The barrier of objects:
From dynamical systems to bounded organizations

Appendices

Walter Fontana

Theoretical Chemistry
University of Vienna
Währingerstraße 17
A-1090 Vienna, Austria
and
International Institute for Applied
Systems Analysis (IIASA)
Schloßplatz 1
A-2361 Laxenburg, Austria

walter@santafe.edu

Leo W. Buss

Department of Biology
and
Department of Geology and
Geophysics
Yale University
New Haven, CT 06520-8104, USA

leo.buss@yale.edu

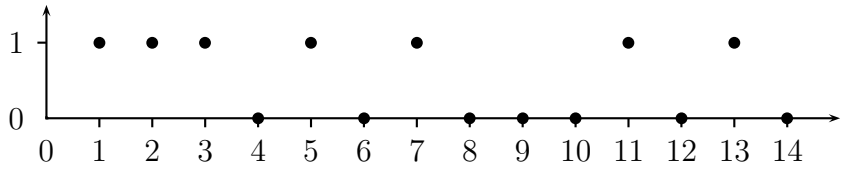
Contents

Appendix

A λ-calculus for tourists

1 Conceptual

The modern view of "function" is that of an arbitrary *set* of pairs - (argument/value) - whose first element is unique. The entire graph of the function is taken to be available at once, as a given, with no "cost" for its generation. For example, you are given the following graph, in which a prime is paired with 1 and a non-prime with 0, as fully completed to the infinite right.



In contrast, an older view emphasizes a function as a *rule of computation*, i.e., as a process of symbolic manipulation that produces a value when *applied* to an argument.

> Given a number n try dividing it by 2 and by each odd integer up to the biggest integer which is smaller or equal to the square root of n. If none of the trial divisors divides n, return 1, otherwise return 0.

The point is that one trades the instant random access to a look-up table which is so big as not to fit into the universe, with a procedure which fits into your pocket, but at the "cost" that it must be carried out. Procedures have to be expressed in some formal language. The requirement of a language, in turn, entails a refinement of the world into "*behavior*" and "*that which behaves.*" The former remains the still "ethereal" graph, while the latter is an "object", that is, a symbolic structure shaped by some sort of syntax that can be subject to new kinds of manipulation. This procedural or computational paradigm lies at the base of our project. It takes seriously the fact that in the physical world one never manipulates behavior, only the objects that behave.

Of course, the above example is no more than a joke; we would have to be equally explicit about what we mean by "divide", "2", "each", "integer", "biggest", "smaller", "or", "equal", "square root". It is no joke, however, that the λ-calculus invented by A. Church in the 1930's [2, 3] (following a trail pioneered by M. Schönfinkel [23]) does just that.

2 Instant Syntax and Semantics

The universe of λ-objects consists of *terms* with a particular structure. This structure is defined inductively, starting from "atoms".

TERMS

λ-calculus	Informal Interpretation	Tourist Notation
x	x is an atomic name taken from some available name space.	x
$\lambda x.A$	Make the λ-term A into a function of x. This is done by turning x in A from being a literal "x" into something replaceable, a variable. Since x now holds a place, it's particular name has lost significance. All that matters is the link to its corresponding λ marker, indicated by writing λx. One says, x has been *abstracted*.	Given the expression $x^2 + 4x + y$ we turn it into a function in x by declaring x to be a variable: $x \to x^2 + 4x + y$.
$(A)B$	If A and B are λ-terms, then their juxtaposition $(A)B$ denotes the *application* of A to B	Given the function $f : x \to x^2 + 4x + y$ and given 6, we can speak of $f(6)$.

So far we have only terms. Let's have some action.

REDUCTION

λ-calculus	Informal Interpretation
$(\lambda x.A)B \to A[x := B]$	When applying a function $\lambda x.A$ to an argument B, we proceed by literally *substituting* for the placeholder x the argument B. The fact that the place(s) held by x has (have) been filled is documented by removing the placeholder declaration λx. In the above example, $f(6) \to 60 + y$.

That's all there is.

Some details on reduction

There are two technical details one should be aware of. (i) The abstractor λ has a *scope* (like an ordinary integral sign), i.e., in $(\lambda x.A)B$ the binding influence of λ stops at A, and does not continue into B. (ii) The idea behind substitution is to replace equals by equals, meaning that the behavior of $(\lambda x.A)B$ should be the same as that of $A[x := B]$. Unbound literals must, therefore, never get bound during substitution. This one, for example, is illegal: $(\lambda x.\lambda y.(x)y)\lambda z.\boxed{\text{y}} \rightarrow \lambda y.(\lambda z.\boxed{\text{y}})y$. The boxed y has been captured by a λy. To perform the substitution safely, one has to rename the bound y into, say, w. We skip the formalization of these statements.

In a reduction step an "application" annihilates an "abstraction". **Normalization** is the process in which all the reductions that are possible within a term are carried out. At that point a term is said to be in **normal form**. The normal form is unique (if it exists - see below). This property of the reduction relation on λ-terms is called **confluence**. The reflexive, symmetric and transitive closure of the reduction relation is an **equivalence** relation on terms, i.e., two λ-terms are equivalent, if they have the same normal form.

Within the scope of our chemical metaphor a normal form is the analogue of a stable molecular form. The application of one (abstraction) term to another is analogous to a reactive encounter. Such a configuration is (usually) not a normal form, and is "stabilized" by normalization - the λ-analogue of a "reaction path". In this view of chemistry, "(free) energy" is that which causes molecular transition states to stabilize into products and is captured by our *requirement* that terms be in normal form. Other aspects of energy, such as differential rate constants, are not captured in Minimal Chemistry Zero (but see section 2.3.3 for MC2).

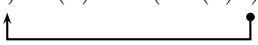
Two examples: $A \stackrel{\text{def}}{=\!=} \lambda x.((x)\lambda y.y)x$ is in normal form; so is $B \stackrel{\text{def}}{=\!=} \lambda u.(u)\lambda v.v$, but not $(A)B \stackrel{\text{def}}{=\!=} (\lambda x.((x)\lambda y.y)x)\lambda u.(u)\lambda v.v$. We normalize (underlining the subterms being reduced at each step):

$$\underline{(\lambda x.((x)\lambda y.y)x)\lambda u.(u)\lambda v.v} \rightarrow (\underline{(\lambda u.(u)\lambda v.v)\lambda y.y})\lambda u.(u)\lambda v.v \rightarrow$$
$$(\underline{(\lambda y.y)\lambda v.v})\lambda u.(u)\lambda v.v \rightarrow \underline{(\lambda v.v)\lambda u.(u)\lambda v.v} \rightarrow \lambda u.(u)\lambda v.v$$

In this case B is a fixed point of A.

Not every term has a normal form. For example, here's a term which is not normalizable:

$$(\lambda x.(x)x)\lambda x.(x)x \rightarrow (\lambda x.(x)x)\lambda x.(x)x$$



A calculus in which every term has a normal form is called **strongly normalizing**. Our model utilizes only terms with a normal form. Indeed, we even discard those terms which fail to normalize within some specified limits.

It is worth pointing out that in this version of λ-calculus *every term can be applied to every term.* Thus, any term can be filled into the place held by a variable. Or,

by means of slogan, there's no syntactical distinction between function and data. *Everything in λ is, in some sense, a function.* This is a very powerful concept. You may have wondered where the "numbers" are, or where the familiar "addition" has gone. No such operations are given. Everything has to be constructed just with what we've got, i.e., variables, abstraction and application. (It can be done.) The intriguing feature of λ-calculus is in forcing one to realize that something is, for example, a numeral (a representation of a number), if it *behaves* - via application and reduction - like a number. Something is a numeral when it is a member of a sequence of distinct terms that have a successor function, and there exists a test for a distinguished element "zero". Likewise, something is an "addition", if it *behaves* like an addition *in relation to some system of numerals.* Change the system of numerals and the object that behaved like an addition doesn't anymore. Here's an example of a numeral system:

$$\begin{array}{rcl}
\lambda f.\lambda x.x & \text{corresponds to} & 0 \\
\lambda f.\lambda x.(f)x & \text{corresponds to} & 1 \\
\vdots & \vdots & \vdots \\
\lambda f.\lambda x.\underbrace{(f)...(f)}_{n \text{ times}}x & \text{corresponds to} & n \\
\vdots & \vdots & \vdots
\end{array}$$

Relative to it, the addition operation becomes $+ \stackrel{\text{def}}{=} \lambda m.\lambda n.\lambda f.\lambda x.((m)f)((n)f)x$. The normalization of $3 + 2$ is displayed in table A1. It looks slightly frightening, but most of the 54 intermediate steps are just necessary rearrangements in preparation for reductions.

Note then the "relativity" of the system – one defines *what behaves* and generates behaviors. The power of the system derives from just this flexibility. "Behavior," in the example above, was treated as a device which sends numerals into numerals. This frame need not be maintained. In fact, nothing prevents us from taking the addition function, $+$, and apply it to something else than a numeral - to itself, say. So here is $(+)+$:

$$\begin{array}{rl}
 & (\lambda m.\lambda n.\lambda f.\lambda x.((m)f)((n)f)x)\lambda m.\lambda n.\lambda f.\lambda x.((m)f)((n)f)x \rightarrow \cdots \\
\cdots \rightarrow & \lambda n.\lambda f.\lambda x.\lambda u.\lambda v.((f)u)((((n)f)x)u)v
\end{array}$$

(Where the variable names u and v come from renaming during normalization.)

Can we still meaningfully say that a "function" has been computed? No. In full λ-calculus the notion of a "function" is better replaced by the more vague notion of an "operator". This point is crucial for our usage of λ-calculus. Indeed, *the individual interactions in our reactor by and large don't compute anything, they solely rearrange symbolic structures.* The interpretation of their "behavior"

is framed by the algebraic and kinetic properties of the organization that their actions participate in maintaining. The same can be said of chemistry.

$$((\underbrace{\lambda m.\lambda n.\lambda f.\lambda x.((m)f)((n)f)x}_{+})\ \underbrace{\lambda f.\lambda x.(f)(f)(f)x}_{3})\ \underbrace{\lambda f.\lambda x.(f)(f)x}_{2}$$

```
 1 (λn.(λm.λf.λx.((m)f)((n)f)x)λf.λx.(f)(f)(f)x)λf.λx.(f)(f)x
 2 (λn.λf.(λm.λx.((m)f)((n)f)x)λf.λx.(f)(f)(f)x)λf.λx.(f)(f)x
 3 (λn.λf.λx.(λm.((m)f)((n)f)x)λf.λx.(f)(f)(f)x)λf.λx.(f)(f)x
 4 (λn.λf.λx.((λm.(m)f)λf.λx.(f)(f)(f)x)(λm.((n)f)x)λf.λx.(f)(f)(f)x)λf.λx.(f)(f)x
 5 λf.(λn.λx.((λm.(m)f)λf.λx.(f)(f)(f)x)(λm.((n)f)x)λf.λx.(f)(f)(f)x)λf.λx.(f)(f)x
 6 λf.λx.(λn.((λm.(m)f)λf.λx.(f)(f)(f)x)(λm.((n)f)x)λf.λx.(f)(f)(f)x)λf.λx.(f)(f)x
 7 λf.λx.((λm.(m)f)λf.λx.(f)(f)(f)x)(λn.(λm.((n)f)x)λf.λx.(f)(f)(f)x)λf.λx.(f)(f)x
 8 λf.λx.(((λm.m)λf.λx.(f)(f)(f)x)(λm.f)λf.λx.(f)(f)(f)x)(λn.(λm.((n)f)x)λf.λx.(f)(f)(f)x)λf.λx.(f)(f)x
 9 λf.λx.((λf.λx.(f)(f)(f)x)(λm.f)λf.λx.(f)(f)(f)x)(λn.(λm.((n)f)x)λf.λx.(f)(f)(f)x)λf.λx.(f)(f)x
10 λf.λx.(λx.(λf.(f)(f)(f)x)(λm.f)λf.λx.(f)(f)(f)x)(λn.(λm.((n)f)x)λf.λx.(f)(f)(f)x)λf.λx.(f)(f)x
 ⋮                                                                                    ⋮
25 λf.λx.(f)(f)(λy1.(λx.(y1)x)(λn.(λm.((n)f)x)λf.λx.(f)(f)(f)x)λf.λx.(f)(f)x)f
26 λf.λx.(f)(f)((λy1.λx.(y1)x)f)(λy1.(λn.(λm.((n)f)x)λf.λx.(f)(f)(f)x)λf.λx.(f)(f)x)f
27 λf.λx.(f)(f)(λx.(λy1.(y1)x)f)(λy1.(λn.(λm.((n)f)x)λf.λx.(f)(f)(f)x)λf.λx.(f)(f)x)f
28 λf.λx.(f)(f)(λx.((λy1.y1)f)(λy1.x)f)(λy1.(λn.(λm.((n)f)x)λf.λx.(f)(f)(f)x)λf.λx.(f)(f)x)f
29 λf.λx.(f)(f)((λy1.y1)f)(λx.(λy1.x)f)(λy1.(λn.(λm.((n)f)x)λf.λx.(f)(f)(f)x)λf.λx.(f)(f)x)f
30 λf.λx.(f)(f)(f)(λx.(λy1.x)f)(λy1.(λn.(λm.((n)f)x)λf.λx.(f)(f)(f)x)λf.λx.(f)(f)x)f
 ⋮                                                                                    ⋮
43 λf.λx.(f)(f)(f)(λx.((λf.f)(λn.f)λf.λx.(f)(f)x)(λf.(f)x)(λn.f)λf.λx.(f)(f)x)(λn.x)λf.λx.(f)(f)x
44 λf.λx.(f)(f)((λf.f)(λn.f)λf.λx.(f)(f)x)(λx.(λf.(f)x)(λn.f)λf.λx.(f)(f)x)(λn.x)λf.λx.(f)(f)x
45 λf.λx.(f)(f)(f)((λn.f)λf.λx.(f)(f)x)(λx.(λf.(f)x)(λn.f)λf.λx.(f)(f)x)(λn.x)λf.λx.(f)(f)x
46 λf.λx.(f)(f)(f)(f)(λx.(λf.(f)x)f)(λn.x)λf.λx.(f)(f)x
47 λf.λx.(f)(f)(f)(f)((λx.λf.(f)x)(λn.x)λf.λx.(f)(f)x)(λx.f)(λn.x)λf.λx.(f)(f)x
48 λf.λx.(f)(f)(f)(f)(λf.(λx.(f)x)(λn.x)λf.λx.(f)(f)x)(λx.f)(λn.x)λf.λx.(f)(f)x
49 λf.λx.(f)(f)(f)(f)(λf.(f)(λx.x)(λn.x)λf.λx.(f)(f)x)(λx.f)(λn.x)λf.λx.(f)(f)x
50 λf.λx.(f)(f)(f)(f)((λf.f)(λx.f)(λn.x)λf.λx.(f)(f)x)(λf.(λx.x)(λn.x)λf.λx.(f)(f)x)(λx.f)(λn.x)λf.λx.(f)(f)x
51 λf.λx.(f)(f)(f)(f)((λx.f)(λn.x)λf.λx.(f)(f)x)(λf.(λx.x)(λn.x)λf.λx.(f)(f)x)(λx.f)(λn.x)λf.λx.(f)(f)x
52 λf.λx.(f)(f)(f)(f)(f)(λf.(λx.x)(λn.x)λf.λx.(f)(f)x)f
53 λf.λx.(f)(f)(f)(f)(λx.x)(λn.x)λf.λx.(f)(f)x
54 λf.λx.(f)(f)(f)(f)(f)(λn.x)λf.λx.(f)(f)x
```

$$\overbrace{\lambda f.\lambda x.(f)(f)(f)(f)(f)x}^{5}$$

Table A1: $3 + 2 = 5$ in λ-calculus.

λ-calculus is a theory of equality based on substitution. In a more specialized sense, λ-calculus is a general theory of functions. Having served as a template for LISP, it inspired the "functional style" of programming. Moreover, as detailed in text, λ-calculus has served as a tool in constructive proof-theory and, accordingly, in mechanizing parts of logic (see section 1). This makes λ-calculus an almost obligate check point when introducing new paradigms of computation to which properties such as termination, confluence, normalization, or substitution are

central. *In the computational sciences these ingredients of λ-calculus play a role comparable to that of the fundamental principles of physics* [13]. Good introductions are Hankin [9] or Lalement [13], an encyclopedic treatment for aficionados with a good pair of shoes is Barendregt [1].

3 Beyond λ

The limits of λ-calculus are found in its inherently sequential paradigm. This is reflected by its non-commutative basic mode of interaction, application. What would be a commutative analogue? This question leads one to parallelism, or more precisely, *concurrency.* In contrast to sequentiality, the concurrent paradigm of computation considers a system of many heterogenously behaving independent entities that "interact" with one another (usually asynchronously). The proper technical word for interaction in such a setting is *communication* and the entities are called *processes.* The theory of communication and concurrency is among the most exciting and challenging frontiers in today's computational sciences. This is obviously not the place for a tutorial in concurrency; some places to start are [11, 16, 18]. Here we paint with a broad brush just some of the issues at stake so as to situate our work relative to it.

The transition from the concept of "function" to that of a "process" is illustrated by means of an example due to Robin Milner [18]. Consider the behavior of the following program, A, where "`:=`" means an assignment:

```
1: x := 2;          (assign 2 to x)
2: y := x+3;        (assign to y the content of x plus three)
3: print y;
```

Clearly, the program A will print `5`. Suppose now that there is a further concurrently running program B that has access to the memory location referred to by `x` in A. Such a program can alter the value at `x` after A has executed its first statement and before it executes the second. As a consequence, the observation of A does not yield a specific result anymore; it may print anything, depending on the behavior of B. This is the kind of situation that the theme of "communication and concurrency" is roughly about. The concept of "process" emphasizes behavior primarily as the ability to communicate at various points in time [10] rather than a computational activity. The issue, as emphasized in [21], is one of an apparent duality of time and information.

What is being communicated? The simplest kind of communication is a synchronization between two processes, i.e. a "handshake". One process pauses until it receives a signal from another upon which it resumes its behavior [15]. The

next order of communication involves the sending and receiving of port names themselves. This yields a system where "pointers" are passed around, thereby changing the communication topology of the system over time [19, 20]. At the next order whole processes rather than their address can be communicated [22]. This in turn raises the issues of "access" and "privacy". On the formal side the challenge is to find "calculi" which enable to reason about various notions of processes and their equivalence, in analogy to what λ-calculus does for the sequential realm.

Communication occurs between *ports* of processes. Ports are named, and each name has a complement. For example: a and $\overline{a}$, where a may stand for an input port and $\overline{a}$ for an output port. Communication can only occur between ports that bear complementary names. This ensures commutativity of communication by definition.

π-calculus

One foundational attempt at mobile processes (systems with changing communication topology) is the π-calculus of Milner, Parrow and Walker [19, 20]. Just to give a glimpse of it for the purpose of comparison with λ-calculus, here's a π-expression:

$$\underbrace{\bar{x}y.A}_{1} \mid \underbrace{x(u).\bar{u}v.B}_{2} \mid \underbrace{y(z).C}_{3}$$

It denotes a soup of three processes, $1, 2, 3$, that co-exist independently. This concurrence is expressed by the operator $|$. The processes in the example are only partially specified, since A, B and C stand for further structure which we disregard. $\bar{x}y$ means "output the name y along channel x", while $x(u)$ means "receive a name along channel x and substitute that name for u in the remaining process" (this input prefix binds u much like a λ). In the above example a communication can occur between process 1 and process 2 along channel x, yielding:

$$\underbrace{A}_{1'} \mid \underbrace{\bar{y}v.B[u := y]}_{2'} \mid \underbrace{y(z).C}_{3}$$

The point is that, as a result of this event, process 2 (now $2'$) has obtained a port name that enables it to communicate with process 3:

$$\underbrace{A}_{1'} \mid \underbrace{B[u := y]}_{2''} \mid \underbrace{C[z := v]}_{3'}$$

Concurrent processes can occur embedded within a process, such as in $\bar{x}y.(A|B)$. Furthermore, there is a scoping operator ν which restricts the use of x to process A in $(\nu x)A$, and there is a choice operator $+$ behaving so that in $A + B$ a communication with A destroys B and vice versa. Finally, there is a replication operator $!A$ which permits process A to spin off further copies of itself allowing for recursion.

"|" might be a way to notate the concurrency of the λ-particles in our flow-reactor. As in λ-calculus, a "type"-discipline for π-calculus can be defined. For

the further development of our model, we are inclined towards the logic path to concurrency rather than π-calculus, as developed in the text. Readers wishing to further explore π-calculus are referred to [17].

The world of functions and the world of processes emphasize the halting problem differently. While termination is a *desideratum* for functions or algorithms, the opposite is typically true for processes. There one looks for conditions under which a community of processes is guaranteed never to dead-lock, as there are many situations where ongoing communication or interactivity is required. Examples include operating systems, whether in air traffic control systems, computer systems, mobile telephone networks, or...living and cognizing systems. *The focus on the absence of dead-lock shifts the attention from computation to organization.* This clearly locates concurrency very close to our project.

B Types for tourists

1 The chemistry of types

Types are a high-level statement about the *behavior* of objects. A conventional addition function, for example, has type $N \times N \to N$, meaning that it accepts pairs of integers, and returns integers. This is not sufficient to distinguish it from a subtraction function, but is enough to distinguish it from a function that adds "carriage returns" to a string of characters. The definition of a *type system* decides on how much about the actual behavior of an object is conveyed by its type. A type system also provides a procedure to infer the type of a compound object from the types of its components.

We first need a way to express types. The notation is inductive like the syntax of λ-calculus. We start by defining a set of atomic types, called *simple (or ground) types*, say $\mathcal{T} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \ldots\}$. From simple types we construct compound types with the help of *type constructors*. This is analogous to λ-calculus where compound terms are built from two *term constructors* - abstraction and application (see Appendix A). The choice of type constructors reflects the kind of actions one seeks to capture. For the sake of simplicity we consider here only one type constructor: the *function type* "$\to$". A type, then, is either

- a simple type: $\mathtt{s} \in \mathcal{T}$, or
- a function type: $\mathtt{s} \to \mathtt{t}$, where $\mathtt{s}$ and $\mathtt{t}$ are types.

The function type $\mathtt{s} \to \mathtt{t}$ denotes a mapping which accepts objects of type $\mathtt{s}$ and

returns objects of type `t`. Think of it as one kind of chemical bond ("$\rightarrow$"). It links together the *action(s)* of atoms (or groups of atoms).

The type system is coupled to the λ-calculus by means of *inference rules* based on the structure of λ-terms. Let's proceed intuitively at first: if a variable x in λ-calculus has type `s`, notated $x : \mathtt{s}$, and if the expression E has type `t`, notated $E : \mathtt{t}$, then the term $\lambda x.E$ has type $\mathtt{s} \rightarrow \mathtt{t}$, notated $\lambda x.E : \mathtt{s} \rightarrow \mathtt{t}$. What we have just made is a bond connecting action `s` with action `t`. The corresponding term - the abstraction $\lambda x.E$ - *is* the physical bond - as opposed to its type ($\rightarrow$) which indicates it's potential chemical activity. Indeed, a bond that can be made, can be broken. This holds for types as well. When the object $\lambda x.E$ whose action is $\mathtt{s} \rightarrow \mathtt{t}$ is brought into contact with an object $F : \mathtt{s}$, the bond is broken, and the action `t` is recovered: $(\lambda x.E)F : \mathtt{t}$. The corresponding normalized object can be shown to have the same type. The fact that the type doesn't change upon normalization indicates that *types do not compute results*; the computation is done by the λ-calculus mechanics. Types are just a statement about the possible reactions and results.

Summing up, "abstraction" makes bonds, "application" breaks bonds (of the "$\rightarrow$" kind). A bond works here like an "if-then" relation, since $\mathtt{s} \rightarrow \mathtt{t}$ specifies the conditions that have to be met to break and to release the "then" portion (in this case simply to encounter an object of type `s`).

2 Polymorphism

A function of type $\mathtt{s} \rightarrow \mathtt{t}$, where `s` and `t` are simple types, is *monomorphic*, because - in terms of our metaphor - it only has one shape: it recognizes only things of the shape `s`, and it returns things of the particular shape `t`. Seen this way there are infinitely many identity operations, $\lambda x.x$, with different types - such as: $\mathtt{a} \rightarrow \mathtt{a}$, $\mathtt{b} \rightarrow \mathtt{b}$, $(\mathtt{a} \rightarrow \mathtt{b}) \rightarrow (\mathtt{a} \rightarrow \mathtt{b})$, etc. - yet all do the same thing. In fact, these different types are but specific instances of a generic type $\alpha \rightarrow \alpha$, where α can be anything. Because it can be anything,we can treat it as a variable - a *type variable*. This is expressed as $\forall\alpha.\alpha \rightarrow \alpha$, also known as a *type scheme*. The introduction of type variables yields the concept of a *polymorphic type*. In contrast to a monomorphic operator, a polymorphic one can act on a variety of things with different shapes. For example, a function with type $\forall\alpha.(\alpha \rightarrow \mathtt{a}) \rightarrow (\mathtt{b} \rightarrow \alpha)$ can operate on any object which is an *instance* of the type $\forall\alpha.\alpha \rightarrow \mathtt{a}$, such as $\mathtt{c} \rightarrow \mathtt{a}$ or $\forall\beta.(\mathtt{c} \rightarrow \beta) \rightarrow \mathtt{a}$, but it cannot act on instances of $\forall\beta.\beta \rightarrow \mathtt{c}$. In the context of the chemical metaphor, polymorphism means that our abstract molecules can have different degrees of "specificity". Some could be "rigid" (monomorphic), others could be completely unspecific, and still others could cover the spectrum of specificity in between.

3 Type inference

Given the structure of a λ-term how do we infer its type? To begin with we assign type schemes to certain variables initially. This initial assignment, A, has the status of a boundary condition. It specifies our "chemistry". We will write the derivation of type σ for the expression P under the assumptions A as an inference: $A \vdash P : \sigma$ (read: "from A derive P of type σ"). Here's the complete set of rules for this game [13, 14], which we explain intuitively below:

$$\textbf{Taut}\text{ology} \qquad x : \sigma \in A \vdash x : \sigma$$

$$\textbf{Inst}\text{antiation} \qquad \frac{A \vdash e : \sigma}{A \vdash e : \sigma'} \ (\sigma > \sigma')$$

$$\textbf{Gen}\text{eralization} \qquad \frac{A \vdash e : \sigma}{A \vdash e : \forall\alpha.\sigma} \ (\alpha \text{ not free in } A)$$

$$\textbf{App}\text{lication} \qquad \frac{A \vdash e : \tau' \to \tau \qquad A \vdash e' : \tau'}{A \vdash (e)e' : \tau}$$

$$\textbf{Abs}\text{traction} \qquad \frac{A_x \cup \{x : \tau'\} \vdash e : \tau}{A \vdash \lambda x.e : \tau' \to \tau}$$

$$\textbf{Let} \qquad \frac{A \vdash e : \sigma \qquad A_x \cup \{x : \sigma\} \vdash e' : \tau}{A \vdash (\text{let } x = e \text{ in } e') : \tau}$$

The meaning of the **Taut**-rule is simply that a free variable has the type assigned to it in the boundary condition. If there is no assignment the expression x is not typable, and is barred from the universe.

The meaning of rule **App** is also clear. It is useful, however, to know how the rule is implemented, since it introduces an important concept. Suppose that we have an object e whose type has been established to be σ, and that we want to apply it to an object e' of type τ'. For this to be possible e must have a type of the form $\tau' \to \tau$ where τ stands for a generic unknown type of $(e)e'$ that needs to be determined. Hence, for the interaction $(e)e'$ to be possible e's established type σ and the required type $\tau' \to \tau$ must be made equal. This may be possible, since σ and τ' may contain type variables which can be made more specific in order to satisfy the equality. This means we must look for some type substitution T of the free variables in σ and in $\tau' \to \tau$ such that $T\sigma = T(\tau' \to \tau)$. T is called a *unifier*, and the procedure for finding T is called *unification*. It boils down to solving a set of equations. For details about how this procedure is carried out the reader is referred to any standard textbook on type theory. The point is that a

successful unification will end up with a particular τ, the desired type of $(e)e'$. If unification is not successful, then e cannot be applied to e', i.e., the interaction term $(e)e'$ does not exist.

It is clear now that a type system poses constraints on permissible λ-terms. For example, $\lambda x.(x)x$ is not any longer an element of the universe of objects, for it has no type. To type the subterm $(x)x$ we would have to first assume the generic type α for x, and then use rule **App** which requests that x be of type $\alpha \to \alpha$. But the equation $\alpha = \alpha \to \alpha$ is recursive and has no solution (in this type system).

Recall that we model a chemical reaction by the application of a function: $(\lambda x.E)F$. In **Let** the argument, F, is typed first, and then its type is assigned to the variable x when proceeding in the type synthesis of E. The **Let** rule allows for more general interactions than are otherwise permitted by **App**. We use **Let** to model a reaction.

Finally, the **Abs**-rule is used like this. If the variable x has a type assigned in the boundary condition A, τ' say, then we must use τ' in the derivation of the type for the function body e. If e is determined to have type τ, then the whole expression is $\tau' \to \tau$. On the other hand, if x has no assignment, then we are free to temporarily assume one. We assume a generic α, and proceed to derive the type for e. During this process the assumed type α may need to be specialized into $\tau'(< \alpha)$ to meet type constraints (viz unification). The resulting type for the overall expression is $\tau' \to \tau$, and the boundary condition A is left unchanged.

The other two rules, **Gen** and **Inst**, are used to generalize and to instantiate (specialize) a type in a particular way. Their explanation is not crucial at this level of discussion, and we skip it.

C Logic background

1 The Curry-Howard isomorphism

The Curry-Howard isomorphism [12] provides a rigorous link between the computational sciences and logic.

Recall the two rules for typing abstraction and application in λ-calculus (Appendix B):

$$\textbf{Abs} \quad \frac{A \cup \{x : \tau'\} \vdash e : \tau}{A \vdash \lambda x.e : \tau' \to \tau}$$

$$\textbf{App} \quad \frac{A \vdash e : \tau' \to \tau \qquad A \vdash e' : \tau'}{A \vdash (e)e' : \tau}$$

The notation is understood as a rule which links the two hypotheses (above the horizontal line) with a conclusion (below the line).

Take for instance the **App**lication rule and consider what remains when everything but the type information is erased:

$$\frac{\tau' \to \tau \qquad \tau'}{\tau}$$

Now read τ' and τ as *logical propositions*, and interpret the function arrow " $\to$" to mean logical *implication*. Then, *if* we know that τ' `implies` τ, *and if* we know that τ' actually holds, *then* we can conclude that τ holds. This logical inference is known as *modus ponens*. For example, empiricists routinely use this inference, reasoning that if an event τ is known (say, by prior experiment) to be contingent upon an event τ', and τ' is an empirical observation in a current experiment, then we observe τ in the current experiment.

In logic, "to know that a proposition holds" means to *prove* it, i.e. to stepwise assemble the proposition with the help of a "scaffold" (the proof). Made of special building blocks (rules of inference), a proof is a syntactical object just like a λ-term. Let us symbolize the text documenting the proof of a hypothesis with vertical dots (meaning, "insert the formal steps of proof here"):

$$\frac{\begin{array}{cc}\vdots & \vdots \\ \text{a proof} & \text{another proof} \\ \tau' \to \tau & \tau'\end{array}}{\tau}$$

Now, this could be seen as a proof of the proposition τ by combining a proof of $\tau' \to \tau$ and one of τ'. Indeed, *modus ponens* is a step in the construction of proofs. We could use the following scheme to name the steps in the proof:

$$\left.\begin{array}{c}\dfrac{\begin{array}{cc}\vdots & \vdots \\ e & e' \\ \tau' \to \tau & \tau'\end{array}}{\tau}\end{array}\right\} (e)e' \tag{1}$$

This, however, is precisely the rule for *typing* an "application" (here, $(e)e'$) in λ-calculus (**App**). From this point of view the rule for typing an **Abs**traction is interpreted as:

$$\left.\begin{array}{c}\dfrac{\begin{array}{c}[\tau'] \\ \vdots \\ e \\ \vdots \\ \tau\end{array}}{\tau' \to \tau}\end{array}\right\} \lambda x.e \tag{2}$$

This is meant to illustrate that a proof of τ, using the assumption τ', is a proof of $\tau' \to \tau$ where τ' has been removed from the list of assumptions. One says that τ' has been *discharged*[1]. In typed λ-calculus a free (unbound) x is, therefore, seen to stand for an assumption made (of type τ'). "Abstraction" - i.e., the binding of x as a variable - discharges that assumption, yielding a logical implication. Indeed, the object $\lambda x.e$ is a function. The function takes a (proof of the) proposition τ' and returns a (proof of the) proposition τ; hence it proves $\tau' \to \tau$.

The two rules (1) and (2) together define what "$\to$" means by stating how to eliminate and how to introduce it, respectively, from a logical formula. The implication connective is *introduced* by shuffling an assumption from the proof (the meta-language) into the logical formula (the object-language) which now keeps track of it[2]. The implication connective is *eliminated* by supplying a proof for the assumption expressed in the implication. Analogous rules of introduction and elimination exist for disjunction ($\vee$), conjunction ($\wedge$), and for the existential ($\exists$) and universal ($\forall$) quantifiers in the predicate case. This style of proof-presentation is called "natural deduction". We have introduced it here not for its direct utility in the chemical metaphor, but because it provides the simplest and most gentle connection between a logic and the typing of λ-terms.

Proof-normalization

How is the λ-calculus reduction process reflected in proof-theory? Reduction in λ-calculus is triggered by the application of a λ-expression to another: $(\lambda x.e)e'$ which becomes $e[x := e']$. The expression $(\lambda x.e)e'$ corresponds to a proof where the introduction of an implication (abstraction) is immediately followed by its elimination (application). Consider the case sketched below.

$$\dfrac{\dfrac{\begin{array}{c}[\tau'] \\ \vdots \\ e \\ \vdots \\ \tau\end{array}}{\tau' \to \tau} \qquad \begin{array}{c}\vdots \\ e' \\ \vdots \\ \tau'\end{array}}{\tau} \quad \longrightarrow \quad \begin{array}{c}\vdots \\ e' \\ \vdots \\ \tau' \\ \vdots \\ e \\ \vdots \\ \tau\end{array}$$

Here a proof is "normalized" by replacing copies of the derivation ending in τ' (i.e., right branch) for every discharged assumption τ' in the derivation on the

[1]This fact, however, must be recorded, for example by wrapping τ' into square brackets. This introduces a "non-local" action, i.e., an annotation at a location in the proof tree that is removed from the horizontal bar in (2) where things are currently happening. This will be avoided in another syntactical system introduced in Appendix 2.

[2]Stated differently, if we can prove β from assumption α (i.e. $\alpha \vdash \beta$), then we can prove $\alpha \to \beta$ from no assumption (i.e. $\vdash \alpha \to \beta$). This is the "deduction property" of logical consequence ($\vdash$).

left branch (i.e., top-most segment of left branch).

2 Sequent calculus

Whenever a theory has "objects" as its subject, notation becomes of paramount importance. The reason is that to a good extent the theory *is* the notation. Major perspectives on logic are, therefore, characterized by differring notational systems.

One of them is Gentzen's sequent calculus, introduced here and elsewhere [5, 4], as a natural bridge between the natural deduction systems/types (discussed above) and linear logic (a discussion of which follows). The judgements derived in sequent calculus are not individual formulae like in the previous case, but rather ensembles of formulae. These judgements are called "sequents", and are of the form $\Gamma \vdash \Delta$, where the turnstile indicates "logical consequence" and Δ and Γ are *multisets* of formulae, i.e. sets where some formula may occur more than once. The connection between sequent calculus and natural deduction is direct in the case where the right side of the turnstile contains a single formula: the proof of the judgement $\Gamma \vdash \psi$ corresponds to the deduction of ψ under the hypotheses Γ. The sequent notation is a device to keep track of all assumptions made and all formulae derived up to any point in the proof tree (collecting assumptions on the left and conclusions on the right). In contrast to "natural deduction" (Appendix 1, footnote 10), this makes the proof tree construction entirely local; what can be done at any stage depends solely on the end point of the tree.

The rules of the calculus - which we do not explain in detail here - taken together define the exact meaning of a sequent. The sequent

$$\phi_1, \ldots, \phi_n \vdash \psi_1, \ldots, \psi_m$$

means that the *conjunction* of the "antecedents" (ϕ_1 and ϕ_2 and ... and ϕ_n) *implies* the *disjunction* of the "succedents" (ψ_1 or ψ_2 or ... or ψ_m), i.e. $\phi_1 \wedge \ldots \wedge \phi_n \rightarrow \psi_1 \vee \ldots \vee \psi_m$ Stated in terms of a Boolean valuation $\mathcal{B}$ the sequent (3) says that "if all ϕ_i are true (under $\mathcal{B}$), then at least one ψ_i is true (under $\mathcal{B}$)".

Sequent calculus makes the symmetries and the algebraic properties of the logical connectives visible. In sequent calculus, like in natural deduction, proofs of judgements are built inductively by linking together other judgments through specific rules all the way up to assumptions or axioms. To provide the reader with the flavor of the system, we show the rules for implication ($\rightarrow$). Capital letters denote multisets of formulae, lower case letters denote individual formulae.

$$\text{left}\ \frac{\Gamma \vdash \phi, \Delta \qquad \Gamma', \psi \vdash \Delta'}{\Gamma, \Gamma', \phi \rightarrow \psi \vdash \Delta, \Delta'} \qquad\qquad \text{right}\ \frac{\Gamma, \phi \vdash \psi, \Delta}{\Gamma \vdash \phi \rightarrow \psi, \Delta} \tag{3}$$

For the purpose of an intuitive explanation, let us suppose that each judgement contains only one succedent, i.e., $\Delta = \emptyset$ and $\Delta' = \delta$. The left rule then means: (i) we know that under the stated conditions ϕ holds (left branch above the horizontal bar), and (ii) we know that under the stated conditions the assumption of ψ gives us δ (right branch above the bar). Clearly, if we can show that ϕ (what we have) implies ψ (what we lack), then δ would follow (under the stated conditions). This is tantamount to saying that we can derive δ from *assuming* the formula $\phi \rightarrow \psi$ in that context, and that is what appears below the bar.

Conversely, the right rule says that if - in a given context Γ - we can derive ψ by assuming ϕ, then we can derive from the context Γ alone that $\phi \rightarrow \psi$. This transfers the assumption from the meta-language of the proof to the object-language of the logical formula.

In sequent calculus logical connectives are introduced on the right and the left side of a judgement corresponding to introduction and elimination rules, respectively, in natural deduction (see Appendix 1 and the rules for implication (3)). Similar symmetric schemes hold for the other logical connectives. Sequent calculus needs no axioms beyond the rules of proof, since it allows the use of arbitrary identities at any time:

$$\frac{}{\phi \vdash \phi}$$

An important feature of the calculus is the existence of three "structural" rules to manipulate proofs. They do not introduce logical connectives on either side:

$$\frac{\Gamma \vdash \Delta}{\Gamma, \phi \vdash \Delta} \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \phi, \Delta} \qquad \text{weakening}$$

$$\frac{\Gamma, \phi, \phi \vdash \Delta}{\Gamma, \phi \vdash \Delta} \qquad \frac{\Gamma \vdash \phi, \phi, \Delta}{\Gamma \vdash \phi, \Delta} \qquad \text{contraction}$$

The weakening rule "weakens" a proof by introducing antecedents or succedents that are unnecessary. If the weakening of the succedent strikes you as peculiar, remember that the succedent of a judgement is the disjunction of its formulae. *In λ-calculus weakening corresponds to the declaration of a variable which never occurs in the body of the function*, e.g., $\lambda x.\lambda y.y$. Operationally it means "discarding an input", since the argument supplied for x evaporates. The contraction rule means that one can use as many copies of a formula as one wishes. In other words: there is no resource accounting in classical logic. *In λ-calculus this corresponds to "nonlinearity", i.e., to the multiple occurrences of the same variable within the body of a function*, e.g., $\lambda x.(x)x$.

The third structural rule is the so-called **cut**-rule:

$$\frac{\Gamma \vdash \phi, \Delta \qquad \Gamma', \phi \vdash \Delta'}{\Gamma, \Gamma' \vdash \Delta, \Delta'} \qquad \textbf{cut} \tag{4}$$

The cut rule achieves a result in two steps: (i) by using a particular assumption ϕ (right branch) and by (ii) proving that assumption (left branch). This corresponds to a proof which uses "lemma" ϕ. Notice that in the proven sequent (below the bar) ϕ has been annihilated.

Cut and modus ponens

The cut rule is just another way of stating *modus ponens* (m.p.) of natural deduction. The sequent version of *modus ponens* is:

$$\frac{\Gamma \vdash \phi \to \psi \qquad \Gamma \vdash \phi}{\Gamma \vdash \psi}$$

In

$$\dfrac{\dfrac{\boxed{\Gamma \vdash \phi \to \psi}}{\Gamma, \Delta \vdash \phi \to \psi} \qquad \dfrac{\boxed{\Delta \vdash \phi}}{\Delta, \Gamma \vdash \phi}}{\boxed{\Delta, \Gamma \vdash \psi}}\ \text{m.p.}$$

we have used weakening and m.p. to derive a generalized m.p. (boxed sequents) with unequal contexts in the assumptions. From this generalized m.p. one derives cut:

$$\dfrac{\boxed{\Gamma \vdash \phi} \qquad \dfrac{\boxed{\phi, \Delta \vdash \psi}}{\Delta \vdash \phi \to \psi}\ \text{right} \to}{\boxed{\Delta, \Gamma \vdash \psi}}\ \text{m.p.}$$

Cut-elimination

The cut-rule deserves a special place in the order of things, and we explain why this is so at some length. The cut-rule (9) enables the combination of two proofs into a single proof, provided they can - metaphorically speaking - "trade" on a formula ϕ. The necessity to "trade" arises if one proof needs $\phi^{\perp}$ (it books ϕ as an *assumption*), while the other provides ϕ (it books ϕ as a *conclusion*).

The key point about sequent calculus is the famous Hauptsatz of Gentzen, which says that cut is not needed, meaning that the sequent calculus with cut can prove as much as the one without it. The main message comes *from how this is*

achieved. The theorem is proven by exhibiting a procedure through which the cut-rule can be eliminated from a proof without affecting the overall conclusion. The crucial step consists of replacing the occurrence of a cut by one or more cuts on formulae with smaller complexity. In this way the cut(s) bubble toward the leaves of the original proof-structure until they encounter an identity and disappear, i.e., cutting $\Gamma \vdash \psi, \Delta$ with $\psi \vdash \psi$ leaves $\Gamma \vdash \psi, \Delta$.

Cut-elimination

As an example consider the following cut on an implication introduced by the left and right rules (3):

$$\dfrac{\dfrac{\vdots}{\Gamma, \phi \vdash \Delta, \psi}}{\Gamma \vdash \Delta, \phi \to \psi} \qquad \dfrac{\dfrac{\vdots}{\Pi \vdash \Omega, \phi} \qquad \dfrac{\vdots}{\psi, \Xi \vdash \Lambda}}{\Pi, \Xi, \phi \to \psi \vdash \Omega, \Lambda}$$

$$\dfrac{\Gamma \vdash \Delta, \phi \to \psi \qquad \Pi, \Xi, \phi \to \psi \vdash \Omega, \Lambda}{\Pi, \Xi, \Gamma \vdash \Delta, \Omega, \Lambda}\ \text{cut on } \phi \to \psi$$

$$\vdots$$

In the process of cut-elimination this proof-segment is replaced by:

$$\dfrac{\dfrac{\vdots}{\Pi \vdash \Omega, \phi} \qquad \dfrac{\vdots}{\Gamma, \phi \vdash \Delta, \psi}}{\Pi, \Gamma \vdash \Omega, \Delta, \psi}\ \text{cut on } \phi \qquad \dfrac{\vdots}{\psi, \Xi \vdash \Lambda}$$

$$\dfrac{\Pi, \Gamma \vdash \Omega, \Delta, \psi \qquad \psi, \Xi \vdash \Lambda}{\Pi, \Xi, \Gamma \vdash \Delta, \Omega, \Lambda}\ \text{cut on } \psi$$

$$\vdots$$

The two cuts which replace the previous one occur on less complex formulae, and since formulae are finite, this process will bottom out when a cut is finally made on an identity at the leaf of the proof tree. Similar procedures can be carried out for all connectives, independently of whether they are introduced left and right at the same level.

Cut-elimination does something analogous (but not formally identical) to reduction in λ-calculus. It replaces each occurrence of the assumption ϕ with a proof of it. Proofs that use cut are much easier to understand, since they are "modular" in the sense of using generic packages, which can be specialized "on demand" in different ways; for example, the package $\phi \vdash \Gamma$ can be specialized by means of $\Psi \vdash \phi$ to give $\Psi \vdash \Gamma$, or with $\Omega \vdash \phi$ to give $\Omega \vdash \Gamma$, etc. More specifically, a proof may first derive the theorem $(a+b)^2 = a^2 + 2ab + b^2$ and then use it via cut twice, once with $a = 5$ and once with $a = 2$. In the cut-free proof the $(a+b)^2$-theorem would be derived once specifically as $(5+b)^2 = 25 + 10 \cdot b + b^2$ and once specifically as $(2+b)^2 = 4 + 4 \cdot b + b^2$ without exploiting the fact that these are two instances of the same generic structure.

What happens is similar to the application of a function to a particular argument. In fact, for certain versions of the logic (e.g., if sequents are limited to only one formula on their right side) **cut** is exactly analogous to functional application, and the process of cut-elimination corresponds to the evaluation of the function, i.e. it represents the computation. In that case a cut-free proof basically corresponds to a normalized proof in natural deduction; it is a canonical proof [8].

3 Linear logic for tourists

In 1986 Jean-Yves Girard introduced *linear logic*. The system may be regarded as a theory about the control of the contraction and weakening rules (see Appendix 2) of classical logic. In linear logic a formula ψ stands by default for a single occurrence which must be used exactly once. A formula may be, nonetheless, explicitly marked as potentially available in any number of copies ($!\psi$, read as "of course ψ"). Such a supply, however, may be accessed only by another modifier ($?\psi$, read as "why not ψ"). A naked ψ (not under the scope of ! or ? modalities) means exactly one copy of the formula ψ. In this spirit the logical connectives become descriptions of actions in which formulae are consumed.

To avoid confusion with the classical meanings, linear logic has its own notation. Linear implication is written as $\psi \multimap \phi$, and means that ψ is used up when giving rise to ϕ. Linear implication is, therefore, a *causal* relation. The symbol $\otimes$ (read: "cross") denotes a linear conjunction, for example, $\psi \otimes \psi$ indicates the cumulation of two instances of ψ obtained from disjoint resources. The resource sensitivity of linear implication does not permit, for example, $\psi \multimap (\psi \otimes \psi)$. This is in marked contrast to the classical case where $\psi \rightarrow (\psi \wedge \psi)$ is a provable formula.

To appreciate the meaning of this discipline, suppose that atomic formulae stand for real-world tokens. To use a textbook example, consider a vending machine which distributes soda cans and chocolate bars for one dollar each. We could use linear logic to characterize its behavior. Actions like $\texttt{dollar} \multimap \texttt{soda}$ or $\texttt{dollar} \multimap \texttt{chocolate}$ are possible, but not $\texttt{dollar} \multimap (\texttt{soda} \otimes \texttt{chocolate})$. However, we surely have $(\texttt{dollar} \otimes \texttt{dollar}) \multimap (\texttt{soda} \otimes \texttt{chocolate})$. Note that $\texttt{dollar} \multimap \texttt{soda}$ is - like dollar - an action resource that can be used only once; it specifies the conversion of a *particular* dollar into a *particular* soda. To express the idea that this vending machine always converts dollars into sodas or into chocolates, we write $!(\texttt{dollar} \multimap \texttt{soda})$ and $!(\texttt{dollar} \multimap \texttt{chocolate})$.

The control over weakening and contraction has the consequence of requiring us to distinguish between different flavors of the classical connectives according to the way resources are being used. Classical conjunction, $\wedge$, splits into two linear connectives $\otimes$ ("cross") and & ("with"). The formal reason is shown in the detail-box below. The difference can be roughly summarized as follows. $\otimes$

acts as an accumulator of resources. In $\phi \otimes \psi$ both ϕ and ψ have been obtained from disjoint resources, then glued together into a pair which we must use as a unit. In $\phi \& \psi$ both ϕ and ψ arise from the *same* resource, and, therefore, we cannot have them both, but must choose one of them. By projecting out ϕ from $\phi \& \psi$ we lose ψ and vice versa. This difference is reflected by our vending machine which doesn't have an action $\texttt{dollar} \multimap (\texttt{soda} \otimes \texttt{chocolate})$, but does behave like $\texttt{dollar} \multimap (\texttt{soda} \& \texttt{chocolate})$. Note that choice is in the hands of the consumer, hence & is also called an "internal choice".

The splitting of classical conjunction [24]

Consider the case of classical conjunction, $\wedge$, in sequent calculus. We could use the following right $\wedge$-introduction rule:

$$\mathrm{R}_\wedge\colon \quad \frac{\Gamma_0 \vdash \phi, \Delta_0 \qquad \Gamma_1 \vdash \psi, \Delta_1}{\Gamma_0, \Gamma_1 \vdash \phi \wedge \psi, \Delta_0, \Delta_1}$$

However, we could equally well use:

$$\mathrm{R}^*_\wedge\colon \quad \frac{\Gamma \vdash \phi, \Delta \qquad \Gamma \vdash \psi, \Delta}{\Gamma \vdash \phi \wedge \psi, \Delta}$$

In fact, by virtue of weakening and contraction both rules are equivalent. We can derive $\mathrm{R}^*_\wedge$ from $\mathrm{R}_\wedge$:

$$\dfrac{\dfrac{\boxed{\Gamma \vdash \phi, \Delta} \qquad \dfrac{\dfrac{\boxed{\Gamma \vdash \psi, \Delta}}{\Gamma, \Gamma \vdash \psi, \Delta}\,\text{weakening}}{\Gamma, \Gamma \vdash \psi, \Delta, \Delta}\,\text{weakening}}{\Gamma, \Gamma, \Gamma \vdash \phi \wedge \psi, \Delta, \Delta, \Delta}\,\mathrm{R}_\wedge}{\boxed{\Gamma \vdash \phi \wedge \psi, \Delta}}$$

where the double bar indicates the use of multiple contractions. And we can derive $\mathrm{R}_\wedge$ from $\mathrm{R}^*_\wedge$:

$$\dfrac{\dfrac{\dfrac{\boxed{\Gamma_0 \vdash \phi, \Delta_0}}{\Gamma_0, \Gamma_1 \vdash \phi, \Delta_0}}{\Gamma_0, \Gamma_1 \vdash \phi, \Delta_0, \Delta_1} \qquad \dfrac{\dfrac{\boxed{\Gamma_1 \vdash \psi, \Delta_1}}{\Gamma_0, \Gamma_1 \vdash \psi, \Delta_1}}{\Gamma_0, \Gamma_1 \vdash \psi, \Delta_0, \Delta_1}}{\Gamma_0, \Gamma_1, \Gamma_0, \Gamma_1 \vdash \phi \wedge \psi, \Delta_0, \Delta_1, \Delta_0, \Delta_1}\,\mathrm{R}^*_\wedge}{\boxed{\Gamma_0, \Gamma_1 \vdash \phi \wedge \psi, \Delta_0, \Delta_1}}$$

Similarly, the classical left $\wedge$-introduction rule can be written as:

$$\mathrm{L}_\wedge\colon \quad \frac{\Gamma, \phi, \psi \vdash \Delta}{\Gamma, \phi \wedge \psi \vdash \Delta}$$

But equally well one could use:

$$\mathrm{L}^{1*}_{\wedge}\colon\ \frac{\Gamma, \phi \vdash \Delta}{\Gamma, \phi \wedge \psi \vdash \Delta} \qquad\qquad \mathrm{L}^{2*}_{\wedge}\colon\ \frac{\Gamma, \psi \vdash \Delta}{\Gamma, \phi \wedge \psi \vdash \Delta}$$

The classical equivalence of $\mathrm{L}^{\{1,2\}*}_{\wedge}$ with $\mathrm{L}_{\wedge}$ is again easily established. From $\mathrm{L}^{\{1,2\}*}_{\wedge}$ to $\mathrm{L}_{\wedge}$ by means of contraction:

$$\dfrac{\dfrac{\dfrac{\boxed{\Gamma, \phi, \psi \vdash \Delta}}{\Gamma, \phi \wedge \psi, \psi \vdash \Delta}}{\Gamma, \phi \wedge \psi, \phi \wedge \psi \vdash \Delta}}{\boxed{\Gamma, \phi \wedge \psi \vdash \Delta}}$$

We skip the other direction, from $\mathrm{L}_{\wedge}$ to $\mathrm{L}^{\{1,2\}*}_{\wedge}$, where weakening is used.

When contraction and weakening are absent, as in linear logic, the two sets of left/right introduction rules are no longer equivalent. Consequently, the connectives they introduce must be distinguished. Let us denote by $\otimes$ ("cross") the conjunctive connective obtained by the previously unstarred L/R pair:

$$\mathrm{R}_{\otimes}\colon\ \frac{\Gamma_0 \vdash \phi, \Delta_0 \qquad \Gamma_1 \vdash \psi, \Delta_1}{\Gamma_0, \Gamma_1 \vdash \phi \otimes \psi, \Delta_0, \Delta_1} \qquad\qquad \mathrm{L}_{\otimes}\colon\ \frac{\Gamma, \phi, \psi \vdash \Delta}{\Gamma, \phi \otimes \psi \vdash \Delta} \tag{5}$$

As can be seen from the rules, $\otimes$ is a "context-free" or "multiplicative" version of conjunction, in the sense that there is no restraint on the contexts for the $\mathrm{R}_{\otimes}$-rule (Γ_0, Γ_1). With the connective $\otimes$, the side formulae of each premises are accumulated in the conclusion. It is in this sense that that $\otimes$ acts as an accumulator of resources.

The other conjunctive connective, & ("with"), is defined by the previously starred $\mathrm{L}^*/\mathrm{R}^*$ pair:

$$\mathrm{R}_{\&}\colon\ \frac{\Gamma \vdash \phi, \Delta \qquad \Gamma \vdash \psi, \Delta}{\Gamma \vdash \phi \& \psi, \Delta} \qquad \mathrm{L}^{1}_{\&}\colon\ \frac{\Gamma, \phi \vdash \Delta}{\Gamma, \phi \& \psi \vdash \Delta} \qquad \mathrm{L}^{2}_{\&}\colon\ \frac{\Gamma, \psi \vdash \Delta}{\Gamma, \phi \& \psi \vdash \Delta}$$

In contrast to $\otimes$, the contexts must be the *same* for the R-rule to be applicable here. This makes & "contextual" or "additive" in the sense of a superposition. Said differently: the side formulae in each premise *coincide* with the side formulae of the conclusion, and, hence, & is not accumulative.

It is worth pointing out that, in the absence of contraction and weakening, the split between the introduction rules must occur in the way just shown. There cannot be a conjunction introduced, for example, by the L^*/R pair of rules. Cut-elimination would fail otherwise (see [24]). (In the classical case - where contraction and weakening ensure equivalence between $\otimes$ and & - it is, however, customary to use the L^*/R pair of rules to introduce $\wedge$.)

Similar arguments hold for classical disjunction, $\vee$, which in the absence of contraction and weakening splits into two linear connectives: $\parr$ ("par") and $\oplus$ ("either"). Again, with respect to resources the former is "cumulative" and the latter

expresses "superposition". Like "with", $\oplus$ is a choice, but in contrast to "with" the choice is external to the action. For example, our vending machine may be defective at times and swallow your `dollar` returning `nothing`. This choice is not under the control of the customer, hence: `dollar` $\multimap$ ((`soda`&`chocolate`) $\oplus$ `nothing`).

The connective $\parr$ in $\phi \parr \psi$ expresses a mutual dependency of ϕ and ψ, which can be stated through linear implication. $\phi \parr \psi$ is equivalent to both $\phi^\perp \multimap \psi$ or $\psi^\perp \multimap \phi$. The symbol $\perp$ denotes "linear negation", and is *defined* by formal fiat. First, one postulates (like in classical logic) equivalence between $\phi^{\perp\perp}$ and ϕ, i.e.

$$\phi^{\perp\perp} \stackrel{\text{def}}{=\!=} \phi$$

This property is also called "involutivity". Second, negation expresses *dualities* between the linear connectives, much like the deMorgan laws in classical logic (e.g., $\phi \vee \psi = \neg(\neg\phi \wedge \neg\psi)$, where $\neg$ is classical negation):

$$\begin{aligned}
(\phi \otimes \psi)^\perp &\stackrel{\text{def}}{=\!=} \phi^\perp \parr \psi^\perp \\
(\phi \parr \psi)^\perp &\stackrel{\text{def}}{=\!=} \phi^\perp \otimes \psi^\perp \\
(\phi \oplus \psi)^\perp &\stackrel{\text{def}}{=\!=} \phi^\perp \& \psi^\perp \\
(\phi \& \psi)^\perp &\stackrel{\text{def}}{=\!=} \phi^\perp \oplus \psi^\perp
\end{aligned}$$

The involutivity of negation allows to pass from two-sided sequents, $\Gamma \vdash \Delta$, to equivalent one-sided sequents, $\vdash \Gamma^\perp, \Delta$, where $\Gamma^\perp$ is the linear negation of all formulae in Γ. With respect to logical consequence, $\vdash$, linear negation behaves like a matrix transposition in linear algebra.

One-sided sequents

One-sided sequents utlize the dualities expressed by $\perp$ to halve the rules needed for defining the linear connectives. For example, take the defining rules for $\otimes$ (5), and pass to the one-sided version by linearly negating what's on the left:

$$\mathrm{R}_\otimes\colon \frac{\vdash \phi, \Gamma_0^\perp, \Delta_0 \qquad \vdash \psi, \Gamma_1^\perp, \Delta_1}{\vdash \phi \otimes \psi, \Gamma_0^\perp, \Gamma_1^\perp, \Delta_0, \Delta_1} \qquad\qquad \mathrm{L}_\otimes\colon \frac{\vdash \phi^\perp, \psi^\perp, \Gamma^\perp, \Delta}{\vdash \phi^\perp \parr \psi^\perp, \Gamma^\perp, \Delta}$$

Because one-sided sequents are right-sided, nothing much changes with respect to the right rules. The left rule of $\otimes$, however, becomes the right rule for $\parr$. A similar situation occurs with the two-sided rules for $\parr$, the right one stays, and the left one turns into the right one of $\otimes$.

This has a very important consequence. The cut rule becomes *symmetric*, in the sense that there is no distinction between a premise and a conclusion within a sequent:

$$\frac{\vdash \Delta, \phi \qquad \vdash \phi^\perp, \Gamma}{\vdash \Delta, \Gamma}$$

The asymmetry between "premise" and "conclusion" is mirrored in the computational arena by the role-asymmetry between function and data, despite their syntactic indistinguishability. A function sends a premise into a conclusion and the datum supplies the premise. Ultimately this asymmetry reflects the *sequential* paradigm of a functional calculus. Its removal makes linear logic one approach to concurrency (Appendix 3).

Cut can occur between any formula and its dual (negation). The connective $\parr$, for example, is the dual of $\otimes$, and $\phi \parr \psi$ can be cut with $\phi^\perp \otimes \psi^\perp$. The situation has an especially elegant "geometric" interpretation in Girard's proof-net concept [6] (see Appendix 3.2) which is a sequent-calculus stripped to its syntactical bare bones.

3.1 The rules of the game

For the purpose of reference we conclude with a table of the connectives and the standard set of rules for the multiplicative and additive fragment of linear logic (i.e., MALL, this the fragment without ! and ?).

The linear connectives

disjunction	conjunction	resource use
$\parr$	$\otimes$	cumulative
$\oplus$	$\&$	superposition
$\longleftarrow$ duality $\longrightarrow$		

The rules for the linear connectives

(Although we have treated sequents as (multi)sets, i.e. $\vdash \phi, \phi, \psi$ is the same as $\vdash \phi, \psi, \phi$, the permutation rule is stated as an explicit reminder that sequents are modulo permutation.)

Identity and cut

$$\frac{}{\vdash \phi, \phi^\perp}\ \textit{identity} \qquad\qquad \frac{\vdash \Gamma, \phi \qquad \vdash \phi^\perp, \Delta}{\vdash \Gamma, \Delta}\ \textit{cut}$$

Permutation

$$\frac{\vdash \Gamma}{\vdash \Gamma'}\ \Gamma' \textit{ is a permutation of } \Gamma$$

Connectives

$$\boxed{\dfrac{\vdash \Gamma, \phi \qquad \vdash \psi, \Delta}{\vdash \phi \otimes \psi, \Gamma, \Delta}\ \textit{times}} \qquad \boxed{\dfrac{\vdash \Gamma, \phi, \psi}{\vdash \phi \parr \psi, \Gamma}\ \textit{par}}$$

$$\dfrac{\vdash \Gamma, \phi \qquad \vdash \psi, \Gamma}{\vdash \phi \& \psi, \Gamma}\ \textit{with} \qquad \dfrac{\vdash \Gamma, \phi}{\vdash \phi \oplus \psi, \Gamma}\ \textit{l-plus} \qquad \dfrac{\vdash \Gamma, \psi}{\vdash \phi \oplus \psi, \Gamma}\ \textit{r-plus}$$

The system with only the boxed connectives is known as the multiplicative fragment of linear logic (MLL). We have not formally used the exponential modalities ! and ? in this appendix or the main text. We therefore skip their proof-theoretic definition. The reader is referred to [7] for details.

3.2 Proof-nets

Consider a proof of the sequent $\vdash (A \otimes B^\perp) \otimes C, (A^\perp \parr B) \parr (C^\perp \otimes D), D^\perp$ using the rules listed in the previous section:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdash A, A^\perp \qquad \vdash B, B^\perp}{\vdash A \otimes B^\perp, A^\perp, B}\ \textit{times}}{\vdash A \otimes B^\perp, A^\perp \parr B}\ \textit{par} \qquad \vdash C, C^\perp}{\vdash (A \otimes B^\perp) \otimes C, A^\perp \parr B, C^\perp}\ \textit{times} \qquad \vdash D, D^\perp}{\vdash (A \otimes B^\perp) \otimes C, A^\perp \parr B, C^\perp \otimes D, D^\perp}\ \textit{times}}{\vdash (A \otimes B^\perp) \otimes C, (A^\perp \parr B) \parr (C^\perp \otimes D), D^\perp}\ \textit{par}$$

The atoms A, B, C, D and their negations are introduced as identities at the leaves of the proof. All other occurrences of these atoms derive from their first introduced instance. Writing the connectives as labelled wires between formulae, and connecting with a straight wire a formulae and its negation (as they always are introduced together), we can draw a picture of the above proof where only the essential information is recorded. Every formulae occurs exactly as many times as it has been introduced through identities. Figure 1 shows the proof above in this more concise notation; it is called a *proof-net* [6]. The rules for building proof-nets within MLL are fairly straightforward. A formulae and its negation are always connected by a wire. They form the simplest proofnet. A

$\otimes$ connects two disconnected proof-nets, and a $\parr$ connects two parts within the same proofnet.

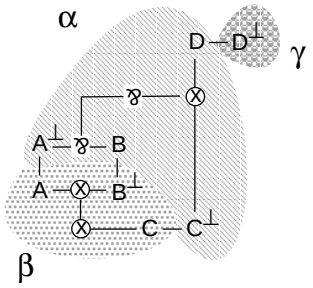


Figure 1: A proofnet.

The three shaded regions α, β, γ partition the proof-net in figure 1 exactly into the formulae of the conclusion sequent: $\beta \stackrel{\text{def}}{=\!=} (A \otimes B^{\perp}) \otimes C$, $\alpha \stackrel{\text{def}}{=\!=} (A^{\perp} \parr B) \parr (C^{\perp} \otimes D)$ and $\gamma \stackrel{\text{def}}{=\!=} D^{\perp}$. The boundaries of each region are always given by atomic links. The formula represented by the β-region can be cut with its dual $\beta^{\perp}$ $(= ((A \otimes B^{\perp}) \otimes C)^{\perp} = (A^{\perp} \parr B) \parr C)$ from another proof-net. The elimination of the cut amounts to disconnecting the cut-formulae at their atomic links and discarding the cut-formulae (but see figure 7, section 2.3.3). For an excellent introduction, see the appendix by Y. Lafont in [8].

The proof-net concept can be extended to full linear logic with the exponentials, ! and ?, as well as the additive connectives $\oplus$ and $\&$. The handling of proof-nets, however, becomes much trickier than in the simple case considered here.

References

[1] H. G. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, second revised edition, 1984.

[2] A. Church. A set of postulates for the foundation of logic. *Annals of Math. (2)*, 33:346–366, 1932.

[3] A. Church. A set of postulates for the foundation of logic (erratum). *Annals of Math. (2)*, 34:839–864, 1932.

[4] M. E. Szabo (ed.). *The Collected Papers of Gerhard Gentzen*. North Holland, Amsterdam, 1969.

[5] G. Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935.

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

[7] J.-Y. Girard. Linear logic: Its syntax and semantics. In J.-Y. Girard, Y. Lafont, and L. Regnier, editors, *Advances in Linear Logic*, pages 1–42. Cambridge University Press, 1995. Proceedings of the Workshop on Linear Logic, Ithaca, New York, June 1993.

[8] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science 7. Cambridge University Press, 1988.

[9] C. Hankin. *Lambda Calculi. A Guide for Computer Scientists*. Clarendon Press, Oxford, 1994.

[10] M. Hennessy. *Algebraic Theory of Processes*. The MIT Press, Cambridge, Mass., 1988.

[11] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, Englewood Cliffs, 1985.

[12] W. A. Howard. The formulae-as-types notion of construction. In J. R. Hindley and J. P. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[13] R. Lalement. *Computation as Logic*. Prentice Hall, Englewood Cliffs, 1993.

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

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

[16] R. Milner. *Communication and Concurrency*. Prentice Hall, New York, 1989.

[17] R. Milner. The polyadic π-calculus: a tutorial. Report ECS-LFCS-91-180, University of Edinburgh, 1991.

[18] R. Milner. Elements of interaction. *Comm. ACM*, 36:78–89, 1993.

[19] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I. *Information and Computation*, 100:1–40, 1992.

[20] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, II. *Information and Computation*, 100:41–77, 1992.

[21] V. R. Pratt. The duality of time and information. In W. Cleaveland, editor, *Proceedings of the Third International Conference on Concurrent Theory*, pages 237–253, New York, 1992. Springer-Verlag.

[22] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD dissertation, University of Edinburgh, 1992.

[23] M. Schönfinkel. Über die Bausteine der mathematischen Logik. *Math. Annalen*, 92:305–316, 1924.

[24] A. S. Troelstra. *Lectures on Linear Logic*. CSLI Lecture Notes 29, Center for the Study of Language and Information, Stanford, California, 1992.