^* P\bar{Q}R$, and by the induction hypothesis $P\bar{Q}R\Downarrow$. Hence by 7.10, $\mathtt{Der}(P)\bar{Q}Q\Downarrow$.

(12) Contraction:

$$\frac{\vdash \Theta; \bar{t} : \Gamma, t : ?A, u : ?B}{\vdash \Theta; \bar{t} : \Gamma, t@u : ?A}$$

Let $P = \Theta;\bar{t},t$, and fix $\eta \in \mathsf{TEnv}$, $\bar{Q} \in [\![\Gamma^\perp]\!]\eta$. We must show that for all $Q \in [\![(\,?A)^\perp]\!]\eta$, $\mathtt{Con}(P)\bar{Q}Q\Downarrow$. Reasoning as in the case for Par, it suffices to consider Q of the form $\mathtt{OfCourse}(R)$, $R \in [\![A^\perp]\!]\eta$. But $\mathtt{Con}(P)\bar{Q}\mathtt{OfCourse}(R) \longrightarrow^* \mathtt{Con}_k(P\bar{Q}RR)$, where

$$\mathtt{Con}_k(\Xi;\bar{u},v_1,\ldots,v_k,w_1,\ldots,w_k) = \Xi;\bar{u},v_1@w_1,\ldots,v_k@w_k.$$

(Recall the stipulations on renaming in the definition of $P \cdot Q$.) Now

$$\mathtt{Con}_k(P\bar{Q}RR)\Downarrow \iff P\bar{Q}RR\Downarrow,$$

but by induction hypothesis $P\bar{Q}RR\Downarrow$, hence by 7.10, $\mathtt{Con}(P)\bar{Q}Q\Downarrow$.

(13) Weakening:

$$\frac{\vdash \Theta;\bar{t} : \Gamma}{\vdash \Theta;\bar{t} : \Gamma, \_ : ?A}$$

Let $P = \Theta;\bar{t}$, and fix $\eta \in \mathsf{TEnv}$, $\bar{Q} \in [\![\Gamma^\perp]\!]\eta$. We must show that for all $Q \in [\![(\,?A)^\perp]\!]\eta$, $\mathtt{Weak}(P)\bar{Q}Q\Downarrow$. Once again, it suffices to consider Q of the form $\mathtt{OfCourse}(R)$, $R \in [\![A^\perp]\!]\eta$. But $\mathtt{Weak}(P)\bar{Q}\mathtt{OfCourse}(R) \longrightarrow^* \mathtt{Weak}^k(P\bar{Q})$, and $\mathtt{Weak}^k(P\bar{Q})\Downarrow \iff P\bar{Q}\Downarrow$. By induction hypothesis $P\bar{Q}\Downarrow$, hence by 7.10, $\mathtt{Weak}(P)\bar{Q}Q\Downarrow$.

(14) Of Course:

$$\frac{\vdash \Theta;\bar{t} : ?\Gamma, t : A}{\vdash \Theta;\bar{x} : ?\Gamma, \bar{x}(P) : !A}$$

where $P = \Theta;\bar{t},t$. Fix $\eta \in \mathsf{TEnv}$, $\bar{Q} \in [\![\,?\Gamma^\perp]\!]\eta$. We must show that for all $Q \in [\![(\,!A)^\perp]\!]\eta$, $\mathtt{OfCourse}(P)\bar{Q}Q\Downarrow$. By induction hypothesis, for all $R \in [\![A^\perp]\!]\eta$, $P\bar{Q}R\Downarrow$, hence $\sigma(P\bar{Q}) \in ([\![A^\perp]\!]\eta)^\perp$, so $\mathtt{OfCourse}(\sigma(P\bar{Q})) \in ([\![\,?(A^\perp)]\!]\eta)^\perp$, and $\mathtt{Cut}(\mathtt{OfCourse}(\sigma(P\bar{Q})),Q)\Downarrow$. We must show that this implies that $\mathtt{OfCourse}(P)\bar{Q}Q\Downarrow$.

Firstly, we claim that it is sufficient to prove that $\mathtt{OfCourse}(P)\bar{Q}Q\Downarrow$ for $\bar{Q}$ of the form

$$\mathtt{OfCourse}(\bar{R}) = \mathtt{OfCourse}(R_1),\ldots,\mathtt{OfCourse}(R_k).$$

To see this, define $\sigma^{-1}(\Theta;\bar{t},t) = \Theta;t,\bar{t}$, and note that

$$\begin{array}{ll}
\mathtt{OfCourse}(P)\bar{Q}Q\Downarrow & \iff \\
\mathtt{Par}^{k-1}(\sigma^{-1}(\mathtt{OfCourse}(P))Q)\mathtt{Times}^{k-1}(\bar{Q})\Downarrow & \iff \\
\sigma(\mathtt{Par}^{k-1}(\sigma^{-1}(\mathtt{OfCourse}(P))Q)) \perp \mathtt{Times}^{k-1}(\bar{Q}), &
\end{array}$$

and that

$$\begin{array}{rcl}
\sigma(\mathtt{Par}^{k-1}(\sigma^{-1}(\mathtt{OfCourse}(P))Q)) & \perp & [\![\bigotimes(\,?\Gamma^\perp)]\!]\eta \\
& = & \{\mathtt{Times}^{k-1}(\mathtt{OfCourse}(\bar{R})) \mid \bar{R} \in [\![\Gamma^\perp]\!]\eta\}^{\perp\perp}
\end{array}$$

if and only if

$$\sigma(\mathtt{Par}^{k-1}(\sigma^{-1}(\mathtt{OfCourse}(P))Q)) \perp \{\mathtt{Times}^{k-1}(\mathtt{OfCourse}(\bar{R})) \mid \bar{R} \in [\![\Gamma^\perp]\!]\eta\},$$

by Proposition 7.12(iii).

Next, we will establish the desired relationship between $\mathtt{Cut}(\mathtt{OfCourse}(\sigma(P\bar{Q})), Q)\Downarrow$ and $\mathtt{OfCourse}(P)\bar{Q}Q\Downarrow$. At the corresponding point in his proof in [Gir87], Girard is able to use the commutative conversions, under which we would have

$$\mathtt{OfCourse}(P)\bar{Q}Q \longrightarrow^* \mathtt{Cut}(\mathtt{OfCourse}(\sigma(P\bar{Q})), Q).$$

As the commutative conversions are not part of our calculus, we need a more elaborate argument. We shall use a technique inspired by concurrency theory [Mil89]. We say that a relation $\mathcal{R} \subseteq \mathbb{PE}^2$ is a *simulation* if it satisfies

$$P\mathcal{R}Q \;\&\; P \longrightarrow R \implies \exists P', Q'.(R \longrightarrow^* P' \;\&\; Q \longrightarrow^+ Q' \;\&\; P'\mathcal{R}Q').$$

We need to establish some notation. We can write

$$\begin{aligned}
\mathtt{OfCourse}(P)\bar{Q}Q &= \Xi_0, \bar{x} \perp \tilde{y}(\tilde{Q}), \bar{x}(P) \perp v; \tilde{y}, \bar{v} \\
\mathtt{Cut}(\mathtt{OfCourse}(\sigma(P\bar{Q})), Q) &= \Xi_0, \bar{z}(R) \perp v; \tilde{z}, \bar{v}
\end{aligned}$$

where

$$\begin{aligned}
\bar{Q} &= \tilde{y}; \tilde{y}(\tilde{Q}) = \bar{y}_1; \bar{y}_1(\bar{Q}_1), \ldots, \bar{y}_k; \bar{y}_k(\bar{Q}_k), \\
R &= \bar{t} \perp \tilde{y}(\tilde{Q}), \Theta; \tilde{y}, t.
\end{aligned}$$

Moreover, $\mathcal{N}(\Xi_0; \bar{v}, v)$ is disjoint from the names occurring in the remainder of these expressions.

Now we define a relation $\mathcal{R}$ by: $P'\mathcal{R}Q'$ iff

$$\begin{aligned}
P' &= \Xi, \bar{x}^{s_1} \perp \tilde{y}(\tilde{Q})^{s_1}, \ldots, \bar{x}^{s_k} \perp \tilde{y}(\tilde{Q})^{s_k}, \bar{x}(P)^{s_1} \perp v_1, \ldots, \bar{x}(P)^{s_k} \perp v_k; \tilde{w}, \bar{v}' \\
Q' &= \Xi, \bar{z}(R)^{s_1} \perp v_1, \ldots, \bar{z}(R)^{s_k} \perp v_k; \tilde{w}[\tilde{z}^{s_1}, \ldots, \tilde{z}^{s_k}/\tilde{y}^{s_1}, \ldots, \tilde{y}^{s_k}], \bar{v}'
\end{aligned}$$

where $k \geq 0$, $\{s_1, \ldots, s_k\}$ is a pairwise incompatible subset of $\{l, r\}^*$, $\tilde{y}^{s_1}, \ldots, \tilde{y}^{s_k} \in \mathrm{A}\mathcal{N}(\tilde{w})$, and $\mathcal{N}(\Xi) \cup \bigcup_{i=1}^{k} \mathcal{N}(v_i) \cup \mathcal{N}(\bar{v}')$ is disjoint from the names occurring in the remainder of these expressions. Taking $k = 1$, $s_1 = \epsilon$, $v_1 = v$, $\Xi = \Xi_0$, $\tilde{w} = \tilde{y}$, we have $\mathtt{OfCourse}(P)\bar{Q}Q\mathcal{R}\mathtt{Cut}(\mathtt{OfCourse}(\sigma(P\bar{Q})), Q)$. We claim that $\mathcal{R}$ is a simulation. To prove this, suppose $P''\mathcal{R}Q''$, and consider the various cases for $P'' \longrightarrow S$. If the coequations to which a rule is applied occur in Ξ, we can take $P' = S$, and apply the same rule to the corresponding coequation in Q'' to get Q' with $P'\mathcal{R}Q'$. If the Communication rule is applied to some $x \perp u$ in Ξ, and some $\bar{x}(P)^{s_i} \perp v_i$ for which $v_i = x$ (the only possibility by the condition on names in Ξ incorporated into the definition of $\mathcal{R}$), then again the corresponding rule can be applied to Q'' to obtain Q' such that $P'\mathcal{R}Q'$, where $P' = S$. The final case to be considered is when some rule is applied to one of the coequations $\bar{x}(P)^{s_i} \perp v_i$. Without loss of generality (by the Magical Mixing rule) we can take $i = 1$. Now there are three sub-cases.

(i) The Read rule is applied, say with $v_1 = ?v'$. In this case, we have $S \longrightarrow^* P'$, $Q'' \longrightarrow^+ Q'$, where

$$\begin{aligned}
P' &= \Xi, \Theta^{s_1}, \bar{t}^{s_1} \perp \tilde{y}(\tilde{Q})^{s_1}, t^{s_1} \perp v', \\
&\quad\; \bar{x}^{s_2} \perp \tilde{y}(\tilde{Q})^{s_2}, \ldots, \bar{x}^{s_k} \perp \tilde{y}(\tilde{Q})^{s_k}, \bar{x}(P)^{s_2} \perp v_2, \ldots, \bar{x}(P)^{s_k} \perp v_k; \tilde{w}, \bar{v}' \\
Q' &= \Xi, \Theta^{s_1}, \bar{t}^{s_1} \perp \tilde{y}(\tilde{Q})^{s_1}, t^{s_1} \perp v', \\
&\quad\; \bar{z}(R)^{s_2} \perp v_2, \ldots, \bar{z}(R)^{s_k} \perp v_k; \tilde{w}[\tilde{z}^{s_2}, \ldots, \tilde{z}^{s_k}/\tilde{y}^{s_2}, \ldots, \tilde{y}^{s_k}], \bar{v}'.
\end{aligned}$$

(ii) The Discard rule is applied, with $v_1 = \_$. In this case, we have $S \longrightarrow^* P'$, $Q'' \longrightarrow^+ Q'$, where

$$\begin{aligned} P' &= \Xi, \bar{x}^{s_2} \perp \tilde{y}(\tilde{Q})^{s_2}, \ldots, \bar{x}^{s_k} \perp \tilde{y}(\tilde{Q})^{s_k}, \\ &\quad \bar{x}(P)^{s_2} \perp v_2, \ldots, \bar{x}(P)^{s_k} \perp v_k; \tilde{w}[\tilde{\_}/\tilde{y}^{s_1}], \bar{v}' \\ Q' &= \Xi, \bar{z}(R)^{s_2} \perp v_2, \ldots, \bar{z}(R)^{s_k} \perp v_k; \tilde{w}[\tilde{z}^{s_1}, \ldots, \tilde{z}^{s_k}/\tilde{y}^{s_1}, \ldots, \tilde{y}^{s_k}][\tilde{\_}/\tilde{z}^{s_1}], \bar{v}'. \end{aligned}$$

(iii) The Copy rule is applied, with $v_1 = v'@v''$. In this case, we have $S \longrightarrow^* P'$, $Q'' \longrightarrow^+ Q'$, where

$$\begin{aligned} P' &= \Xi, \bar{x}^{s_1 l} \perp \tilde{y}(\tilde{Q})^{s_1 l}, \bar{x}^{s_1 r} \perp \tilde{y}(\tilde{Q})^{s_1 r}, \bar{x}^{s_2} \perp \tilde{y}(\tilde{Q})^{s_2}, \ldots, \bar{x}^{s_k} \perp \tilde{y}(\tilde{Q})^{s_k}, \\ &\quad \bar{x}(P)^{s_1 l} \perp v', \bar{x}(P)^{s_1 r} \perp v'', \bar{x}(P)^{s_2} \perp v_2, \ldots, \bar{x}(P)^{s_k} \perp v_k; \tilde{w}[\tilde{y}^{s_1 l}@\tilde{y}^{s_1 r}/\tilde{y}^{s_1}], \bar{v}' \\ Q' &= \Xi, \bar{z}(R)^{s_1 l} \perp v', \bar{z}(R)^{s_1 r} \perp v'', \bar{z}(R)^{s_2} \perp v_2, \ldots, \bar{z}(R)^{s_k} \perp v_k; \\ &\quad \tilde{w}[\tilde{z}^{s_1}, \ldots, \tilde{z}^{s_k}/\tilde{y}^{s_1}, \ldots, \tilde{y}^{s_k}][\tilde{z}^{s_1 l}@\tilde{z}^{s_1 r}/\tilde{z}^{s_1}], \bar{v}'. \end{aligned}$$

Now we can use $\mathcal{R}$ to prove that $\mathtt{Cut}(\mathtt{OfCourse}(\sigma(P\bar{Q})), Q)\Downarrow \implies \mathtt{OfCourse}(P)\bar{Q}Q\Downarrow$. We define a (finite or infinite) sequence (P_n, Q_n), with $P_n \mathcal{R} Q_n$ for all n, as follows:

- $P_0 = \mathtt{OfCourse}(P)\bar{Q}Q$, $Q_0 = \mathtt{Cut}(\mathtt{OfCourse}(\sigma(P\bar{Q})), Q)$.
- If P_n is in normal form, then the sequence terminates at n. Otherwise, choose some P'' with $P_n \longrightarrow P''$, and then use the fact that $P_n \mathcal{R} Q_n$ to obtain P_{n+1}, Q_{n+1} such that $P'' \longrightarrow^* P_{n+1}$, $Q_n \longrightarrow^+ Q_{n+1}$, and $P_{n+1} \mathcal{R} Q_{n+1}$.

Now suppose that $\mathtt{OfCourse}(P)\bar{Q}Q\Uparrow$. There are two cases:

- If $\mathtt{OfCourse}(P)\bar{Q}Q$ has an infinite $\longrightarrow$-sequence, then applying 7.10, we can argue by induction that each P_n has an infinite $\longrightarrow$-sequence, and hence that the sequence (P_n, Q_n) is infinite. This means that (Q_n) is an infinite $\longrightarrow^+$-sequence.
- If $\mathtt{OfCourse}(P)\bar{Q}Q \longrightarrow^* S$, where S is a non-canonical normal form, then applying Determinacy and 7.10, some P_n in the above sequence is a non-canonical normal form. Clearly $P_n \mathcal{R} Q_n$ then implies that Q_n is a non-canonical normal form.

In either case, we see that

$$\mathtt{OfCourse}(P)\bar{Q}Q\Uparrow \implies \mathtt{Cut}(\mathtt{OfCourse}(\sigma(\mathtt{OfCourse}(P)\bar{Q})), Q)\Uparrow,$$

and so

$$\mathtt{Cut}(\mathtt{OfCourse}(\sigma(P\bar{Q})), Q)\Downarrow \implies \mathtt{OfCourse}(P)\bar{Q}Q\Downarrow,$$

as required.

(15) All:

$$\frac{\vdash \Theta; \bar{t} : \Gamma, t : A}{\vdash \Theta; \bar{t} : \Gamma, t : \forall\alpha.A} \quad (*)$$

Let $P = \Theta; \bar{t}, t$ and fix $\eta \in \mathsf{TEnv}$, $\bar{Q} \in [\![\Gamma^\perp]\!]\eta$. We must show that for all $Q \in [\![(\forall\alpha.A)^\perp]\!]\eta$, $P\bar{Q}Q\Downarrow$. Reasoning as in the case for Par, it suffices to consider $Q \in F(U)^\perp$ for some $U \in \mathcal{U}$, where $F = \lambda U.[\![A]\!]\eta[\alpha \mapsto U]$. By the eigenvariable condition $[\![\Gamma^\perp]\!]\eta = [\![\Gamma^\perp]\!]\eta[\alpha \mapsto U]$, so by the induction hypothesis (with respect to $\eta[\alpha \mapsto U]$), $P\bar{Q}Q\Downarrow$.

(16) Exists:

$$\frac{\vdash \Theta; \bar{t} : \Gamma, t : A[B/\alpha}{\vdash \Theta; \bar{t} : \Gamma, t : \exists\alpha.A}$$

Let $P = \Theta; \bar{t}, t$ and fix $\eta \in \mathsf{TEnv}$, $\bar{Q} \in [\![\Gamma^\perp]\!]\eta$. We must show that for all $Q \in [\![(\exists\alpha.A)^\perp]\!]\eta$, $P\bar{Q}Q\Downarrow$. By induction hypothesis, for all $R \in [\![A[B/\alpha]^\perp]\!]\eta$, $P\bar{Q}R\Downarrow$. Now

$$[\![A[B/\alpha]^\perp]\!]\eta = [\![A^\perp]\!]\eta[\alpha \mapsto [\![B]\!]\eta] = F(U)^\perp,$$

where $F = \lambda U.[\![A^\perp]\!]\eta[\alpha \mapsto U]$, $U = [\![B]\!]\eta$. (It is just at this point in the proof that second-order comprehension is used.) Hence $\sigma(P\bar{Q}) \in \forall(F)^\perp = ([\![(\exists\alpha.A)^\perp]\!]\eta)^\perp$, so $\mathtt{Cut}(\sigma(P\bar{Q}), Q)\Downarrow$. But $\mathtt{Cut}(\sigma(P\bar{Q}), Q) \Rightarrow^* P\bar{Q}Q$, so $P\bar{Q}Q\Downarrow$. ∎

As an immediate consequence of the Realizability Theorem, we get

Theorem 7.18 (Convergence) $\mathsf{PE}_2 \vdash \Theta; \bar{t} : \Gamma \implies \Theta; \bar{t}\Downarrow$.

Proof. By the Realizability Theorem,

$$\mathsf{PE}_2 \vdash \Theta; \bar{t} : \Gamma \implies \models \Theta; \bar{t} : \Gamma.$$

Now choose $\eta \in \mathsf{TEnv}$, $\bar{Q} \in [\![\Gamma^\perp]\!]\eta$, and conclude that $P\bar{Q}\Downarrow$, which implies $P\Downarrow$ by Lemma 7.13. ∎

7.3 Canonical vs. Cut-free

We would expect that the distinction between canonical and cut-free arises in practice only because of the lazy types: we do not fully evaluate proofs of lazy types in advance of the information telling us which arm of a case statement is to be evaluated (&), or how many copies are required (!). Our task in this subsection is to turn this expectation into a theorem. This will also provide a nice illustration of the use of acyclicity, which guarantees "deadlock freedom".

Proposition 7.19 *If* $\mathsf{PE}_2 \vdash \Theta; \bar{t} : \Gamma$, $\Theta; \bar{t}\Downarrow\Xi; \bar{u}$, *and* Γ *does not contain any occurrences of* &, ! *or* $\exists$, *then* $\mathrm{P}\mathcal{N}(\bar{u}) = \varnothing$.

Proof. We sketch the proof only, as the result is not surprising, and the details would be quite lengthy. The idea is to introduce a typed version of PE_2—in the "Church style" (cf. [BH90])—in which the proof expressions have embedded types; in particular, names are decorated with types. The quantifier rules are interpreted non-trivially in this version:

$$\frac{\vdash \Theta, \Xi; \bar{t} : \Gamma, t : A}{\vdash \Theta; \bar{t} : \Gamma, \Lambda\alpha.(\Xi; t) : \forall\alpha.A} \quad (*) \quad \frac{\vdash \Theta; \bar{t} : \Gamma, t : A[B/\alpha}{\vdash \Theta; \bar{t} : \Gamma, \langle B, t\rangle : \exists\alpha.A}$$

where α does not occur free in Θ in the *For All* rule. The rules of the Linear CHAM are modified in a fairly obvious fashion, the most interesting case being the Communication rule, which becomes:

$$t \perp x^A, x^{A^\perp} \perp u \longrightarrow t \perp u$$

(Note that $t : A^{\perp}$, $u : A$, so the new coequation is still well-typed.) Also, there is one new reaction rule, for the quantifier forms:

$$\Lambda\alpha.(\Theta; t) \perp \langle B, u\rangle \longrightarrow \Theta[B/\alpha], t[B/\alpha] \perp u$$

The previous results of this section can be transferred to this typed version of the system. Now consider the following property of typed proof expressions P:

For all $t : A$ occurring in P, if A does not contain any occurrences of $\&$, ! or $\exists$, then $\mathrm{P}\mathcal{N}(t) = \varnothing$.

This property is easily checked to hold for derivable proof expressions, and to be preserved under the transition relation. Stripping off the types again, this establishes the Proposition. ∎

The appearance of $\exists$ in the hypotheses of this proposition should not be a surprise. $\exists$ provides "information hiding" (cf. [MP85]), and the information hidden may well include the use of lazy types.

Theorem 7.20 *If* $\mathsf{PE}_2 \vdash \Theta; \bar{t} : \Gamma$, Γ *does not contain any occurrences of* $\&$, ! *or* $\exists$, *and* $\Theta; \bar{t} \Downarrow P$, *then* P *is cut-free.*

Proof. Suppose for a contradiction that $P = \Xi; \bar{u}$ is not cut-free, *i.e.* that Ξ contains some coequation $x \perp t$. By linearity, there must be another occurrence of x in P, either elsewhere in Ξ, or in the main body $\bar{u}$. However, an active occurrence in $\bar{u}$ would contradict P canonical, while a passive occurrence is precluded by the assumption on Γ and Proposition 7.19. So the other occurrence must be in Ξ, and we have

$$t \smile x \frown u \smile y \frown \cdots$$

(note that we cannot have $u = x$, since P is canonical). Using the same reasoning, we can continue this path indefinitely, but since $\mathcal{G}(P)$ is finite, we must eventually get a cycle, contradicting the acyclicity of P. ∎

We give some examples to illustrate the use of this Theorem. Firstly, note that the unique cut-free proof of $\mathbf{1}$ is the axiom

$$\frac{}{\vdash ; * : \mathbf{1}}$$

and the only cut-free proofs of $\mathbf{1} \oplus \mathbf{1}$ are

$$\frac{\dfrac{}{\vdash ; * : \mathbf{1}}}{\vdash ; \mathtt{inl}(*) : \mathbf{1} \oplus \mathbf{1}} \qquad \frac{\dfrac{}{\vdash ; * : \mathbf{1}}}{\vdash ; \mathtt{inr}(*) : \mathbf{1} \oplus \mathbf{1}}$$

So we can think of $\mathbf{1} \oplus \mathbf{1}$ as a type Bool of *booleans*, taking say $\mathsf{tt} = ; \mathtt{inl}(*)$, $\mathsf{ff} = ; \mathtt{inr}(*)$. Any proof of type Bool (containing cuts) will yield a proof expression P such that $P \Downarrow Q$ implies Q cut-free, *i.e.* $Q = \mathsf{tt}$ or $Q = \mathsf{ff}$. So any computation of type Bool will yield an "honest to God" explicit boolean value.

If we now consider the standard representation in System F of the natural numbers by the type of Church numerals

$$\forall \alpha. ((\alpha \supset \alpha) \supset (\alpha \supset \alpha))$$

its translation into CLL will yield the type

$$\forall\alpha.\,(\,?(\,!\alpha \otimes \alpha^{\perp}) \invamp ?\alpha^{\perp} \invamp \alpha)$$

which contains !, so computations of this type will not in general yield cut-free results (normal forms). However, as pointed out in [Gir87], the only essential use of the exponential here is in iterating the "successor" function, so the natural numbers can in fact be defined in Linear Logic by the type

$$\mathtt{nat} = \forall\alpha.\,(\,!(\alpha \multimap \alpha) \multimap \alpha \multimap \alpha) = \forall\alpha.\,(\,?(\alpha \otimes \alpha^{\perp}) \invamp \alpha^{\perp} \invamp \alpha).$$

Theorem 7.20 *does* apply to this type; so we get cut-free proofs of type `nat`. The same idea can be applied to all the usual first-order algebraic data-types (lists, trees, ...); *our operational semantics yields fully evaluated cut-free proofs at all observable types.*

8 Implementation Notes

Firstly, we consider a sequential implementation of proof expressions. The main point is to show that the "magical mixing" of the Linear CHAM can in fact be implemented in a simple and efficient way. The implementation uses two data structures: a stack of coequations to be processed, and a *name queue*. The internal representation of names is as pointers to entries in this queue. Such an entry can be in one of two states: *empty* or *pending*. Initially, the entry for each name is empty. When a coequation $x \perp t$ is encountered on the coequation stack, the name queue entry pointed to by x is inspected. If it is empty, the state is changed to pending, and t is stored there. If it is pending, with some term u stored there, then the entry is deleted from the name queue (and returned to the free store, since there can be no other references to it), and the coequation $t \perp u$ pushed on the coequation stack. This implements the Communication rule. All the other reaction rules can be implemented very straightforwardly (note that the Copy rule requires the creation of new entries in the name queue).

It is worth pausing at this point to note that this very simple implementation, with no closures, environments or garbage collection, supports a very powerful higher order functional programming language (System F), and both lazy and eager modes of evaluation.

Now we consider the prospects for a parallel implementation. We assume the following architecture: a network of *agents*, *i.e.* processor-memory pairs, each capable of sending data to any other. The coequations are distributed over this network. Each agent executes an instance of the sequential interpreter described above, with a coequation stack and name queue held in its local memory. However, names x are now represented by pairs (i, l), where i is the (network-wide unique) identifier of the agent on whose name queue the entry for x is held, and l the location in the local memory of agent i for that entry. So names—and only names—are represented by global addresses. The only difference to the way each interpreter works is that when a coequation $x \perp t$ is encountered, with $x = (i, l)$, it must be sent to agent i for further processing (and, of course, incoming requests of this kind must be handled).

As in the Sherlock Holmes story, the main point about this implementation scheme is the dog that *didn't* bark. In particular:

- The *only* requirement for inter-agent sharing and synchronization arises from the handling of names as described above. This seems much simpler than recent proposals e.g. for architectures to perform parallel graph reduction [JJK88].

- There are no centralized resources. Indeed, there are no distinguished nodes in the network. Each agent runs an instance of the same program.
- The elimination of garbage collection is probably of much greater value in a parallel implementation than a sequential one.
- There are good prospects for applying static analysis techniques to obtain "good" mappings of sets of coequations onto the agent network, e.g. to optimize "locality of reference", so that most of the time when a name (i, l) is encountered in an agent j, i is "near" j, thus reducing the cost of communication between them. In particular, by linearity channels are used exactly once, so to determine the "strength" of the connection between terms t and u, we can simply count $\mathcal{N}(t) \cap \mathcal{N}(u)$. (Contrast this with `occam` [LTD88] or CSP [Hoa85], where what matters is not the number of channels two processes have in common, but the number of times the channels will be used.) A mapping algorithm could then attempt to optimize locality of reference by making the distance between two coequations inversely proportional to the strength of their connection.

Of course, this is far from the whole story. One significant point is the need for *load-balancing*, *i.e.* maintaining an even loading of coequations over the network. However, even here the structure of the Linear CHAM offers some support. The only rule which *increases* the size of the proof expression is the Copy rule, so this provides a natural place for load-balancing to be performed. Again, one might hope to use static analysis techniques to "compile in" load balancing, keeping a good trade-off with locality of reference.

The remarks in this Section are speculative; detailed work is needed to evaluate the ideas. However, I believe that they do have genuine promise. The key point is that the closer marriage of mathematical form with computational content in Linear Logic seems to offer much better possibilities for efficient implementations to arise naturally, from the logical structure of the language.

References

[Abr90] S. Abramsky. Computational interpretations of linear logic. Technical Report DOC 90/20, Imperial College, Department of Computing, October 1990.

[AH87] S. Abramsky and C. L. Hankin, editors. *Abstract Interpretation for Declarative Languages*. Ellis Horwood, 1987.

[Bar84] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, revised edition, 1984.

[BB90] G. Berry and G. Boudol. The Chemical Abstract Machine. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 81–94, 1990.

[BH90] H. Barendregt and K. Hemerik. Types in lambda calculi and programming languages. In *Proceedings of ESOP 90*, 1990.

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

[Coh81] P. M. Cohn. *Universal Algebra*. D. Reidel, 1981.

[DM82] L. Damas and R. Milner. Principal type schemes for functional programs. In *Conference Record of the Ninth Annual ACM Symposium on the Principles of Programming Languages*, pages 207–212, 1982.

[FH88] A. J. Field and P. G. Harrison. *Functional Programming*. Addison Wesley, 1988.

[Gal90] J. Gallier. On Girard's "Candidats de Reductibilité". In P.-G. Odifreddi, editor, *Logic and Computer Science*. North Holland, 1990.

[Gir72] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur*. PhD thesis, University of Paris VII, 1972.

[Gir87] J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[Gir89a] J.-Y. Girard. Geometry of interaction 1: Interpretation of System F. In R. Ferro et al., editor, *Logic Colloquium 88*. North Holland, 1989.

[Gir89b] J.-Y. Girard. Towards a geometry of interaction. In J. W. Gray and Andre Scedrov, editors, *Categories in Computer Science and Logic*, volume 92 of *Contemporary Mathematics*, pages 69–108. American Mathematical Society, 1989.

[GJ86] H. Ganzinger and N. D. Jones, editors. *Programs as Data Objects*, volume 217 of *Lecture Notes in Computer Science*. Springer-Verlag, 1986.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

[GSS90] J.-Y. Girard, A. Scedrov, and P. J. Scott. Bounded linear logic. In S. R. Buss and P. J. Scott, editors, *Proc. Math. Sci. Institute Workshop on Feasible Mathematics*. Birkhauser, 1990.

[Hen80] P. Henderson. *Functional Programming: Applications and Implementation*. Prentice Hall, 1980.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Hol88] Sören Holmström. Linear functional programming. In T. Johnsson, S. Peyton Jones, and K. Karlsson, editors, *Proceedings of the Workshop on Implementation of Lazy Functional Languages*, pages 13–32, 1988.

[Hue90] G. Huet, editor. *Logical Foundations of Functional Programming*. Addison Wesley, 1990.

[HW88] P. Hudak and P. Wadler. Report on the functional programming language Haskell. Technical Report YALEU/DCS/RR666, Department of Computer Science, Yale University, November 1988.

[JJK88] T. Johnsson, S. Peyton Jones, and K. Karlsson, editors. *Proceedings of the Workshop on Implementation of Lazy Functional Languages*. Programming Methodology Group, Chalmers University, 1988.

[Kah87] G. Kahn. Natural semantics. In *Proc. Symposium on Theoretical Aspects of Computer Science*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 1987.

[Laf88] Y. Lafont. The Linear Abstract Machine. *Theoretical Computer Science*, 59(1,2):157–180, 1988.

[Lan64] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.

[Lan65] P. J. Landin. A correspondence between ALGOL 60 and Church's lambda notation. *Communications of the ACM*, 8:89–101,158–165, 1965.

[LTD88] INMOS LTD. *occam 2 Reference Manual*. Prentice Hall, 1988.

[MC88] A. Meyer and S. Cosmodakis. Semantical Paradigms. In *Proc. Third Annual Symposium on Logic in Computer Science*, pages 236–255. Computer Society Press, 1988.

[McC63] J. McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–69. North Holland, 1963.

[Mil80] R. Milner. *A Calculus for Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1980.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

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

[ML84] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Naples, 1984.

[MP85] J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 37–51, 1985.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Myc80] A. Mycroft. The theory and practice of transforming call-by-need into call-by-value. In B. Robinet, editor, *International Symposium on Programming*, volume 83 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[PJ87] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[Plo75] G. D. Plotkin. Call-by-name, call-by-value and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[Plo85] G. D. Plotkin. Lectures on predomains and partial functions. Notes for a course given at the Center for the Study of Language and Information, Stanford, 1985.

[Rey85] J. C. Reynolds. Three approaches to type structure. In H. Ehrig, C. Floyd, M. Nivat, and J. Thatcher, editors, *Mathematical Foundations of Software Development*, volume 185 of *Lecture Notes in Computer Science*, pages 97–138. Springer-Verlag, 1985.

[San89] D. Sands. Complexity analysis for a lazy higher order language. In *Proceedings of the Second Glasgow Workshop on Functional Programming*, 1989.

[Sol89] U. Solitro. A typed calculus based on a fragment of linear logic. *Theoretical Computer Science*, 68:333–342, 1989.

[Tur85] D. A. Turner. Miranda—a non-strict functional language with polymorphic types. In J.P. Jouannaud, editor, *Functional programming languages and Computer Architectures*, volume 201 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[Tur90] D. A. Turner, editor. *Research Topics in Functional Programming*. Addison Wesley, 1990.

[Val89] S. Valentini. The judgement calculus for intuitionistic linear logic: proof theory and semantics. Technical Report 64/89, University of Milan, Dip. Scienze dell'Informazione, 1989.

[Wad90] P. Wadler. Linear types can change the world! In M. Broy and C. B. Jones, editors, *Programming Concepts and Methods*. North Holland, 1990.