Functional Organization in Molecular Systems and the λ-calculus

Dissertation
zur Erlangung des akademischen Grades
Doctor rerum naturalium

Eingereicht an der
Formal- und Naturwissenschaftlichen Fakultät
der Universität Wien

von
Stefan Müller

Institut für Theoretische Chemie und
Molekulare Strukturbiologie

Wien, im Juli 1999

Meinen Eltern

Before we begin:
Thank you, thank you, thank you, ...

Walter. Was soll man zu diesem Mann noch sagen?
Den Meisterdenkern Peter und Peter. Und Ivo.
Den Gesellen Christoph, Stephan und Alex.
Den Lehrlingen Thomas, Bärbel, Ronke, Susi,
Martin, Jan, Christian, Stefan,
Andreas, Günther, Roman,
Kurt, Wolfgang, Dagmar und Michael.

Zusammenfassung

Biologische Evolution wird in ihrer mathematischen Formalisierung als dynamischer Prozeß von Allelen (Genvarianten) gesehen, der von Fitness und Transmissionsregeln beherrscht wird. Der gegenwärtige Ansatz vernachlässigt, daß Konzepte wie Gen oder Fitness entscheidend vom funktional organisierten Kontext des Phänotyps abhängen. Diesem Zusammenhang kommt jedoch bei Fragestellungen, die die Entstehung des Lebens oder die großen Übergänge in seiner Entwicklung betreffen, zentrale Bedeutung zu. Es ist deshalb notwendig, die Theorie dynamischer Systeme um eine Theorie funktionaler Organisation zu erweitern, deren Anwendungsbereich auf der Ebene chemischer Reaktionsnetzwerke beginnen sollte.

Die Untersuchung von Entstehung und Entwicklung molekularer Organisation erfordert eine adäquate Formalisierung der Chemie. In einem ersten diesbezüglichen Schritt bedienen wir uns einer Korrespondenz zwischen Grundbegriffen der Chemie und Konzepten des sogenannten λ-Kalküls, eines syntaktischen Systems aus der Theorie der Berechenbarkeit. Moleküle werden als symbolische Strukturen dargestellt, die ihrerseits mathematische Funktionen repräsentieren; Reaktionen werden als Funktionsanwendungen aufgefaßt, die weitere Funktionen erzeugen. Diese Modellchemie liefert unter den dynamischen Bedingungen eines stochastischen Flußreaktors Organisationen, die an molekulare Reaktionsnetzwerke erinnern.

Unter einer *Organisation* verstehen wir dabei eine *kinetisch selbsterhaltende algebraische Struktur*. Die Konstituenten einer so definierten Organisation erhalten einander durch wechselseitige Erzeugung im Reaktor und lassen sich durch syntaktische sowie algebraische Gesetzmäßigkeiten charakterisieren. Für den Fall hinreichend einfacher Anfangsbedingungen können die entsprechenden Organisationen mit analytischen Methoden erzeugt werden.

Obwohl grundlegende Eigenschaften der realen Chemie vorläufig vernachlässigt werden, läßt die Vielfalt der generierten Organisationen weitere Untersuchungen vielversprechend erscheinen. Wir präsentieren verschiedene Ansätze zu einer realistischeren Formalisierung der Chemie. Diese betreffen vor allem Wechselwirkungsspezifität, Symmetrie, Reversibilität, Massenerhaltung und Mehrproduktreaktionen. Außerdem skizzieren wir mögliche Modifikationen der dynamischen Aspekte unseres Modells.

Abstract

The current mathematical framework of evolutionary theory lacks a theory of functional organization. The evolutionary process is codified as a problem in the dynamics of alleles (gene variants) governed jointly by fitness and transmission rules. Yet concepts like gene and fitness depend on a functionally organized context known as phenotype. To address problems like the origin of life or the major transitions in evolution, the traditional dynamical systems approach has to be extended with a theory of functional organization. At the most basic level such a theory must apply to molecular reaction networks.

The study of the emergence and evolution of molecular organization requires an adequate formalization of chemistry. In the absence of such a formalism, we start out by exploring a formal metaphor for chemistry called λ-calculus, a syntactical system at the core of computation theory. Molecules are regarded as symbolic structures denoting mathematical functions. Chemical reactions appear as functional applications constructing new functions. Placing this model chemistry in the dynamical setting of a stochastic flow-reactor generates a diversity of organizations reminiscent of molecular reaction networks.

The main conceptual result is a useful working definition of what we mean by an organization. *An organization is a kinetically self-maintaining algebraic structure.* The members of an organization maintain each other in the reactor by mutual production, and they can be characterized by syntactical and algebraic regularities. As a consequence, we apply analytical methods to determine the organizations generated by sufficiently simple seeding sets.

Despite doing violence to chemistry-as-we-know-it, the kinds of organizations generated in our model invite further investigation. We outline several approaches to a more faithful formalization of chemistry. In particular, we address issues pertaining to interaction specificity, symmetry, reversibility, mass conservation, and multiple product reactions. We also discuss modifications to the dynamical setting of our model.

Contents

1 Introduction

> It is the teleonomic apparatus, as it functions when a mutation first expresses itself, that lays down the essential *initial conditions* for the admission, temporary or permanent, or rejection of the chance-bred innovative attempt. It is teleonomic performance, the aggregate expression of the properties of the network of constructive and regulatory interactions, that is judged by selection; and that is why evolution itself seems to be fulfilling a design, seems to be carrying out a "project" [...].

Jacques Monod
Chance and Necessity

1.1 Biology's missing theory

Biology can claim two theories: Darwin's natural selection and Mendel's rules of genetic transmission. Both are correct, and their joint operation can be nicely formalized. Yet together they are insufficient to account for the history of life as we know it.

Consider what is lacking. The formalization of Mendelism and Darwinism, known as the "Modern Synthesis", codified the evolutionary process as a problem in the dynamics of alleles (gene variants) governed jointly by fitness and transmission rules [15, 26, 56]. Yet concepts like gene and fitness depend on a functionally organized context known as organism (phenotype). Our current formal framework for theorizing about evolution assumes, therefore, the prior existence of organisms, that is, the very entities it is meant to explain. Such a theory cannot be reasonably expected to account for the *origin* of biological organization, nor for the *transitions* to new organizational grades – from self-reproducing molecules to self-maintaining metabolisms (or vice versa), to modern cells containing organizational elements that once were autonomous simple cells, to multicellular units with cellular differentiation, to cognizing entities. Evolution is not a process that solely shifts allele frequencies, it also is a process that constructs, modifies and repackages the phenotypic organizations that these alleles give rise to and relative to which they "make sense". Reasoning about the origin and evolution of phenotypes requires at least an adequate representation (and *a fortiori* a definition) of "organization".

These remarks are not meant to downplay the achievement of the Modern Synthesis. The genius of its crafters consisted precisely in abstracting away the organism. They recognized that the synthesis of natural selection and genetics need not await a "theory of the organism" which could not be achieved in the 1930s. That theory is still unavailable, and the broader goal of this thesis is to explore the mathematical framework enabling such a theory to be developed.

This is new territory, and we cannot refer to a widely agreed upon and shared understanding of the problem. We begin, therefore, with a more detailed introduction emphasizing how we have come to understand its nature.

1.2 The methodological problem

A theory of biological organization must be grounded in a representation of what organisms are composed of. At its lowest level the theory must be grounded in chemistry. Chemistry is an illuminating level of analysis for locating the *formal* difficulties towards a theory of organizaton in biology.

The central mathematical tool employed in population biology and genetics is dynamical systems theory. Quite generally, when reasoning about the world in terms of dynamical systems, we associate with each relevant entity a variable that holds the value of a quantifiable property. For example, in the context of a chemical reaction network one such quantity is the concentration of molecules. The interactions among entities are viewed as causing their associated quantities to change. In our example a chemical reaction causes the concentrations of specific molecules to change. Formalizing these changes as coupled systems of nonlinear differential equations (or difference equations) requires the explicit *advance* specification of the relevant variables and their couplings. In chemistry this means knowledge of which reactants generate which products, as well as the advance knowledge of *all* molecules participating in a chemical reaction network. The collective phenomena captured by this methodology are self-sustaining patterns of quantitative change, such as, fixed points, limit cycles, or strange attractors in phase space (that is, concentration space in the chemical example).

The very success of dynamical systems theory is based on a serious limitation: the entities constituting the system are kept outside of the theoretical description. What the formalism describes is the endogenous change of quantities associated with the entities, but never any change of the entities themselves. The formal machinery of dynamical systems does not permit the constituent entities of a system to modify each other to generate new constituents that have not been specified in advance; the formalism is not equipped for handling situations in which *new variables* are generated from within the system, and hence is not equipped for reasoning about innovation. The dynamical systems formalization requires the constituents to be already in place, and, hence, cannot be used to theorize about their formation.

To make this even more explicit, suppose that we wish to study the consequences of adding a new molecule to an existing network. At the level of the dynamical systems formalism there is clearly no way to deduce the cascade of new molecules (and hence new variables) that may be generated. Consequently, there is no way to formally reason about whether and how the network organization is going to be affected by this perturbation. We would have to actually perform the experiment in a real system, figure out the chemical network consequences, and translate them back into a set of coupled differential equations describing the kinetic behavior of the innovated network. This is *post hoc* story telling.

Traditional dynamical systems can give rise to new units of aggregation, or collective phenomena, such as vortices in turbulent flow, which express dynamical feed-backs among the constituent parts of a system. However, if dynamical systems are *extended* to include constructive dependencies, we should expect new kinds of aggregates expressing invariant relationships of mutual production among the components. Phenomena of this kind do capture aspects of biological organization which cannot be represented with traditional dynamics alone. Chemistry is but the simplest physical level where this situation is realized. We have come to call such systems *constructive dynamical systems*.

1.3 Adaptation and organization

To place this framing in an evolutionary perspective, contrast it with approaches to adaptation based solely on population genetics or various flavors of genetic algorithms [40], classifier systems [28] or genetic programming [35]. Like chemistry, many natural or artificial adaptive systems consist of a population dynamics involving objects of a combinatorial nature. These objects might be RNA sequences [19], neural networks, cellular automata [10], assembler code programs [46], strategies in a game [37] or any other kind of discrete structure. The combinatorial nature of these objects makes the set of possible variations astronomically large, such that typical population sizes can realize only a vanishingly small fraction of all possibilities. The standard approach to innovation is *randomization*, where new objects are generated from old by random modification. As a consequence, the process of adaptation (which includes optimization) becomes a search engine driven by a population dynamics which sorts and filters objects according to a performance (or fitness) criterion.

Yet at the phenotypic level biological objects do not vary in the same unconstrained fashion as their underlying genotypes. To understand the process of innovation requires a representation of the phenotype. The innovation of molecular networks is a case in point. Their constrained variation derives from the generative character of chemistry, the consequence of which is the *endogenous* and *specific* (as opposed to random) generation of further molecular components in response to a random perturbation (such as the modification of an existing molecule, or the import of a new one). Seen from this perspective, chemistry has the flavor of a "logic" characterized by a consequence relation enabling new predicates to be inferred from old. Chemistry emphasizes an *algebra* of objects whose internal structure causes specific action yielding further objects that enable further action. The closure of such generative action establishes a concept of organization that permits to distinguish the random origin of a perturbation from its far less random consequences. An organization differs from an arbitrary collection of entities to the extent that it channels change along specific "directions" while being resilient along others. Organizations can adapt, but both the emergence and innovation of organization need not always be adaptations. To address these issues requires a framework where organizations are outcomes of a process rather than being assumed to begin with.

1.4 An abstraction of chemistry

To produce a theory of molecular organizations, how they arise and how they evolve, we must formalize chemistry appropriately (chemical kinetics alone is insufficient). Chemistry comes complete with its own theoretical structure. Yet quantum mechanics is inadequate for our purposes, because it does not formalize molecules as agents of transformation. From the vantage point of quantum mechanics a molecule is a probability amplitude for electrons and nuclei. The point of view we take is one of chemistry as a set of structures which are at the same time "functions" from that set into itself. By failing to formalize molecules as structure-action relations, quantum mechanics misses the algebraic aspect of chemistry which makes it a medium for the organizations we are seeking. What molecular biology lacks is a high level description of chemistry capable of abstracting molecular actions, of plugging them together (like electronic components), and of generating and analyzing the network closures of these actions under a variety of boundary conditions.

Identifying an appropriate level of abstraction for chemistry that is useful to molecular and evolutionary biology is a daunting task. It is equally clear, however, that the utility of such an abstraction goes well beyond the purposes described here, and its implications are limited solely by our imagination (think of "industrial metabolisms"). Yet it is not certain whether such a theory is attainable. In particular, it is not obvious which aspects of chemistry must be abstracted.

Our research tactic follows two guidelines. First, we seek a model that has "theoretical appeal". The value-added of a scientific theory consists in connecting chemistry (and entailed natural phenomena) with powerful mathematical concepts and their associated mathematical theories. We aim at generality. The point here is explicitly *not* to concoct one's arbitrary private chemistry for the sole purpose of *imitating* a particular biological organization on a computer. Second, we begin with rather general features of chemistry, attempting refinements and adjustments as the research program proceeds. We explore the consequences of already existing and familiar formalisms when used *as if* they were chemistry. This implies that we produce *generalized chemistries* which do violence to chemistry-as-we-know-it. We try to understand why extant formalisms fail, and we suggest ways of modifying them or defining new ones.

1.5 Organization of this work

We continue with a brief sketch of our approach and the current status of research. We discuss the analogy between chemistry and the λ-calculus, and present a base model of functional organization (chapter 2).

The λ-calculus is the canonical mathematical theory about functions as rules of computation. We give an introduction to its syntax and basic theory, and consider determinacy and reduction strategy (chapter 3).

Our exploration of organization space starts with an exhaustive enumeration of λ-terms. Next, we consider iterated actions within a set of λ-terms, thereby switching off the dynamical aspect of the base model. As our main result, we present a variety of organizations generated by simple λ-terms (chapter 4).

After drawing the conclusions from our results (chapter 5), we outline several approaches to a more faithful formalization of chemistry, and discuss modifications to the dynamical aspect of the base model (chapter 6).

2 A model of functional organization

In the following we provide an executive summary of the published work which established this area of research [16, 17, 18].

2.1 Chemistry as a calculus

Chemistry deals with structures that encode potential actions. The reactive interaction of one structure with another triggers a specific joint action which yields further structures. Chemists have developed a variety of symbolisms and a suite of informal rules to express these structures and their combinatory relationships. If we turn to mathematics for a theory concerned with an *analogous* situation, we are invariably led to the theoretical foundations of computation.

At its most basic level computation requires a collection of *mechanisms* (devices with predictable local behavior) for constructing syntactical objects and for establishing equivalences between syntactical objects by means of rewrite rules[1]. A representation of computation in this spirit is provided by a formalism known as the λ-calculus, which is a theory of symbolic structures that encode possible transformations of just such symbolic structures.

> **λ-calculus**
>
> The λ-calculus, a syntactical system, is a convenient means to reason about functions as rules of computation. Symbolic structures (λ-*terms*) which are defined inductively are intended to denote mathematical functions. The definitions are inspired by basic computational concepts, such as the declaration (*abstraction*) of a variable, or the *application* of a function. An elementary calculation is represented by a rewrite rule (*reduction*) that replaces a variable in the function term by the argument term (*substitution*). A term that cannot be rewritten anymore is said to be in *normal form*, it corresponds to the result of an evaluation. Reduction strategy, i.e. the order of applying the rewrite rule, does not affect the resulting normal form - if it exists. See Chapter 3 for a detailed presentation.

[1] Arithmetics, for example, establishes equality between syntactically distinct objects such as $6 + 5 - 4$ and $3 \times 2 + 1$ by lawfully rewriting them into syntactically identical objects.

The basic idea underlying our *ansatz* is to *cartoon* molecules as symbolic representations of functions expressed in λ-calculus. Such functions act on each other to produce new functions. The analogy between chemistry and the λ-calculus can be summarized as follows:

chemistry	...	**λ-calculus**
physical molecule	...	symbolic representation of a function
molecule's behavior	...	function's action
chemical reaction	...	evaluation of a functional application

It is helpful to make a distinction between *being* a function and *having* a function. The latter, more intuitively semantic notion, emerges within the context of other objects that *are* functions and a population dynamics (see below). Our model makes this distinction explicit: molecules *are* functions to begin with; which functions they can *have* is but a restatement of the question about the structure of molecular organization.

2.2 Why λ-calculus?

The choice of the λ-calculus as a toy-theory of chemistry is motivated by its role in the wider context of computation and logic.

- The most important analogy to chemistry is that the λ-calculus implements a set Λ of elements (λ-terms) such that every element can be rigorously and consistently interpreted as a function from Λ into Λ.
- In the λ-calculus there is no syntactical distinction between a "function" and an "argument". Each λ-term can be in both roles.
- The λ-calculus is a formal bridge between computation and logic. It represents (in conjunction with a type system) a proof-theory of constructive logic in natural deduction style. This connection suggests an interpretation of a chemical reaction as a kind of deduction, and chemical synthesis (e.g., several reactions in a row) as a kind of proof construction.

2.3 The base model

A base model was implemented to explore the consequences of this *ansatz* [16, 17, 18].

The base model has two core ingredients. As outlined above, the first consists in a rough analogy between, on the one side, molecules and their constructive interactions, and, on the other side, λ-terms representing functions and their mutual application. The second core ingredient consists in a simple constrained dynamics mimicking a well-stirred flow reactor. In the present context this means an ensemble of functions meeting randomly and interacting by application. The interacting functions are treated here as "catalysts", remaining unchanged themselves, yet inducing change (generating a new function as the value of the application). The dynamical constraint results from imposing a finite lifetime on each function in the system, after which it disappears. The behavior of such a reactor is explored by means of computer experiments.

The main observation is that self-maintaining invariant sets of functions emerge with time. A set is self-maintaining, if the pairwise application of all functions in the set regenerates at least that set[2]. Self-maintenance is here the consequence of a *constructive feed-back loop* which occurs when the constructive actions induced by the components of a system lead to the continuous regeneration of these same components. In our model the invariant properties of a self-maintaining set turn out to consist in (i) a specific grammar defining the syntax of all members of the set, thus separating "inside" (member objects) from "outside" (non-member objects) despite the absence of space, and (ii) a collection of algebraic laws which axiomatize the mutual actions among the members. Hence, once convergence to a self-maintaining (fixed-point) collective has occurred, a concise description of it can be given independently of the generic λ-calculus within which the convergence took place. (To understand the biochemistry of the liver, a medical student need not understand the evolution of livers, let alone all of chemistry.) The *algebraic* structure together with its *kinetic* persistence define our notion of "organization".

[2]Such a set reproduces itself at the set level without necessarily involving individual elements that are self-reproducing.

A random aggregation of functions (or molecules) would track every exogenous change imposed on them. An organization, in contrast, is a collection of functions distinguished by both a resistance to change and a capacity to channel change in specific ways. These properties are captured by the algebraic laws characterizing an organization, and which constrain an organization's response to "subtractive" and "additive" perturbation. By subtractive perturbation we mean the removal of a subset of functions participating in an organization, and by additive perturbation we refer to the exogenous introduction of new functions.

Organizations arising in the base model are robust. They resist subtractive perturbations, because the remaining functions restore lost members. Such "generating sets" implement a constructive version of redundancy and fault-tolerance. Organizations also resist additive change, because the permanent inclusion of a newly added function requires that its interactions spawn pathways which eventually feed back to secure its own production. Such pathways are highly constrained by the laws specific to an organization. However, when permanent inclusion occurs, the algebraic laws of the old organization are not destroyed, but rather extended with additional equations. Finally, different organizations can be combined into higher level organizations, while maintaining their autonomy. When this occurs, their coexistence is structural, because it is mediated by a "glue" consisting of cross-interaction products which do not belong to either component organization.

3 λ-calculus

In this presentation of the λ-calculus we follow the encyclopaedic book by Barendregt [3], as well as the excellent textbook by Hankin [27].

3.1 History

The λ-calculus is a theory about functions as rules, rather than as graphs. "Functions as rules" is the old fashioned notion of function and refers to the process of going from argument to value, a process coded by a definition. The idea, usually attributed to Dirichlet, that functions could also be considered as graphs, that is, as set of pairs of argument and value, was an important mathematical contribution. Nevertheless the λ-calculus regards functions again as rules in order to stress their computational aspects.

For example, we may think of functions given by definitions in ordinary English applied to arguments also expressed in ordinary English. Or, more specifically, consider functions as given by programs for machines applied to, that is, operating on, other such programs. In both cases we have a type free structure, where the objects of study are at the same time function and argument. This is the starting point of the type free λ-calculus. In particular a function can be applied to itself. For the usual notion of function in mathematics, as in Zermelo-Fraenkel set theory, this is impossible.

There are three aspects of the λ-calculus:

- Foundations of mathematics
- Computations
- Pure λ-calculus

3.1.1 Foundations of mathematics

The founders of the λ-calculus and the related theory of combinatory logic had two aims in mind:

(1) To develop a general theory of functions, dealing, for example, with formula manipulations.

(2) To extend that theory with logical notions providing a foundation for logic and (parts of) mathematics.

The first point was explicitly stressed by Schönfinkel and Curry, the founders of combinatory logic, and it was also implicit in the work of Church founding the λ-calculus.

Unfortunately all attempts to provide a foundation for mathematics failed. Church's original system [7] was inconsistent as shown by Kleene and Rosser [30]. As was pointed out in [1], Frege's well known inconsistent theory [21] essentially contains the λ-calculus; so that was in fact another failure.

Curry [11] did provide a consistent theory fulfilling the first point above (pure combinatory logic), but the logical part of this theory is too weak to be adequate as a foundation.

After the Kleene-Rosser paradox Church was discouraged in his foundational program. Church [8] gave a consistent (as shown by the Church-Rosser theorem) subtheory of his original system dealing only with the functional part. This theory is the λI-calculus.

Curry on the other hand did not want to "run away from the paradoxes". He proposed to extend pure combinatory logic in order to fulfil the second aim and to give at the same time an analysis of the paradoxes [12, 13]. Although Curry's program has not been completed, some progress has been made; see [50, 52, 1].

There are also relations between the λ-calculus and proof theory. The λ-calculus used in this context is, however, the typed λ-calculus.

3.1.2 Computations

Recursion theory

The part of the λ-calculus dealing only with functions turned out to be quite successful. Using this theory Church proposed a formalization of the notion "effectively computable" by the concept of λ-definability. Kleene [29] showed that λ-definability is equivalent to (partial) recursiveness and in the meantime Church formulated his thesis, stating that recursiveness is the proper formalization of effective computability. Turing [54, 55] gave an analysis of machine computability and showed that the resulting notion (Turing computability) is equivalent to λ-definability.

After the discovery of the paradoxes, Kleene translated results on λ-definability and obtained several fundamental recursion theoretic theorems; see [9].

Computer Science

The rise of the computer in the last few decades resulted in an extensive development of programming languages. The λ-calculus possesses several features of programming languages and their implementations. For eaxmple, bound variables in the λ-calculus correspond to formal parameters in a procedure; and the type free aspect corresponds to the fact that for a machine a program and its data are the same, namely a sequence of bits.

By the analysis of Turing it follows that in spite of its very simple syntax, the λ-calculus is strong enough to describe all mechanically computable functions. Therefore the λ-calculus can be viewed as a paradigmatic programming language. This certainly does not imply that one should use it to write actual programs. What is meant is that in the λ-calculus several programming problems, specifically those concerning procedure calls, are in a pure form. A study of these can bear some fruits for the design and analysis of programming languages.

For example, several programming languages have features inspired by λ-calculus (perhaps unconsciously so). In Algol or Pascal, procedures can be arguments of procedures. In LISP the same is true and moreover a procedure can be the output of a procedure. For relations between λ-calculus and programming languages, see [36, 42, 24].

Because of the similarities of λ-calculus and some programming languages, ideas for the semantics of the former may be applied to the latter. Landin gave a semantics of Algol by translating this language in λ-calculus and by describing an *operational semantics* for the latter. For further work on this kind of semantics, see [44, 45, 43, 5].

In the meantime there was a need for a *denotational semantics* of programming languages (in order to express what is the functional meaning of a program). A similar problem had occurred in a pure form in the λ-calculus. Due to the type free character it was not clear how to construct models for that theory. One would want a set X in which its function space can be embedded, i.e. $X \cong X \to X$; for cardinality reasons this is impossible. The difficulty was overcome by Scott, who constructed λ-calculus models by restricting $X \to X$ to *continuous* functions on X (with a proper topology). Only then did it become clear how to develop a denotational semantics for programming languages. This is because Scott's method is powerful enough to give also a meaning to the following two features of programming languages: recursion and data types. See [51, 38, 53, 25].

3.1.3 Pure $\boldsymbol{\lambda}$-calculus

λ-calculus, as treated in the rest of this section, is not directed towards applications as above, but is studied for its own interest.

The formal (type free) λ-calculus, a theory denoted by $\boldsymbol{\lambda}$, studies functions and their applicative behaviour. Therefore *application* is a primitive operation of $\boldsymbol{\lambda}$. The function f applied to the argument a is denoted by fa. Complementary to application there is *abstraction*. Let $t(x)$ be an expression possibly containing x. Then $\lambda x.t(x)$ is the function f that assigns to the argument a the value $t(a)$. That is

$$(\lambda x.t(x))a = t(a). \tag{β}$$

That one considers only application and abstraction for unary functions is reasonable, following an observation of Schönfinkel [49]. He remarked that functions of several variables could be reduced to unary functions. If $f(x, y)$ is e.g. a binary function, then define $f_x = \lambda y.f(x, y)$ and $a = \lambda x.f_x$. Then one has $(ax)y = f_x y = f(x, y)$.

The theory $\boldsymbol{\lambda}$ has as terms the set Λ (λ-*terms*) built up from variables using application and abstraction. The statements of $\boldsymbol{\lambda}$ are equations between λ-terms, and $\boldsymbol{\lambda}$ has as its only mathematical axioms the scheme (β). (Some trivial identifications like $\lambda x.x = \lambda y.y$ have to be made however.)

As was mentioned above, the λ-calculus studies functions as rules. Because of this intensional character, the linguistic means to describe these functions, the λ-terms, play a central role. By the theory $\boldsymbol{\lambda}$ some of these λ-terms are identified (so-called *convertible* terms). But it is important to note that the theory $\boldsymbol{\lambda}$ is itself not the focus of interest; it is just a way to generate the principal objects, namely the *terms modulo convertibility*. Only then does the real λ-calculus start: the study of these objects using all possible mathematical tools.

The following questions show how these objects are studied:

(1) What kind of functions (on terms) are definable (as term)?

(2) Let M, N be non-convertible terms. Is it reasonable to identify M and N (taking a less intensional view)?

(3) What is the correspondence between the applicative behaviour and the syntactic form of terms?

Some typical answers to question (1) are as follows:

- Restricted to numerals exactly the recursive functions are definable.
- The range of all definable functions on terms is always infinite or a singleton.
- There is a natural topology on terms, the so-called tree topology.

  The topology is obtained by associating to terms so-called Böhm trees and then translating the Scott topology on these trees to terms.

Question (2) can be approached by either giving consistency proofs of natural extensions of $\boldsymbol{\lambda}$ or by constructing models for $\boldsymbol{\lambda}$ and considering the set of equations true in these models. The theory $\boldsymbol{\lambda\eta}$ (extensional λ-calculus) is obtained by the first method. The theories $\mathcal{B}$ (which identifies terms with the same Böhm tree) and $\mathcal{K}^*$ (which identifies solvably equivalent terms) are formed by the second method.

With regard to question (3), let F be a λ-term. One has e.g. the following:

- F is solvable iff F has a head normal form.

  This relates the syntactical notion of head normal form, i.e. $F = \lambda x_1 \ldots x_n.x_i M_1 \ldots M_m$, to solvability, which indicates applicative behaviour, i.e. $\exists n\ \exists P_1 \ldots P_n : F P_1 \ldots P_n = \mathbf{I} \equiv \lambda x.x$.
- F is $\beta\eta$-invertible iff F is a finite hereditary permutation.

  Here $\beta\eta$-invertibility means that there exists a λ-term G such that in the extensional λ-calculus $F \circ G = G \circ F = \mathbf{I}$, where $\circ$ denotes the composition of functions. The finite hereditary permutations are some syntactically defined class of λ-terms.

Now that we have seen the questions and some results, let us indicate how the subject is treated. There is some connection between pure λ-calculus and number theory. The connection is superficial, in that methods and results are totally different, but it is nevertheless good to indicate it in order to obtain some feeling of how the theory $\boldsymbol{\lambda}$ is organized.

An interesting aspect of number theory is that results about $\mathbb{Z}$ and $\mathbb{Q}$ are sometimes proved via results about $\mathbb{R}$ or $\mathbb{C}$. Similarly results about λ-terms are sometimes proved via "continuous" models. In this context the Böhm trees play an important role. These possibly infinite trees may be compared to the continued fraction of the reals. Each tree is the limit of finite trees. A natural topology on λ-terms can be defined using the Böhm trees. With respect to this topology the λ-calculus operations are continuous. This continuity theorem has several applications. Thus the introduction of models and continuity considerations make the theory really λ-*calculus*.

3.2 Notation and basic theory

We now start our study of the λ-calculus. First, we turn to the question of notation and present the inductive definition of λ-terms and some auxiliary notions such as *bound* and *free variables* and *sub-terms*. Next, we present the theory $\boldsymbol{\lambda}$. Central to the theory is the notion of substitution – the driving force behind function application. The theory $\boldsymbol{\lambda}$ is a theory of equality between terms; as already indicated, we are trying to capture intensional equality, but we also show how the theory can be extended to capture extensional equality. Finally, we consider the consistency and completeness of the theories presented.

3.2.1 Notation

λ-terms are constructed from variables, the symbol λ and parentheses:

$x_1, x_2, \ldots$

λ

$(,)$

Of course, this does not define the class of λ-terms; however, it does define the alphabet which can be used. We now give the inductive definition of λ-terms.

Definition 3.1 ($\boldsymbol{\lambda}$-terms)
The class Λ of λ-terms is the least class satisfying the following conditions, where x denotes an arbitrary variable:

$$\begin{array}{lll} (1) & x \in \Lambda & \\ (2) & M \in \Lambda & \Rightarrow \quad (\lambda x M) \in \Lambda \\ (3) & M, N \in \Lambda & \Rightarrow \quad (MN) \in \Lambda \end{array}$$

Terms of the form $(\lambda x M)$ are called *abstractions*; they correspond to functions in programming languages. The variable following the abstractor λ corresponds to the formal parameter of a function, whereas M corresponds to the function body. Thus

$$(\lambda x x)$$

should be compared to

```
(LAMBDA (x) (x))
```

in LISP, or to

```
FUNCTION id(x:integer):integer;
      BEGIN id:=x END;
```

in PASCAL[3].

Terms of the form (MN) are called *applications*; the function term M is applied to the argument term N.

To avoid the proliferation of parentheses, we will use the following notation constantly. It results from Schönfinkel's reduction of functions of many variables to those of one variable.

Definition 3.2 (Notation)
Let the symbol $\equiv$ denote syntactic equality. With $\vec{x} \equiv x_1, \ldots, x_n$ and $\vec{N} \equiv N_1, \ldots, N_n$ we define:

$$\begin{array}{llllll} (1) & \lambda x_1 \ldots x_n.M & \equiv & \lambda \vec{x}.M & \equiv & (\lambda x_1(\lambda x_2 \ldots (\lambda x_n M) \ldots)) \\ (2) & MN_1 \ldots N_n & \equiv & M\vec{N} & \equiv & (\ldots((MN_1)N_2) \ldots N_n) \end{array}$$

[3]The λ-term and the LISP program are type-free. This is in contrast to the PASCAL function which is strongly typed. Both the λ-term and the LISP program are actually equivalent to a whole set of PASCAL functions with an element for every type.

Example 3.3
The following are λ-terms:

$$\begin{array}{rlcl}
(i) & \lambda x.x & \equiv & (\lambda xx) \\
(ii) & xy & \equiv & (xy) \\
(iii) & \lambda xy.yx & \equiv & (\lambda x(\lambda y(yx))) \\
(iv) & \lambda xy.yx(\lambda z.xz) & \equiv & (\lambda x(\lambda y((yx)(\lambda z(xz)))))
\end{array}$$

The symbol λ acts as a variable binder in a similar fashion to $\int \ldots dx$ in integral calculus and the quantifiers $\exists$ and $\forall$ in predicate calculus. A variable x occurs *bound* in a λ-term M if x is in the scope of a λx; x occurs *free* otherwise.

Definition 3.4
$BV(M)$ is the set of bound variables in M and can be defined inductively as follows:

$$\begin{array}{lcl}
BV(x) & = & \{\} \\
BV(\lambda x.M) & = & BV(M) \cup \{x\} \\
BV(MN) & = & BV(M) \cup BV(N)
\end{array}$$

$FV(M)$ is the set of free variables in M and can be defined inductively as follows:

$$\begin{array}{lcl}
FV(x) & = & \{x\} \\
FV(\lambda x.M) & = & FV(M) \setminus \{x\} \\
FV(MN) & = & FV(M) \cup FV(N)
\end{array}$$

When $FV(M)$ is the empty set $\{\}$, M is said to be *closed*. Closed terms are sometimes called *combinators* and the class of all such terms is Λ^0.

In the following, we also make use of the notion of subterm. A *subterm* of a λ-term M is some part of the term which is itself a λ-term.

Definition 3.5
$Sub(M)$ is the set of subterms of M and can be defined inductively as follows:

$$\begin{array}{lcl}
Sub(x) & = & \{x\} \\
Sub(\lambda x.M) & = & Sub(M) \cup \{\lambda x.M\} \\
Sub(MN) & = & Sub(M) \cup Sub(N) \cup \{MN\}
\end{array}$$

When we want to prove something about λ-terms we will often use the technique of *structural induction*. A proof by structural induction has a very similar structure to a proof by mathematical induction. The *basis* consists of a demonstration that the predicate holds for each of the primitive terms and the *inductive step* includes a separate case for each type of composite term in the language (using a hypothesis which states that the predicate is true for the immediate subterms of the composite term). To be more concrete, we give a simple example of a λ-term property.

Example 3.6
Every term in Λ has balanced parentheses.

Proof
Basis:
Variables – trivial since variables contain no parentheses.

Inductive Step:
Consider the abstraction $\lambda x.M \equiv (\lambda x M)$. By the inductive hypothesis we have that M is balanced. Thus $\lambda x.M$ is also balanced.

Consider the application $MN \equiv (MN)$. By the inductive hypothesis twice, we have that both M and N are balanced. Thus MN is also balanced. $\square$

We conclude this section by defining the class of λ-contexts. Often we will need the notion of a partially specified term, that is a term with "holes" in it. Such a term gives a context into which we can put other terms (to fill the holes). The ability to construct contexts will clarify some definitions (e.g. the notion of *compatibility*) and generalize some results (e.g. the Substitution Lemma). We give an inductive definition of contexts for λ-terms:

Definition 3.7 (Contexts)
The class Λ_C of λ-contexts is the least class satisfying:

$$\begin{array}{lll} (1) & x, [\,] \in \Lambda_C & \\ (2) & C[\,] \in \Lambda_C & \Rightarrow \quad (\lambda x C[\,]) \in \Lambda_C \\ (3) & C_1[\,], C_2[\,] \in \Lambda_C & \Rightarrow \quad (C_1[\,]C_2[\,]) \in \Lambda_C \end{array}$$

If $C[\,] \in \Lambda_C$ and $M \in \Lambda$, then $C[M]$ denotes the result of placing M in the holes of $C[\,]$. In this act free variables of M may become bound in $C[M]$.

Example 3.8
Let $C[\,] \equiv \lambda x.x(\lambda y.[\,]y)$ and $M \equiv \lambda z.xz$. Then:
$C[M] \equiv \lambda x.x(\lambda y.(\lambda z.xz)y)$

3.2.2 The theory $\boldsymbol{\lambda}$

We can construct formulas from the terms; a theory then establishes certain formulas as axioms and provides rules of inference which enable us to derive new formulas. The true formulas (either axioms or formulas that can be derived from the rules) are called *theorems*.

We now present a theory of equality (or *convertibility*) between λ-terms. There are a number of reasonable requirements for such a theory:

(1) An application term should be equal to the result obtained by applying the function term to the argument term.

For example, suspending your knowledge of PASCAL, imagine that PASCAL functions can be higher-order (take functions as arguments and produce them as results) and that we have defined a higher-order variant of `id`. Then `id(fun)` should surely be the same function as `fun` (for any appropriate parameter `fun`).

(2) Equality should be an equivalence relation.

(3) Equal terms should be equal in any context.

These requirements go some way to motivating the following theory $\boldsymbol{\lambda}$:

Definition 3.9
The theory $\boldsymbol{\lambda}$ *has formulas of the form* $M = N$, *with* $M, N \in \Lambda$, *and the following axioms and rules:*

(1) $(\lambda x.M)N = M[x := N] \qquad (\beta)$

(2) $M = M$
$M = N \;\Rightarrow\; N = M$
$M = N,\, N = L \;\Rightarrow\; M = L$

(3) $M = N \;\Rightarrow\; MZ = NZ$
$M = N \;\Rightarrow\; ZM = ZN$
$M = N \;\Rightarrow\; \lambda x.M = \lambda x.N \qquad (\xi)$

The axiom (β) corresponds to function evaluation. The notation $M[x := N]$ should be read “replace free occurrences of x in M by N” (some care must be taken – we return to this in the next section). Readers should compare the axiom (β) to their intuitive understanding of function calls in a familiar programming language.

The rule (ξ) is sometimes called the rule of weak extensionality. The classical presentation of the theory also included two additional axioms: (α) which allows a change of bound variable names and (η) which introduces extensional equality; see the next sections for a discussion.

We write $\boldsymbol{\lambda} \vdash M = N$ to mean that $M = N$ is a theorem of $\boldsymbol{\lambda}$ and read the theorem as "M and N are convertible". The notation of the last section and this theory are variously called λ-calculus, $\lambda\beta$-calculus and λK-calculus.

The following quite useful theorem gives an illustration of what can be proved in $\boldsymbol{\lambda}$.

Theorem 3.10 (Fixed Point Theorem)

$$\forall F \in \Lambda \quad \exists X \in \Lambda : \; FX = X$$

Proof
Let $W \equiv \lambda x.F(xx)$ and $X \equiv WW$. Then:

$$X \equiv WW \equiv (\lambda x.F(xx))W = F(WW) \equiv FX \qquad \square$$

X is called a *fixed point* of F; if we apply F to X, the resulting term is convertible with X. In a more familiar context, for example, 1 is a fixed point of the squaring function. The Fixed Point Theorem may seem quite surprising at first sight; It says that all terms have fixed points. For some terms, such as $\lambda x.x$, which is the idendity function, this is obvious (all terms are fixed points of the identity) but for others it is not so clear. However, the proof of the Fixed Point Theorem is constructive; it gives a recipe for constructing a fixed point of any term.

Example 3.11
Let $F \equiv \lambda xy.xy$. Then the above construction leads to:

$$W \equiv \lambda x.(\lambda xy.xy)(xx) = \lambda x.\lambda y.(xx)y \equiv \lambda xy.xxy$$

The required fixed point is thus $X \equiv (\lambda xy.xxy)(\lambda xy.xxy)$. We check this by:

$$X \equiv (\lambda xy.xxy)(\lambda xy.xxy) = \lambda y.(\lambda xy.xxy)(\lambda xy.xxy)y$$
$$FX \equiv (\lambda xy.xy)((\lambda xy.xxy)(\lambda xy.xxy)) = \lambda y.((\lambda xy.xxy)(\lambda xy.xxy))y$$

Fixed points are important in Computer Science. They play a fundamental role in the semantics of recursive definitions. For example, the factorial function

$$F(N) = \begin{cases} 1 & \text{if } N = 0 \\ N \cdot F(N-1) & \text{otherwise} \end{cases}$$

can be λ-defined by

$$\mathit{fac}\ n = \mathit{if}\,(\mathit{zero}\ n)\,\mathit{1}\,(\times n(\mathit{fac}(\mathit{pred}\ n)))$$

which leads to the following fixed point equation:

$$\mathit{fac} = (\lambda \mathit{fn}.\mathit{if}\,(\mathit{zero}\ n)\,\mathit{1}\,(\times n(f\,(\mathit{pred}\ n))))\,\mathit{fac}$$

Of course, we must be careful about reading too much into this equation: *if, zero, 1,* $\times$*, pred* are just formal symbols, they have no deeper significance in the λ-calculus which we have defined so far. We shall return to this point later.

3.2.3 Substitution

We now return to the substitution operation used in the axiom (β). A naive approach to defining this operation leads to the problem of "variable capture". This problem occurs when we naively substitute a term containing a free variable into a scope where the variable becomes bound. For example:

$$(\lambda xy.xy)y \neq \lambda y.yy$$

The free variable y in the left hand term is analogous to a global parameter in programming; in the right hand side it has become confused with the bound variable y (formal parameter). We will consider two different approaches to this problem before selecting the latter for use in the rest of this section.

The Classical Approach

The first approach is based on Church's original treatment of substitution. We use the following definition:

(1) $x[x := N] \equiv N$
(2) $y[x := N] \equiv y,$ if $x \not\equiv y$
(3) $(\lambda x.M)[x := N] \equiv \lambda x.M$
(4) $(\lambda y.M)[x := N] \equiv \lambda y.(M[x := N]),$ if $x \notin FV(M)$ or $y \notin FV(N)$
(5) $(\lambda y.M)[x := N] \equiv \lambda z.(M[y := z])[x := N],$ otherwise, z new
(6) $(M_1 M_2)[x := N] \equiv (M_1[x := N])(M_2[x := N])$

We consider the rules 3 to 5 in a bit more detail. Rule 3 applies when the variable being substituted is bound at the outermost level; in this case there will be no free occurrences of x in the remainder of the expression and thus the substitution has no effect. Rule 4 is applicable when variable capture cannot occur: either x does not occur free in the body (in which case the substitution is a void operation again) or the variable that is bound at the outermost level does not occur in the term being substituted (no capture); in either case the substitution can be pushed through the λ to apply to the body. The final rule 5 applies when variable capture does occur, that is when substitution does take place and the variable bound at the outermost level does occur free in the term being substituted; in this case, we rename the bound variable to a completely new variable.

Rule 5 is valid under the assumption that terms which are similar, having the same free variables and differing only in their bound variables, are equivalent. This is reasonable if we think about programming languages:

```
FUNCTION id(y:integer):integer;
         BEGIN id:=y END;
```

is clearly the same function as the earlier one with the same name; we have only changed the formal parameters. In Church's original presentation of the λ-calculus there was an additional axiom (α) which formalizes the above discussion:

(α) $\lambda x.M = \lambda y.M[x := y],\ y \notin FV(M)$

The Variable Convention

For our second definition of the substitution operation we use the following strategy:

- Identify two terms if one can be transformed into the other by a renaming of bound variables.
- Consider a λ-term as a representative of its equivalence class.

- Interpret substitution as an operation on the equivalence classes, using representatives according to a variable convention.

We start with two definitions:

Definition 3.12 (Change of Bound Variables)
M' is produced from M by a change of bound variables if $M \equiv C[\lambda x.N]$ and $M' \equiv C[\lambda y.(N[x := y])]$ where y does not occur at all in N and $C[]$ is a context with one hole.

Definition 3.13 (α-congruence)
M is α-congruent to N, written $M \equiv_\alpha N$, if N results from M by a series of changes of bound variables.

Example 3.14
According to the above definitions, we have:

$$\begin{array}{rl} (i) & \lambda x.x \equiv_\alpha \lambda y.y \\ (ii) & \lambda x.xy \equiv_\alpha \lambda z.zy \not\equiv_\alpha \lambda y.yy \\ (iii) & \lambda x.x(\lambda x.x) \equiv_\alpha \lambda x'.x'(\lambda x.x) \equiv_\alpha \lambda x'.x'(\lambda x''.x'') \end{array}$$

In contrast to the classical approach, we prefer to identify equivalent terms on a syntactic level. We also adopt a convention about the choice of bound variables.

Definition 3.15 (Congruence Convention)
Terms that are α-congruent are identified: $M \equiv_\alpha N \;\Rightarrow\; M \equiv N$

Definition 3.16 (Variable Convention)
If $M_1, \ldots, M_n$ occur in a certain context then in these terms all bound variables are chosen different from free variables.

Definition 3.17 (Substitution)
With the above conventions, we can define substitution as follows:

$$\begin{array}{ll} (1) & x[x := N] \equiv N \\ (2) & y[x := N] \equiv y, \;\; \textit{if } x \not\equiv y \\ (3) & (\lambda y.M)[x := N] \equiv \lambda y.(M[x := N]) \\ (4) & (M_1 M_2)[x := N] \equiv (M_1[x := N])(M_2[x := N]) \end{array}$$

The variable capture problem has disappeared! A free occurrence of y in N in the context

$$(\lambda y.M)[x := N]$$

would breach the variable convention, so we would have to use a different representative of the α-equivalence class of $\lambda y.M$ (this is precisely what rule 5 in the classical approach makes explicit).

In the following, we will adopt this convention of substitution because it is easier to work with (there are less cases to consider in proofs). As an illustration of its use, we return to our first example:

$$(\lambda xy.xy)y \equiv (\lambda xz.xz)y = \lambda z.yz$$

We now present a result which allows us to reorder substitutions:

Lemma 3.18 (Substitution Lemma)
If $x \not\equiv y$ and $x \notin FV(L)$, then:

$$M[x := N][y := L] \equiv M[y := L][x := N[y := L]]$$

Proof
By induction on the structure of M. □

A major property of functional languages is *referential transparency*, which allows equals to be substituted with equals. The formulation of this property, also referred to as Leibniz' Law, is:

Lemma 3.19 (Referential Transparency)
Let $C[]$ be a context. Then:

$$N = N' \;\Rightarrow\; C[N] = C[N']$$

Proof
By induction on the structure of $C[]$. □

3.2.4 Extensionality

Convertibility covers intensional equality; two terms are equal if they encode the same algorithm in some sense. This does not equate some terms which we might naturally think of as equal. For example, $\lambda x.Mx$ and M both yield the same result MN when applied to a term N (provided that x does not occur free in M). According to the classical notion of extensional equality $\lambda x.Mx$ is equal to M, but the formula $\lambda x.Mx = M$ is not a theorem of $\boldsymbol{\lambda}$.

There are two ways we can extend $\boldsymbol{\lambda}$ to make the above formula a theorem. Firstly, we can add a new rule, giving the new theory $\boldsymbol{\lambda + ext}$:

$$(ext) \quad Mx = Nx \ \Rightarrow \ M = N, \ x \notin FV(MN)$$

Alternatively, we can add a new axiom, giving the theory $\boldsymbol{\lambda\eta}$ (as proposed by Church):

$$(\eta) \quad \lambda x.Mx = M, \ x \notin FV(M)$$

In fact, we have the following result:

Lemma 3.20
$\boldsymbol{\lambda + ext}$ and $\boldsymbol{\lambda\eta}$ are equivalent.

Proof
First we show $\boldsymbol{\lambda + ext} \vdash \eta$:
Indeed, $(\lambda x.Mx)x = Mx$ by (β). Hence if $x \notin FV(M)$, then $\lambda x.Mx = M$ by (ext).

Now we show $\boldsymbol{\lambda\eta} \vdash ext$:
Let $Mx = Nx$ with $x \notin FV(MN)$. Then $\lambda x.Mx = \lambda x.Nx$ by (ξ) and $M = N$ by (η) twice. $\square$

The calculus extended with (ext) or (η) is called the $\lambda\eta$-calculus. Practically, from the point of view of functional programming, the $\lambda\eta$-calculus is not as important as the λ-calculus since the axiom (η) is not normally implemented. The term $\lambda x.Mx$ is a weak head normal form and is thus distinguishable from M; the former is a value, the latter may lead to a non-terminating computation. However, the $\lambda\eta$-calculus does have some theoretical significance which we shall return to later.

3.2.5 Consistency and completeness

For a theory to be useful, there must be some theorems and not all closed formulas should be theorems. The former is satisfied provided that the theory has at least one axiom. The latter is slightly trickier and is quite a fragile property; a theory which satisfies this property is called *consistent*.

To formalize the concept, we start with some definitions:

Definition 3.21
An equation *is a formula of the form $M = N$ with $M, N \in \Lambda$.*
An equation is closed *if $M, N \in \Lambda^0$.*

Definition 3.22 (Consistency)
If $\mathcal{T}$ is a theory then $\mathcal{T}$ is consistent*, written $Con(\mathcal{T})$, if it does not prove every closed equation.*
If $\mathcal{E}$ is a set of equations then $\boldsymbol{\lambda} + \mathcal{E}$ is formed by adding the equations of $\mathcal{E}$ as axioms to $\boldsymbol{\lambda}$. $\mathcal{E}$ is consistent, also written $Con(\mathcal{E})$, if $Con(\boldsymbol{\lambda} + \mathcal{E})$.

Both of the theories that we have dealt with, $\boldsymbol{\lambda}$ and $\boldsymbol{\lambda\eta}$, are consistent (see corollary 3.53).

The property of consistency is fairly fragile; it can be disturbed by adding a single equation. This motivates the following definition.

Definition 3.23 (Incompatibility)
$M, N \in \Lambda$ are incompatible*, written $M\#N$, if $\neg Con(M = N)$.*

Example 3.24
$x(yz)\#(xy)z$, i.e. application is not associative.

Proof
Assume that $x(yz) = (xy)z$, then:

$$\begin{array}{lllll}
 & \lambda xyz.x(yz) & = & \lambda xyz.(xy)z & \text{by } \xi \text{ three times} \\
\Rightarrow & (\lambda xyz.x(yz))MNO & = & (\lambda xyz.(xy)z)MNO & \text{for arbitrary M, N, O} \\
\Rightarrow & M(NO) & = & (MN)O & \text{by } \beta \text{ three times} \\
\Rightarrow & LM(NO) & = & L(MN)O & \text{for arbitrary L} \\
\Rightarrow & M & = & MN & \text{choosing } L \equiv \lambda xy.x \\
\Rightarrow & \lambda x.x & = & N & \text{choosing } M \equiv \lambda x.x
\end{array}$$

Since N was arbitrary, all terms are equal to $\lambda x.x$. □

Before we turn to the notion of completeness, we need some more definitions:

Definition 3.25 (Normal Forms)
*If $M \in \Lambda$, then M is a β-*normal form*, written β-nf or nf, if M has no subterms of the form $(\lambda x.R)S$.*
M has *a β-nf if there exists an N such that $M = N$ and N is a β-nf.*

Example 3.26
Consider the following λ-terms:

(i) $\lambda x.x$ is a nf
(ii) $(\lambda xy.x)(\lambda x.x)$ has $\lambda yx.x$ as a nf
(iii) $(\lambda x.xx)(\lambda x.xx)$ does not have a nf

Definition 3.27
If $M \in \Lambda$, then M is a $\beta\eta$-nf if M has no subterms of the form $(\lambda x.R)S$ or $\lambda x.Tx$ with $x \notin FV(T)$.
M has *a $\beta\eta$-nf if there exists an N such that $M = N$ and N is a $\beta\eta$-nf.*

We now state several facts about normal forms:

Fact 3.28
M has a β-nf. $\Leftrightarrow$ M has a $\beta\eta$-nf.

Proof
See corollary 15.1.5 in [3]. □

Fact 3.29
If $M, N \in \Lambda$ are distinct β-nfs then $\boldsymbol{\lambda} \nvdash M = N$.
(Similarly, if $M, N \in \Lambda$ are distinct $\beta\eta$-nfs then $\boldsymbol{\lambda\eta} \nvdash M = N$.)

Proof
See theorem 3.2.10 (3.3.11) in [3]. □

Fact 3.30
If $M, N \in \Lambda$ are distinct $\beta\eta$-nfs then $M \# N$.

Proof
See corollary 10.4.3 in [3]. □

The use of $\beta\eta$-nfs in fact 3.30 is essential; $\lambda x.yx$ and y are distinct β-nfs but not incompatible – they are η-equivalent.

We can now proof the completeness of $\boldsymbol{\lambda\eta}$ for terms having a nf:

Theorem 3.31 (Completeness)
Suppose M and N have nfs.
Then either $\boldsymbol{\lambda\eta} \vdash M = N$ or $\boldsymbol{\lambda\eta} + (M = N)$ is inconsistent.

Proof
By fact 3.28 the terms M and N have $\beta\eta$-nfs, say M' and N'.

Case $M' \equiv N'$.
Then $\boldsymbol{\lambda\eta} \vdash M = M' \equiv N' = N$.

Case $M' \not\equiv N'$.
Then by fact 3.30 one has $M' \# N'$, hence $\boldsymbol{\lambda\eta} + (M = N)$ is inconsistent. □

3.2.6 Summary

The convertibility relation, being an equivalence relation, partitions the class of λ-terms. When dealing with equivalence classes, it is convenient to use canonical representatives. The obvious representatives to use in our study of the λ-calculus are the normal forms. (Take care: What about terms, such as $\Omega \equiv (\lambda x.xx)(\lambda x.xx)$, which have no normal form? If we equate them we get an inconsistent theory; see later.) In the next section, we will study the notion of reduction, in which terms are successively simplified towards normal form. The computational motivation for this study is the correspondence between the process of reduction and the familiar notion of evaluation used in functional programming languages.

3.3 Reduction

Convertibility is a symmetric relation and therefore does not correspond very closely to our intuitions about computing with terms. In this section we study reduction, a relation on terms which better fits our intuitions. Having introduced the basic concepts, we present the Church-Rosser Theorem; this is a central theorem in the λ-calculus and we study it in some detail. The other key theorem is the Standardization Theorem; before presenting this we require the notion of *head normal form*.

3.3.1 Introduction

We have suggested that normal forms should be used as canonical representatives for the convertibility equivalence classes. In a more computational view normal forms are regarded as "results" produced from λ-term "programs". This view is justified by observing that the calculation of the β-normal form of a term consists in the repeated use of the axiom (β); subterms of the form

$(\lambda x.R)S$ – hencefort called β-redexes (*red*ucible *ex*pressions) – are replaced by $R[x := S]$. We have already identified this process with function application in functional languages. We will pursue this view further.

We motivate the following material by considering again the example of the factorial function. We saw that the factorial function can be λ-defined by *fac* which is the fixed point of a term:

$$\mathit{fac} = (\lambda fn.\mathit{if}\,(\mathit{zero}\; n)\mathit{1}\,(\times n(f\,(\mathit{pred}\; n))))\,\mathit{fac}$$

Following the construction used in the proof of the Fixed Point Theorem, we obtain a term $\mathbf{Y}$ which computes the fixed point of a given term:

$$\mathbf{Y} \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

The factorial of 0, for example, may thus be translated into the following λ-term program:

$$(\mathbf{Y}(\lambda fn.\mathit{if}\,(\mathit{zero}\; n)\mathit{1}\,(\times n(f\,(\mathit{pred}\; n)))))\;\mathit{0}$$

Consider now the normal form of this term. We can produce the β-normal form by repeatedly applying the axiom (β); in outline, we perform the following steps[4]:

$$\begin{array}{rcl}
(\mathbf{Y}(\lambda fn.\mathit{if}\ldots))\;\mathit{0} & = & (\lambda fn.\mathit{if}\ldots)(\mathbf{Y}\ldots)\;\mathit{0} \\
 & = & (\lambda n.\mathit{if}\,(\mathit{zero}\; n)\mathit{1}\,(\times n((\mathbf{Y}\ldots)(\mathit{pred}\; n))))\;\mathit{0} \\
 & = & \mathit{if}\,(\mathit{zero}\;\mathit{0})\mathit{1}\,(\times\mathit{0}((\mathbf{Y}\ldots)(\mathit{pred}\;\mathit{0}))) \\
 & = & \mathit{if}\;\mathit{true}\;\mathit{1}\ldots \\
 & = & \mathit{1}
\end{array}$$

Throughout this derivation we have used the convertibility relation introduced in the last section. Convertibility is symmetrical, indeed it is an equivalence relation, but we have used it in a non-symmetrical way. The factorial of 0 is 1, so we are happy to consider the term *1* as the result of the above computation, but it is a little harder to see the original program as the value of *1*. The latter view would associate an infinite set of values with terms such as *1*.

[4] In the first step we used a defining property of fixed point combinators such as $\mathbf{Y}$:
$$\mathbf{Y}F = F(\mathbf{Y}F)$$

In this section, we will study some new relations between λ-terms, notably $\rightarrow_\beta$ (one-step β-reduction) and $\twoheadrightarrow_\beta$ (β-reduction), the reflexive, transitive closure of $\rightarrow_\beta$. We will see that $\twoheadrightarrow_\beta$ is closely related to $=$ but is not symmetric; each $=$ in the above derivation, other than the one in the first step, could be replaced by $\rightarrow_\beta$.

In performing reduction, we are faced with a problem of strategy. For example, after the first step of the above derivation there are two subterms of the form $(\lambda x.R)S$:

$$(\lambda fn.\mathit{if}\ \ldots)(\mathbf{Y}\ldots)$$

and

$$(\mathbf{Y}\ldots)$$

We chose to reduce the first term but consider what would happen if we consistently chose to reduce the subterm involving the fixed point combinator: we would never get to the result, we would merely construct a larger and larger term!

Making the wrong choice is not always so catastrophic, for example:

$$\begin{aligned}(\lambda xy./(+\,x\,y)\mathit{2})((\lambda z.+z\,\mathit{1})\mathit{4})\mathit{6} &\rightarrow_\beta (\lambda y./(+((\lambda z.+z\,\mathit{1})\mathit{4})y)\mathit{2})\mathit{6}\\ &\rightarrow_\beta /(+((\lambda z.+z\,\mathit{1})\mathit{4})\mathit{6})\mathit{2}\\ &\rightarrow_\beta /(+(+\,\mathit{4}\ \mathit{1})\mathit{6})\mathit{2}\end{aligned}$$

but also:

$$\begin{aligned}(\lambda xy./(+\,x\,y)\mathit{2})((\lambda z.+z\,\mathit{1})\mathit{4})\mathit{6} &\rightarrow_\beta (\lambda xy./(+\,x\,y)\mathit{2})(+\,\mathit{4}\ \mathit{1})\mathit{6}\\ &\rightarrow_\beta (\lambda y./(+(+\,\mathit{4}\ \mathit{1})y)\mathit{2})\mathit{6}\\ &\rightarrow_\beta /(+(+\,\mathit{4}\ \mathit{1})\mathit{6})\mathit{2}\end{aligned}$$

The above discussion should pose two questions in the reader's mind:

- Given a term and a number of reduction sequences from that term which all terminate in a normal form, is it possible that some of the sequences might terminate with different normal forms?

- Given that some choices of reduction strategy appear to be better than others in some situations (for example the bottomless pit of $(\mathbf{Y}\ldots)$), is there a best way of choosing what to do next?

The first question is closely related to the issue of *determinacy*; computationally, the question amounts to asking if we can get different results from a program depending on how we execute it. A corollary of the Church-Rosser Theorem, which we will present below, guarantees that the answer to this question is no.

The second question is less precisely formulated; the Standardization Theorem, also presented below, addresses the question by giving a reduction order which is guaranteed to terminate with normal form if any reduction sequence does, but if best is also meant to be optimal then the question is more complicated – see [14] for a more detailed discussion of this topic.

3.3.2 Notions of reduction

Reduction may be viewed as a special form of relation on λ-terms. Why special? Recall the discussion of the constraints on equality in section 3.2.2; it is reasonable to place some of the same constraints on reduction. For example, if one term reduces to another, then it should do so in any context. On the other hand, bearing in mind our earlier discussion, we should not expect a reduction relation to be an equivalence. We make the following definitions:

Definition 3.32
$R \subseteq \Lambda \times \Lambda$ is compatible *if*

$$(M, M') \in R \;\Rightarrow\; (C[M], C[M']) \in R$$

for all $M, M' \in \Lambda$ and all contexts $C[]$ with one hole.

Definition 3.33
$R \subseteq \Lambda \times \Lambda$ is an equality *relation if it is a compatible equivalence relation.*

Definition 3.34
$R \subseteq \Lambda \times \Lambda$ is a reduction *relation if it is compatible, reflexive and transitive.*

We now turn to describing how a one-step reduction relation, a reduction relation and an equality relation can be defined from a given relation. The basic technique is to take closures of the given set; to make the set satisfy some property, we add elements, in an appropriate way, until the set does satisfy the property. For example, consider a subset of $A \times A$ for some set A; to make the subset a reflexive relation on A, we add the pair (a, a) for all $a \in A$ – this generates the reflexive closure of the original subset.

We call an arbitrary relation on Λ, a *notion of reduction*. For example, the notion of reduction that we will be particularly interested in is:

$$\beta = \{((\lambda x.M)N, M[x := N]) \mid M, N \in \Lambda\}$$

Given two notions of reduction, R_1 and R_2, we sometimes write R_1R_2 for $R_1 \cup R_2$; notably in the case that R_1 is β and R_2 is η, we write $\beta\eta$.

The one-step reduction relation induced by some notion of reduction R, written $\rightarrow_R$, is the compatible closure of R. The closure is explicitly constructed as follows:

Definition 3.35 (One-step R-reduction)

$$(M, N) \in R \;\Rightarrow\; M \rightarrow_R N$$

$$\begin{array}{lcl} M \rightarrow_R N & \Rightarrow & MZ \rightarrow_R NZ \\ M \rightarrow_R N & \Rightarrow & ZM \rightarrow_R ZN \\ M \rightarrow_R N & \Rightarrow & \lambda x.M \rightarrow_R \lambda x.N \end{array}$$

The notation $M \rightarrow_R N$ should be read as "M R-reduces to N in one step" or "N is an R-reduct of M". We have already seen the relation $\rightarrow_\beta$, in this case we often say that "M reduces to N in one step" or "N is an reduct of M".

The reduction relation, written $\twoheadrightarrow_R$, is the reflexive, transitive closure of the one-step reduction relation. While, as its name implies, the one-step reduction relation allows a single step of reduction, the reduction relation allows many (including zero – allowed by reflexivity). The reflexive transitive closure is defined formally as follows:

Definition 3.36 (R-reduction)

$$M \rightarrow_R N \;\Rightarrow\; M \twoheadrightarrow_R N$$

$$M \twoheadrightarrow_R M$$
$$M \twoheadrightarrow_R N,\, N \twoheadrightarrow_R L \;\Rightarrow\; M \rightarrow_R L$$

For the notation $M \twoheadrightarrow_R N$, read "M R-reduces to N".

Finally, we consider R-equality (also called R-convertibility), written $=_R$. This is the equivalence relation generated by $\twoheadrightarrow_R$. To generate the equivalence relation, we must take the symmetric closure of the relation. But care must be taken; given some reflexive, transitive relation, the symmetric closure is lo longer transitive in general. Thus, having taken the symmetric closure, it is necessary to take the transitive closure again:

Definition 3.37 (R-convertibility)

$$M \twoheadrightarrow_R N \;\Rightarrow\; M =_R N$$

$$M =_R N \;\Rightarrow\; N =_R M$$
$$M =_R N,\, N =_R L \;\Rightarrow\; M =_R L$$

For the notation $M =_R N$, read "M is R-convertible to N".

We have the following result for these relations:

Proposition 3.38
$\rightarrow_R$, $\twoheadrightarrow_R$ and $=_R$ are all compatible.

Proof
For $\rightarrow_R$, the proof is immediate from the definition.
For $\twoheadrightarrow_R$ and $=_R$, the proof is by induction on the definition. Since we have not seen this kind of induction before, we illustrate the proof for $\twoheadrightarrow_R$:

Basis:
$M \twoheadrightarrow_R N$ because $M \rightarrow_R N$:
Since $\rightarrow_R$ is compatible we have $C[M] \rightarrow_R C[N]$ and thus $C[M] \twoheadrightarrow_R C[N]$ by definition.

$M \twoheadrightarrow_R N$ because $M \equiv N$:
Trivial.

Inductive Step:
$M \twoheadrightarrow_R N$ because $M \twoheadrightarrow_R L$ and $L \twoheadrightarrow_R N$:

By the inductive hypothesis twice we have $C[M] \twoheadrightarrow_R C[L]$ and $C[L] \twoheadrightarrow_R C[N]$, and thus $C[M] \twoheadrightarrow_R C[N]$ by definition. □

We now turn to the notion of *redex* and *normal form*:

Definition 3.39
*An R-*redex *is a term M such that $(M, N) \in R$ for some term N; in this case N is called an R-*contractum *of M.*
*A term M is called an R-*normal form *(R-*nf*) if it does not contain any R-redex; a term N is an R-nf of M if N is an R-nf and $M =_R N$.*

In the following we will need a result which gives some constraints on the form of terms which are related by the one-step reduction relation:

Proposition 3.40
$M \to_R N \;\Leftrightarrow\; M \equiv C[P]$, $N \equiv C[Q]$ and $(P, Q) \in R$ for some $P, Q \in \Lambda$ where $C[]$ has one hole.

Proof
$(\Rightarrow)$
By induction on the definition of $\to_R$:

$M \to_R N$ because $(M, N) \in R$:
Trivial with $C[] = []$.

$M \to_R N$ because $M \equiv SZ$, $N \equiv TZ$ and $S \to_R T$:
The induction hypothesis applies to the reduction from S to T, and thus there is a context $C[]$ such that $S \equiv C[P]$ and $T \equiv C[Q]$ with $(P, Q) \in R$. Thus we can take the context $C[]Z$ to complete the proof.

The other cases are similar.

$(\Leftarrow)$
By the compatibility of $\to_R$. □

This proposition motivates the following definition:

Definition 3.41
Let $M \equiv C[\Delta]$ where $C[]$ has one hole. Write

$$M \xrightarrow{\Delta}_R N$$

if Δ is an R-redex with contractum Δ' and $N \equiv C[\Delta']$.

A corollary of the above proposition gives us some, not unexpected, results relating reduction and normal forms:

Corollary 3.42
Let M be an R-nf, then:

(1) *There is no N such that $M \to_R N$.*
(2) $M \twoheadrightarrow_R N \;\Rightarrow\; M \equiv N$

Proof
(1) By the above proposition and the definition of R-nf.
(2) By (1), since $\twoheadrightarrow_R$ is the reflexive, transitive closure of $\to_R$. $\square$

We are now ready to present the Church-Rosser Theorem.

3.3.3 The Church-Rosser Theorem

We start by introducing the *diamond property*:

Definition 3.43 (Diamond Property)
Let $\rhd$ be a binary relation on Λ, then $\rhd$ satisfies the diamond property*, written $\rhd \models \Diamond$, if for all M, M_1, M_2:*

$$M \rhd M_1 \wedge M \rhd M_2 \;\Rightarrow\; \exists M_3 : M_1 \rhd M_3 \wedge M_2 \rhd M_3$$

If there are two diverging $\rhd$-steps from some term and $\rhd$ satisfies the diamond property, then there is always a way to converge again.

Definition 3.44 (Church-Rosser)
A notion of reduction R is said to be Church-Rosser (CR) *if $\twoheadrightarrow_R \models \Diamond$.*

We then have the following theorem:

Theorem 3.45 (Church-Rosser Theorem)
Let R be CR, then:

$$M =_R N \;\Rightarrow\; \exists Z : M \twoheadrightarrow_R Z \wedge N \twoheadrightarrow_R Z$$

Proof
By induction on the definition of $=_R$:

$M =_R N$ because $M \twoheadrightarrow_R N$:
Choose $Z \equiv N$.

$M =_R N$ because $N =_R M$:
Trivial.

$M =_R N$ because $M =_R L$ and $L =_R N$:
By the inductive hypothesis twice we have:

$$\exists Z_1 : M \twoheadrightarrow_R Z_1 \wedge \underline{L \twoheadrightarrow_R Z_1} \quad \text{and} \quad \exists Z_2 : \underline{L \twoheadrightarrow_R Z_2} \wedge N \twoheadrightarrow_R Z_2$$

Therefore, by the fact that R is CR:

$$\exists Z_3 : Z_1 \twoheadrightarrow_R Z_3 \wedge Z_2 \twoheadrightarrow_R Z_3$$

By transitivity of $\twoheadrightarrow_R$ we get the desired result:

$$M \twoheadrightarrow_R Z_3 \quad \text{and} \quad N \twoheadrightarrow_R Z_3$$ □

The Church-Rosser Theorem has a useful corollary:

Corollary 3.46
Let R be CR, then:

(1) *If N is an R-nf of M then $M \twoheadrightarrow_R N$.*
(2) *A term can have at most one R-nf.*

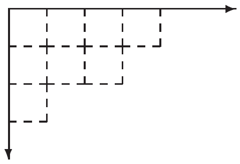
Proof
(1) Let $M =_R N$, where N is a R-nf. Then by the Church-Rosser Theorem there is a Z such that $M \twoheadrightarrow_R Z$ and $N \twoheadrightarrow_R Z$. But since N is an R-nf, we have $N \equiv Z$ by corollary 3.42 (2).
(2) Let N_1 and N_2 both be R-nfs of M. Then $M =_R N_1$ and $M =_R N_2$ and so $N_1 =_R N_2$. By the Church-Rosser Theorem there is a Z to which both R-nfs reduce, thus $N_1 \equiv Z \equiv N_2$ by corollary 3.42 (2). □

Therefore, if we can demonstrate that β is CR, we will have answered our first question. In fact the above corollary tells us more; not only does it guarantee unicity of normal forms, it also guarantees that if a term has a normal form then it will be possible to reduce the term to it.

To demonstrate that β is CR we must show that $\twoheadrightarrow_\beta \models \diamond$. First some notation: if $\rhd$ is some binary relation on a set A then we write $\rhd^*$ for its transitive closure and we have:

$$\rhd \models \diamond \;\Rightarrow\; \rhd^* \models \diamond$$

This result can be justified by consideration of the following diagram:



The axes represent diverging reductions, the side of each small square represents a single step. The small internal squares, some of which are shown with dashed lines, can all be completed by appealing to the CR property of $\rhd$. So if we could show that the reflexive closure of $\to_\beta$ satisfied the diamond property, we would have finished. Alas, this approach is too simple; consider the following term:

$$(\lambda x.xx)((\lambda x.x)(\lambda x.x))$$

We have the following two diverging reductions:

$$(\lambda x.xx)((\lambda x.x)(\lambda x.x)) \to_\beta ((\lambda x.x)(\lambda x.x))((\lambda x.x)(\lambda x.x))$$
$$(\lambda x.xx)((\lambda x.x)(\lambda x.x)) \to_\beta (\lambda x.xx)(\lambda x.x)$$

In the second case there is only one redex after the next reduction:

$$(\lambda x.xx)(\lambda x.x) \to_\beta (\lambda x.x)(\lambda x.x)$$

But there is no way of converging to this term by one step in the first case. So we cannot directly apply the above result to show that β is CR. The approach that we will take involves introducing a new relation which is "sandwiched" by the reflexive closure of $\to_\beta$ and $\twoheadrightarrow_\beta$ and which has $\twoheadrightarrow_\beta$ as its transitive closure.

We define the relation $\twoheadrightarrow_1$. This relation is reflexive and allows multiple $\to_\beta$ steps in one big step. We read $M \twoheadrightarrow_1 N$ as "M grand reduces to N".

Definition 3.47 (Grand Reduction)

$M \twoheadrightarrow_1 M$

$M \twoheadrightarrow_1 M' \;\Rightarrow\; \lambda x.M \twoheadrightarrow_1 \lambda x.M'$
$M \twoheadrightarrow_1 M',\, N \twoheadrightarrow_1 N' \;\Rightarrow\; MN \twoheadrightarrow_1 M'N'$
$M \twoheadrightarrow_1 M',\, N \twoheadrightarrow_1 N' \;\Rightarrow\; (\lambda x.M)N \twoheadrightarrow_1 M'[x := N']$

It is easy to see that $\to_\beta$ is included in $\twoheadrightarrow_1$. In the above example both of the diverging $\to_\beta$ steps are also $\twoheadrightarrow_1$ steps. There are two more possible $\twoheadrightarrow_1$ steps from $(\lambda x.xx)((\lambda x.x)(\lambda x.x))$: the first just states reflexivity and the second results in the term $(\lambda x.x)(\lambda x.x)$.

Evidence that $\twoheadrightarrow_1$ is weaker than $\twoheadrightarrow_\beta$ is furnished by the fact that:

$$(\lambda x.xx)((\lambda x.x)(\lambda x.x)) \twoheadrightarrow_\beta \lambda x.x$$

But the corresponding grand reduction requires at least two steps.

Lemma 3.48
$\twoheadrightarrow_1 \models \diamond$

Proof
By induction on the definition of $M \twoheadrightarrow_1 M_1$:
We show that for all M_2 such that $M \twoheadrightarrow_1 M_2$ there is an M_3 such that $M_1 \twoheadrightarrow_1 M_3$ and $M_2 \twoheadrightarrow_1 M_3$.

$M \equiv M_1$:
Choose $M_3 \equiv M_2$.

$M \equiv \lambda x.P$, $M_1 \equiv \lambda x.P'$ with $P \twoheadrightarrow_1 P'$:
By the definition of Grand Reduction M_2 must be of the form $M_2 \equiv \lambda x.P''$ with $P \twoheadrightarrow_1 P''$. By the induction hypothesis P' and P'' have a common reduct, say P'''. Choose $M_3 \equiv \lambda x.P'''$.

$M \equiv PQ$, $M_1 \equiv P'Q'$ with $P \twoheadrightarrow_1 P'$, $Q \twoheadrightarrow_1 Q'$:
By the definition of Grand Reduction M_2 can have two forms.

> $M_2 \equiv P''Q''$ with $P \twoheadrightarrow_1 P''$, $Q \twoheadrightarrow_1 Q''$:
> By the induction hypothesis P' and P'' (respectively Q' and Q'') have a common reduct, say P''' (respectively Q'''). Choose $M_3 \equiv P'''Q'''$.
>
> $M_2 \equiv R''[x := Q'']$, $P \equiv \lambda x.R$ with $R \twoheadrightarrow_1 R''$, $Q \twoheadrightarrow_1 Q''$:
> By the definition of Grand Reduction $P' \equiv \lambda x.R'$ with $R \twoheadrightarrow_1 R'$. By the induction hypothesis R' and R'' (respectively Q' and Q'') have a common reduct, say R''' (respectively Q'''). It follows that $M_1 \equiv (\lambda x.R')Q' \twoheadrightarrow_1 R'''[x := Q''']$ and $M_2 \equiv R''[x := Q''] \twoheadrightarrow_1 R'''[x := Q''']$ (the latter by induction on the definition of $R'' \twoheadrightarrow_1 R'''$).

$M \equiv (\lambda x.P)Q$, $M_1 \equiv P'[x := Q']$ with $P \twoheadrightarrow_1 P'$, $Q \twoheadrightarrow_1 Q'$:
Again M_2 can have two forms: $M_2 \equiv (\lambda x.P'')Q''$ or $M_2 \equiv P''[x := Q'']$. The appropriate argumentation is similar to the previous case. $\square$

Lemma 3.49
$\twoheadrightarrow_\beta$ is the transitive closure of $\twoheadrightarrow_1$.

Proof
It is easy to see that the reflexive closure of $\rightarrow_\beta$ is included in $\twoheadrightarrow_1$; it is also easy to see that $\twoheadrightarrow_1$ is included in $\twoheadrightarrow_\beta$. Since $\twoheadrightarrow_\beta$ is the transitive closure of the reflexive closure of $\rightarrow_\beta$, it is also the transitive closure of $\twoheadrightarrow_1$. □

Finally, we have the result that we have been waiting for:

Theorem 3.50
β is CR.

Proof
By the above lemmata:

Since $\twoheadrightarrow_1 \models \diamond$, we also have $\twoheadrightarrow_1^* \models \diamond$. That is, $\twoheadrightarrow_\beta \models \diamond$. □

Therefore, using the corollary of the Church-Rosser Theorem, we know that β-nfs are unique and that, if a term has a β-nf, it is possible to reduce the term to that β-nf. This allows us to prove the consistency of the theory $\boldsymbol{\lambda}$. But first, we need to prove the following two propositions:

Proposition 3.51

$$M =_\beta N \;\Leftrightarrow\; \lambda \vdash M = N$$

Proof
($\Rightarrow$) By induction on the definitions of the relations involved.
($\Leftarrow$) By induction on the length of the proof of $M = N$. □

Proposition 3.52
If $M, N \in \Lambda$ are distinct β-nfs, then $\boldsymbol{\lambda} \nvdash M = N$.

Proof
Suppose $\boldsymbol{\lambda} \vdash M = N$, i.e. $M =_\beta N$. Then M would have two β-nfs, M and N, which contradicts the corollary of the Church-Rosser Theorem. □

Corollary 3.53 (Consistency)
The theory $\boldsymbol{\lambda}$ is consistent.

Proof
By the previous proposition one has $\boldsymbol{\lambda} \nvdash M = N$ for distinct β-nfs M, N. □

We can also define a notion of reduction which is related to the extensional theory $\boldsymbol{\lambda\eta}$:

$$\eta = \{(\lambda x.Mx, M) \mid M \in \Lambda,\ x \notin FV(M)\}$$

We can define one-step η-reduction, η-reduction and η-convertibility in the standard way. It is then possible to address the question, if η is CR; however, a more interesting question is whether the derived notion $\beta\eta$ is CR. It turns out that both η and $\beta\eta$ are CR (see theorems 3.3.7 and 3.3.9 in [3]). The consistency of $\boldsymbol{\lambda\eta}$ follows by an argumentation similar to the case of $\boldsymbol{\lambda}$.

We now continue our study of the notion of reduction β. In the following, we omit the symbol β in $\rightarrow_\beta$, $\twoheadrightarrow_\beta$, $=_\beta$, β-redex, β-nf and simply write $\rightarrow$, $\twoheadrightarrow$, $=$, redex, nf.

3.3.4 Head normal forms

We now introduce a variant of normal form: *head normal form*. Head normal forms play an important role in the theory and they are much closer to the concept of “result” employed in lazy functional programming languages, as we shall see.

We start with some formal definitions:

Definition 3.54
$M \in \Lambda$ is a head normal form (hnf) *if M is of the form*

$$\lambda x_1 \ldots x_n.xM_1 \ldots M_m$$

where $n, m \geq 0$. In this case x is called the head variable.
If M is of the form

$$\lambda x_1 \ldots x_n.(\lambda x.M_0)M_1 \ldots M_m$$

where $n \geq 0$, $m \geq 1$ then $(\lambda x.M_0)M_1$ is called the head redex *of M.*
A redex is internal *if it is not a head redex.*

Example 3.55
The following λ-terms are head normal forms:

(*i*) x
(*ii*) xx, xy
(*iii*) $\lambda x.x$, $\lambda x.y$
(*iv*) $\lambda xy.x((\lambda z.z)y)$

Definition 3.56
If $M \stackrel{\Delta}{\to} N$ *and* Δ *is the head redex of* M*, then we write* $M \to_h N$ *and we also write* $\twoheadrightarrow_h$ *for the many-step reduction relation.*
Similarly, if $M \stackrel{\Delta}{\to} N$ *and* Δ *is an internal redex of* M*, then we write* $M \to_i N$ *and we also write* $\twoheadrightarrow_i$ *for the many-step reduction relation.*

Definition 3.57
If P *and* Q *are two redexes in an expression* M *and the first occurrence of* λ *in* P *is to the left of the first occurrence of* λ *in* Q *then we say that* P *is* to the left of Q*. If* P *is a redex in* M *and it is to the left of all the other redexes then* P *is the* leftmost redex*.*

Notice that the head redex of a term is always the leftmost redex, but not conversely. Consider:

$$\lambda xy.x((\lambda z.z)y)$$

This term is a hnf (i.e. it has no head redex) and so the the leftmost redex is the internal redex:

$$(\lambda z.z)y$$

Unlike normal forms, a term does not usually have a unique head normal form. For example:

$$(\lambda x.x(\mathbf{II}))z \quad \text{where} \quad \mathbf{I} \equiv \lambda x.x$$

has hnfs:

$$z(\mathbf{II}) \quad \text{and} \quad z\mathbf{I}$$

However, since any term has only one head redex, every term which has a hnf also has a *principal head normal form*. It is obtained by reducing the head redex at each stage until the hnf is reached. The principal head normal form of the above example is $z(\mathbf{II})$.

Head normal forms play a crucial role in the computability theory associated with the λ-calculus. There must be some way of coding partial functions – functions which are undefined for some elements in the domain. In domain theory, partial functions can be made into total functions by adding an undefined element $\perp$ to the co-domain. In the λ-calculus, the solution is to use a class of terms to represent the undefined element. The first attempt at solving this problem involved equating all the terms without normal form and then using some canonical representative. However, this leads to inconsistency because neither

$$\lambda x.x\mathbf{I}\Omega \quad \text{where} \quad \mathbf{I} \equiv \lambda x.x \quad \text{and} \quad \Omega \equiv (\lambda x.xx)(\lambda x.xx)$$

nor

$$\lambda x.x\mathbf{K}\Omega \quad \text{where} \quad \mathbf{K} \equiv \lambda xy.x$$

has a nf, but it is easy to show that $\lambda x.x\mathbf{I}\Omega \# \lambda x.x\mathbf{K}\Omega$:

$$\begin{aligned} \lambda x.x\mathbf{I}\Omega = \lambda x.x\mathbf{K}\Omega &\Rightarrow (\lambda x.x\mathbf{I}\Omega)\mathbf{K} = (\lambda x.x\mathbf{K}\Omega)\mathbf{K} \\ &\Rightarrow \mathbf{K}\mathbf{I}\Omega = \mathbf{K}\mathbf{K}\Omega \\ &\Rightarrow \mathbf{I} = \mathbf{K} \end{aligned}$$

in contradiction to $\mathbf{I}\#\mathbf{K}$ by proposition 3.52.

Instead, we equate all terms which do not have a head normal form (this is a proper subclass of the class of terms without normal form). This leads to no inconsistency, a canonical representative is Ω.

Practical lazy functional programming systems even stop some way short of hnf. Most lazy systems evaluate terms to *weak head normal form*.

Definition 3.58
$M \in \Lambda$ is a weak head normal form (whnf) *if M is of the form:*

$$xM_0 \dots M_n \quad \textit{or} \quad \lambda x.M'$$

That is, lazy systems do not evaluate inside λs.

3.3.5 The Standardization Theorem

In the following we will need to trace a redex through a reduction sequence. Of course the redex may be transformed through the sequence. For example, in the following sequence

$$
\begin{array}{rl}
 & (\lambda xy.\underline{(\lambda vw.xv)y})M)N \\
\to & (\lambda y.\underline{(\lambda vw.Mv)y})N \\
\to & \underline{(\lambda vw.Mv)N} \\
\to & \lambda w.MN
\end{array}
$$

the underlined redexes are clearly related though different. A descendant of a redex is called a *residual*. In the above example there is no residual of the original redex in the final term; it has been reduced in the final step. A formal treatment of residuals can be found in [31].

Definition 3.59
A reduction sequence

$$M_0 \overset{\Delta_0}{\to} M_1 \overset{\Delta_1}{\to} M_2 \overset{\Delta_2}{\to} \dots$$

is a standard reduction *if* $\forall i\ \forall j < i : \Delta_i$ *is not a residual of a redex to the left of* Δ_j *(in the term* M_j*).*

An alternative description of a standard reduction is as follows: after the reduction of a redex Δ, all the λs to the left of Δ are marked indelibly; no redex whose first λ is marked can be further reduced.

Definition 3.60
If there is a standard reduction from some term M *to some other term* N*, then we write* $M \twoheadrightarrow_s N$*.*

Notice that any head reduction sequence is a standard reduction sequence.

Before we can prove the Standardization Theorem, we must state a result which allows us to factor reductions into a sequence of head reductions followed by a sequence of internal reductions.

Proposition 3.61

$$M \twoheadrightarrow N \ \Rightarrow\ \exists Z : M \twoheadrightarrow_h Z \twoheadrightarrow_i N$$

Proof
See Lemma 11.4.6 in [3]. □

The details of the proof use some additional theory which is beyond the scope of this treatment; it relies on two observations:

- If $M \rightarrow_i M' \twoheadrightarrow_h N$, then there is an equivalent reduction sequence $M \twoheadrightarrow_h M'' \twoheadrightarrow_i N$.
- Any reduction sequence $M \twoheadrightarrow N$ is of the form:

$$M \twoheadrightarrow_h M_1 \twoheadrightarrow_i M_2 \twoheadrightarrow_h M_3 \twoheadrightarrow_i \ldots \twoheadrightarrow_i N$$

The intuition behind the first observation is the difference between call-by-value and call-by-name; an internal redex is an argument, so if we pre-evaluate it we only need to do it once, whereas if we do not pre-evaluate it then it may be duplicated. Since any reduction is either a head reduction or an internal reduction, the second observation is straightforward.

We then have the Standardization Theorem:

Theorem 3.62 (Standardization Theorem)

$$M \twoheadrightarrow N \;\Rightarrow\; M \twoheadrightarrow_s N$$

Proof
By induction on the length of N:
Let $M \twoheadrightarrow N$. By the previous result, we have:

$$\exists Z : M \twoheadrightarrow_h Z \twoheadrightarrow_i N$$

There are two cases to consider:

Case 1.
N is a variable, say x. Then $Z \equiv N$ and so $M \twoheadrightarrow_h N$. Since a head reduction is standard, we are done.

Case 2.
$N \equiv \lambda x_1 \ldots x_n.N_0 N_1 \ldots N_m$ with $n + m > 0$.
Then Z is of the form $\lambda x_1 \ldots x_n.Z_0 Z_1 \ldots Z_m$ with $Z_i \twoheadrightarrow N_i$ for $0 \leq i \leq m$.
By the induction hypothesis $Z_i \twoheadrightarrow_s N_i$ and the result follows. □

Thus we are able to answer the second question posed in the introduction: since we know from the corollary of the Church-Rosser Theorem that if M has a normal form N then $M \twoheadrightarrow N$, then by the Standardization Theorem we know that a standard reduction sequence will lead to the normal form.

3.3.6 Summary

In this section we have studied various aspects of reduction. We have seen how this concept is related to the usual notion of evaluation used in functional languages. The two key results are the Church-Rosser Theorem for β-reduction, which guarantees determinacy of the evaluation process, and the Standardization Theorem which identifies a canonical evaluation order for the reduction process.

4 An exploration of organization space

4.1 Notation

In this chapter we will use a nonconventional way of parenthesizing λ-terms due to Revesz [48].

Definition 4.1 ($\boldsymbol{\lambda}$-terms)
The class Λ of λ-terms is the least class satisfying the following conditions, where x denotes an arbitrary variable:

$$
\begin{array}{lll}
(1) & x \in \Lambda & \\
(2) & M \in \Lambda & \Rightarrow \quad \lambda x.M \in \Lambda \\
(3) & M, N \in \Lambda & \Rightarrow \quad (M)N \in \Lambda
\end{array}
$$

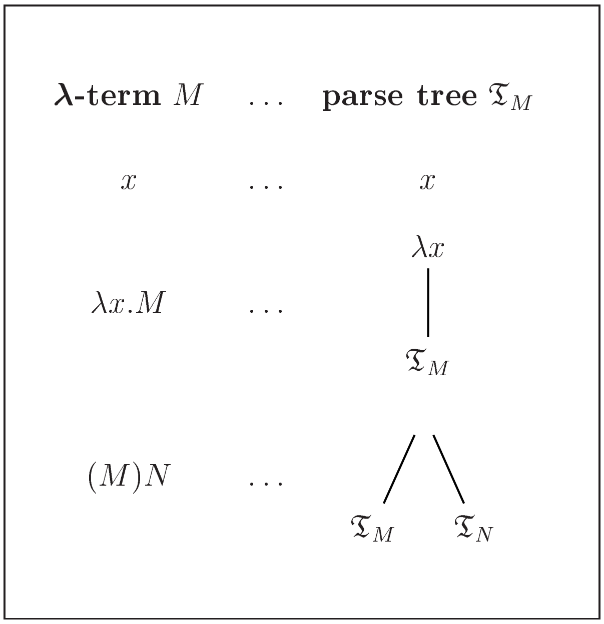
Example 4.2
The following are λ-terms:

$$
\begin{array}{rl}
(i) & \lambda x.x \\
(ii) & (x)y \\
(iii) & \lambda x.(y)x \\
(iv) & \lambda x.\lambda y.((y)x)\lambda z.z
\end{array}
$$

The traditional way of putting the arguments of a function between parentheses has been suggested by the process of function evaluation. In the so-called applicative order of evaluation, the argument of a function is computed before application. The conventional λ-notation has already departed from the traditional notation by using (MN) instead of $M(N)$. Here we go one step further by using $(M)N$, which reflects the so-called normal order evaluation strategy, where the function itself is analyzed before the argument.

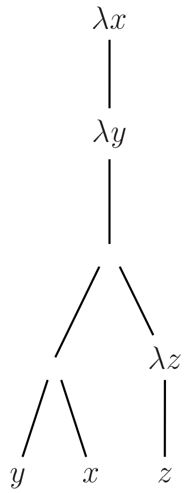
However, the main reason for using this notation is its simplicity. The conventional λ-notation often requires the use of parentheses both in the function and in the argument part of a λ-term. That may become quite confusing when dealing with complicated λ-terms.

Our syntax also represents a context-free grammar for λ-terms, which is very useful for the development of an efficient predictive parser. Every application starts with a left parenthesis, and every abstraction starts with a λ. Every left parenthesis can thus be treated as a prefix application operator with the corresponding right parenthesis being just a delimiter. The syntactic structure of a λ-term is defined by its parse tree with respect to the grammar. This grammar is unambiguous, hence, every λ-term has a unique parse tree. The parsing mechanism can be summarized as follows:



The subtrees of the parse tree correspond to the subterms of the given λ-term. For an illustration of the mechanism, consider the following example:

$\lambda x.\lambda y.((y)x)\lambda z.z$ ...



4.2 An exhaustive enumeration of $\boldsymbol{\lambda}$-terms

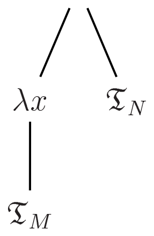
In our base model, molecules are viewed as symbolic representations of functions, or more precisely, as *closed λ-terms in normal form.* We use normal forms since they represent the values of functional applications. We use closed λ-terms since only bound variables encode possible functional actions. Closed λ-terms in normal form are abstractions.

A chemical reaction is viewed as the evaluation of a functional application: Two λ-terms M, N interact by forming the redex $(M)N \equiv (\lambda x.M')N$, such that M assumes the role of a function and is applied to N, which assumes the role of an argument. The following β-reduction of $(M)N$ to normal form yields the value of the interaction.

To get an impression of the variety of functions expressible in λ-calculus, we calculate the number of closed λ-terms in normal form which have a parse tree of a certain depth. The calculation involves a complicated recursion on the structure of parse trees. We begin with the following observations:

- As mentioned above, a closed λ-term in normal form starts with an abstraction. Hence, its parse tree is of the form:

- Additionally, a λ-term in normal form does not contain any redex, which implies that its parse tree does not contain any subtree of the form:



  That is, the left subtree of a branching must not start with an abstraction.

Now let $\mathfrak{T}_d^n$ denote a parse tree (of a closed λ-term in normal form) with n prefix abstractions followed by a subtree $\mathfrak{S}_d^n$ of depth d:

$$\mathfrak{T}_d^n \quad \equiv \quad \begin{array}{c} \lambda x_1 \\ | \\ \lambda x_2 \\ | \\ \vdots \\ \lambda x_n \\ | \\ \mathfrak{S}_d^n \end{array}$$

$\mathfrak{S}_d^n$ may signify a variable ($d = 0$), or start with an abstraction or a branching ($d \geq 1$). Consequently, $\mathfrak{T}_d^n$ starts with at least n abstractions; its total depth is $n + d$.

If the subtree itself does not start with an abstraction, we denote the parse tree by $\tilde{\mathfrak{T}}_d^n$ and the subtree by $\tilde{\mathfrak{S}}_d^n$. Consequently, $\tilde{\mathfrak{T}}_d^n$ starts with exactly n abstractions.

What is the number T_d^n of parse trees $\mathfrak{T}_d^n$ (and subtrees $\mathfrak{S}_d^n$ respectively)? in particular, we are interested in T_d^1, the number of closed λ-terms in normal form which have a parse tree of depth $1 + d$.

To begin with, we calculate the number $\tilde{T}_d^n$ of parse trees $\tilde{\mathfrak{T}}_d^n$ (and subtrees $\tilde{\mathfrak{S}}_d^n$ respectively): $\tilde{T}_d^n$ is the number of parse trees of depth $n+d$ with *exactly* n prefix abstractions; it is equal to the difference between T_d^n (the number of parse trees of depth $n+d$ with *at least* n prefix abstractions) and T_{d-1}^{n+1} (the number of parse trees of depth $n+d$ with at least $n+1$ prefix abstractions):

$$\tilde{T}_d^n = T_d^n - T_{d-1}^{n+1}$$

Our argumentation is valid for $d \geq 1$, that is for parse trees $\tilde{\mathfrak{T}}_d^n$ with subtrees $\tilde{\mathfrak{S}}_d^n$ of minimal depth 1.

For $d = 0$, a parse tree $\mathfrak{T}_0^n$ looks like:

$$
\mathfrak{T}_0^n \quad \equiv \quad
\begin{array}{c}
\lambda x_1 \\
| \\
\lambda x_2 \\
| \\
\vdots \\
\lambda x_n \\
| \\
\mathfrak{S}_0^n \equiv x_i \\
{\scriptstyle (1 \le i \le n)}
\end{array}
$$

The subtree $\mathfrak{S}_0^n$ signifies a variable (which implies that it does not start with an abstraction). Hence, we may also denote the parse tree by $\tilde{\mathfrak{T}}_0^n$ and the subtree by $\tilde{\mathfrak{S}}_0^n$.

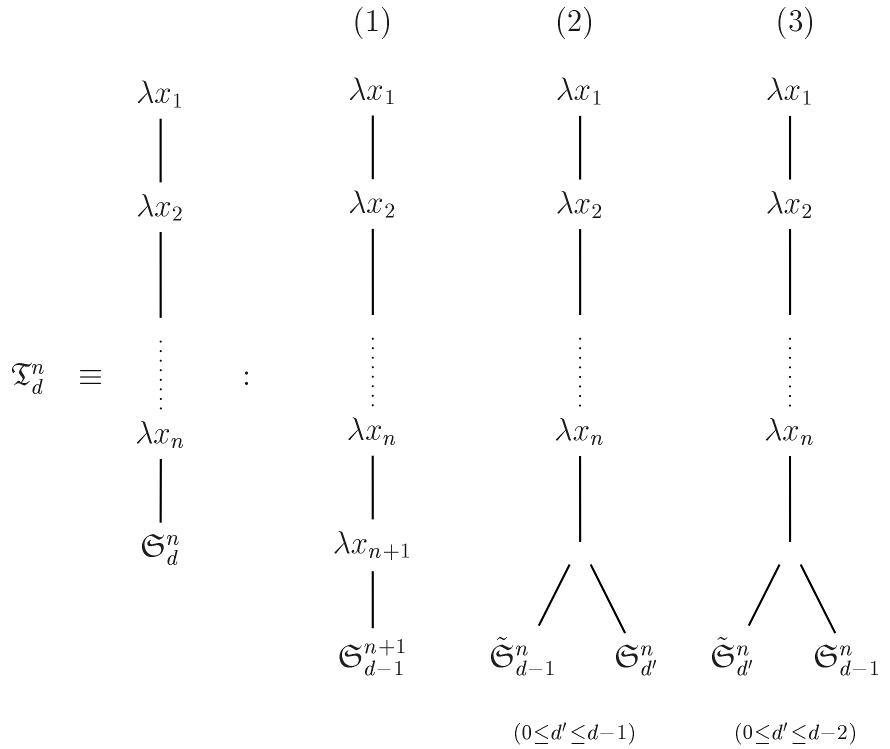
Obviously, $\tilde{T}_0^n$ (the number of parse trees of depth n with exactly n prefix abstractions) is equal to T_0^n (the number of parse trees of depth n with at least n prefix abstractions):

$$\tilde{T}_0^n = T_0^n$$

From the above diagram we can read the basis of our recursion: Since there are n prefix abstractions, the variable x_i may take n different values $x_1, x_2, \ldots, x_n$. That is:

$$T_0^n = n$$

We now turn to the recursion. The following diagram shows the three possible structures of a parse tree $\mathfrak{T}^n_d$:



The subtree $\mathfrak{S}^n_d$ of the parse tree $\mathfrak{T}^n_d$ has depth d. In each case it is specified by subtrees of smaller depth:

(1) $\mathfrak{S}^n_d$ consists of an abstraction followed by a subtree $\mathfrak{S}^{n+1}_{d-1}$ of depth $d-1$.

(2) $\mathfrak{S}^n_d$ is a branching tree. We choose the left subtree to be of depth $d-1$. Since the left subtree of a branching must not start with an abstraction, we have to use $\tilde{\mathfrak{S}}^n_{d-1}$. The depth of the right subtree $\mathfrak{S}^n_{d'}$ may vary from 0 to $d-1$.

(3) Similar to (2), except that we choose the right subtree to be of depth $d-1$. The depth of the left subtree may vary from 0 to $d-2$. (The case of both subtrees being of depth $d-1$ has already been dealt with.)

From the diagram we can read the following recursion:

$$T^n_d = T^{n+1}_{d-1} + \tilde{T}^n_{d-1} \sum_{d'=0}^{d-1} T^n_{d'} + \Big(\sum_{d'=0}^{d-2} \tilde{T}^n_{d'}\Big)\, T^n_{d-1}$$

The above argumentation is valid for $d \geq 1$, except that for $d = 1$ case (3) does not occur. Instead, we get the following equation:

$$T_1^n = T_0^{n+1} + \tilde{T}_0^n T_0^n$$

Using $\tilde{T}_0^n = T_0^n = n$ and $\tilde{T}_d^n = T_d^n - T_{d-1}^{n+1}$ for $d \geq 1$, the recursion may be restated as follows:

$$d = 0: \quad T_0^n = n$$

$$d = 1: \quad T_1^n = T_0^{n+1} + (T_0^n)^2$$

$$d = 2: \quad T_2^n = T_1^{n+1} + (T_1^n - T_0^{n+1})\,(T_0^n + T_1^n) + T_0^n T_1^n$$

$$d \geq 3: \quad T_d^n = T_{d-1}^{n+1} + (T_{d-1}^n - T_{d-2}^{n+1}) \sum_{d'=0}^{d-1} T_{d'}^n + (T_0^n + \sum_{d'=1}^{d-2} (T_{d'}^n - T_{d'-1}^{n+1}))\, T_{d-1}^n$$

As mentioned above, we are particularly interested in T_d^1, the number of closed λ-terms in normal form which have a parse tree of depth $1 + d$. To calculate T_d^1, we have to start our recursion with T_0^{1+d}. As an example, consider $d = 2$:

$T_0^3 = 3$

$T_0^2 = 2$
$T_1^2 = T_0^3 + (T_0^2)^2 = 7$

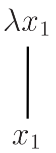
$T_0^1 = 1$
$T_1^1 = T_0^2 + (T_0^1)^2 = 3$
$T_2^1 = T_1^2 + (T_1^1 - T_0^2)(T_0^1 + T_1^1) + T_0^1 T_1^1 = 14$

In the following table, we list the values of T_d^1 for $d \leq 6$. (The recursion is approximately quadratic.)

$T_0^1 = 1$
$T_1^1 = 3$
$T_2^1 = 14$
$T_3^1 = 217$
$T_4^1 = 42\,325$
$T_5^1 = 1\,647\,464\,882$
$T_6^1 = 2\,686\,232\,153\,187\,877\,131$

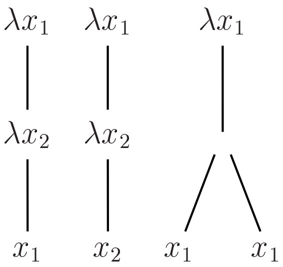
To conclude this section, we explicitly construct $\mathfrak{T}^1_d$ for $d \leq 2$:

$\boxed{\mathfrak{T}^1_0}$



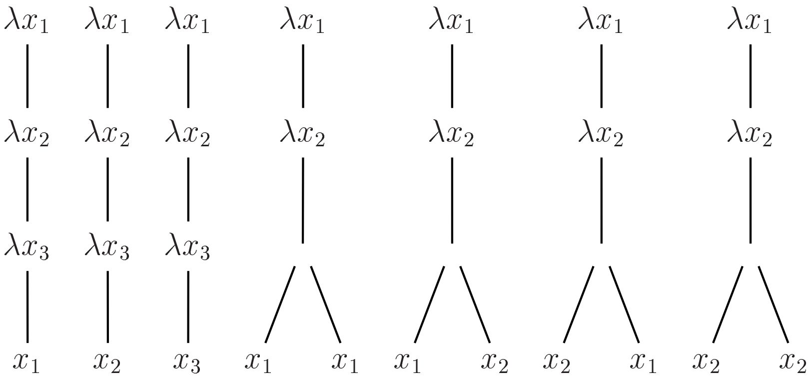
$\boxed{\mathfrak{T}^1_1}$

(1) (2)

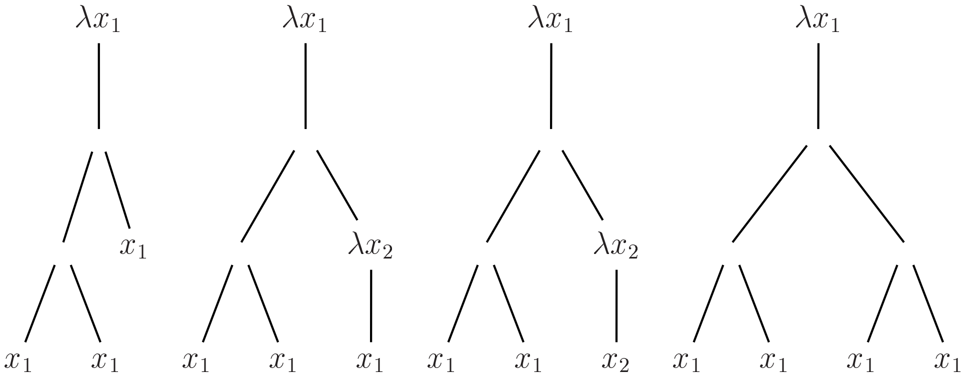


$\boxed{\mathfrak{T}^1_2}$

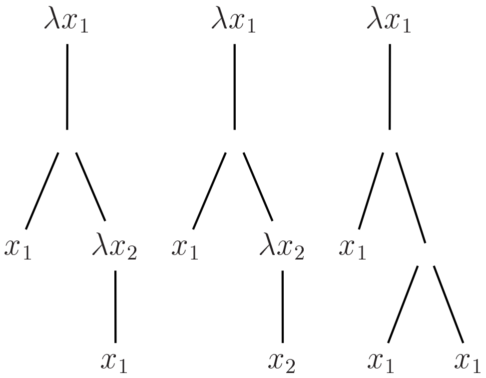
(1)



(2)



(3)



4.3 Iterated set actions

The base model has two core ingredients. As outlined in the previous section, the first consists in a rough analogy between, on the one side, molecules and their constructive interactions, and, on the other side, λ-terms and their mutual application. The second core ingredient consists in a simple constrained dynamics mimicking a well-stirred flow reactor. In the present context this means an ensemble of λ-terms meeting randomly and interacting by application.

For our computer experiments, we use the following procedure:

(0) **Fill reactor**

Produce an initial (multi-)set of n λ-terms. In the simplest case, generate n instances of a starting λ-term A.

(1) **Collide**

Choose two of the n λ-terms at random. Let the first λ-term be M, and the second one N.

(2) **React**

Compute the normal form L of the application $(M)N$ according to β-reduction.

If no normal form is reached within a certain number of reduction steps, the collision is elastic, and the procedure continues with step (1).

(3) **Add product**

Add the normal form L to the reactor. Choose one λ-term X at random, and eliminate it (compensatory outflow).

(4) **Apply flow (optional)**

Add one element of the initial set to the reactor (constant inflow). Choose one λ-term at random, and eliminate it (compensatory outflow).

(5) **Iterate**

The procedure continues with step (1).

The interacting λ-terms are treated here as "catalysts", remaining unchanged themselves, yet inducing change (generating a new term as the value of the application):

$$M + N \longrightarrow M + N + L$$

L is the normal form of $(M)N$, that is $(M)N \twoheadrightarrow_\beta L$. This way, the total number of λ-terms in the reactor is increased by one with each reactive collision.

Each time a new λ-term L has been produced, a randomly chosen one X is eliminated:

$$X \longrightarrow \{\}$$

The overall number n of λ-terms is thereby kept exactly constant. This means that each λ-term has a finite lifetime, even though it is not consumed at the moment of a reaction.

To motivate the constant inflow from the initial set, consider the following situation: All instances of the starting λ-term A are consumed after several iterations, and A is not reproduced by any reaction of two other λ-terms. In this case, it is necessary to add one instance of A with each iteration, to produce as many reactive consequences of A as possible.

Since any two λ-terms interact to produce a particular λ-term with a frequency proportional to their concentration, the above reaction scheme acts to favor convergence to a population of λ-terms whose relations of production yield λ-terms extant within the reactor. The motion in *object space* settles upon a set of objects that produce one another.

The reaction scheme, however, does obvious violence to the chemical analogy. Indeed, the present analogy and its instantiation through λ-calculus have a number of limitations which we discuss further in chapter 6.

We now switch off the dynamical aspect of the base model by considering only *iterated actions* within a set of *object species*. Doing so permits us to introduce concepts that prove useful in later studies. We proceed to define a few simple set valued iterated maps.

Definition 4.3
Let $\mathcal{M}$ and $\mathcal{N}$ be subsets of Λ. The action of $\mathcal{M}$ on $\mathcal{N}$ is the set $\mathcal{L} \subseteq \Lambda$ defined as:

$$\mathcal{L} = \mathcal{M} \circ \mathcal{N} \equiv \{L \,|\, (M)N \twoheadrightarrow_\beta L,\ (M,N) \in \mathcal{M} \times \mathcal{N},\ L \textit{ in nf}\,\}$$

Two set valued iterated maps are of special interest. The first is a quadratic map that replaces the old set at each iteration.

Definition 4.4
Let $\mathcal{A}$ be an initially given set. The map $\mathbf{m}$ induces the following iteration:

$$\begin{aligned} \mathcal{A}_0 &= \mathcal{A} \\ \mathcal{A}_{n+1} &= \mathbf{m}(\mathcal{A}_n) = \mathcal{A}_n \circ \mathcal{A}_n,\ n \geq 0 \end{aligned}$$

We also write:

$$\mathcal{A}_n = \mathbf{m}^n(\mathcal{A}) \equiv \underbrace{\mathbf{m}(\mathbf{m}(\ldots \mathbf{m}(}_{n}\mathcal{A})\ldots))$$

A variant of the quadratic map keeps the previous set. The resulting sequence of sets is non-decreasing.

Definition 4.5
Let $\mathcal{A}$ be an initially given set. The map $\mathbf{M}$ *induces the following iteration:*

$$\begin{aligned}\mathcal{A}_0 &= \mathcal{A}\\ \mathcal{A}_{n+1} &= \mathbf{M}(\mathcal{A}_n) = (\mathcal{A}_n \circ \mathcal{A}_n) \cup \mathcal{A}_n,\ n \geq 0\end{aligned}$$

We also write:

$$\mathcal{A}_n = \mathbf{M}^n(\mathcal{A}) \equiv \underbrace{\mathbf{M}(\mathbf{M}(\ldots \mathbf{M}(}_{n}\mathcal{A})\ldots))$$

This procedure turns out to be equivalent to refreshing the initial set at each iteration.

Proposition 4.6
Let $\mathbf{M}$ *and* $\tilde{\mathbf{M}}$ *induce the following iterations:*

$$\begin{aligned}\mathcal{A}_0 &= \mathcal{A}\\ \mathcal{A}_{n+1} &= \mathbf{M}(\mathcal{A}_n) = (\mathcal{A}_n \circ \mathcal{A}_n) \cup \mathcal{A}_n,\ n \geq 0\\ \tilde{\mathcal{A}}_0 &= \mathcal{A}\\ \tilde{\mathcal{A}}_{n+1} &= \tilde{\mathbf{M}}(\tilde{\mathcal{A}}_n) = (\tilde{\mathcal{A}}_n \circ \tilde{\mathcal{A}}_n) \cup \mathcal{A},\ n \geq 0\end{aligned}$$

Then, the iterations are equivalent. That is:

$$\mathcal{A}_n = \tilde{\mathcal{A}}_n$$

Proof
Basis:
$\mathcal{A}_0 = \tilde{\mathcal{A}}_0 = \mathcal{A}$

Inductive step:
$\mathcal{A}_{n+1} = (\mathcal{A}_n \circ \mathcal{A}_n) \cup \mathcal{A}_n \overset{\text{IH}}{=} (\mathcal{A}_n \circ \mathcal{A}_n) \cup \tilde{\mathcal{A}}_n = (\mathcal{A}_n \circ \mathcal{A}_n) \cup (\tilde{\mathcal{A}}_{n-1} \circ \tilde{\mathcal{A}}_{n-1}) \cup \mathcal{A} \overset{\text{IH}}{=} (\mathcal{A}_n \circ \mathcal{A}_n) \cup (\mathcal{A}_{n-1} \circ \mathcal{A}_{n-1}) \cup \mathcal{A} = (\mathcal{A}_n \circ \mathcal{A}_n) \cup \mathcal{A} \overset{\text{IH}}{=} (\tilde{\mathcal{A}}_n \circ \tilde{\mathcal{A}}_n) \cup \mathcal{A} = \tilde{\mathcal{A}}_{n+1}$ $\square$

The set $\mathbf{M}^n(\mathcal{A})$ contains all terms that result from n collision events among elements of $\mathcal{A}$, whereas it contains only particular terms among those that result from $2^n - 1$ collision events. This motivates the following definition.

Definition 4.7
Let $\mathcal{A}$ be an initially given set. The n^{th} set in the sequence

$$
\begin{aligned}
s_0(\mathcal{A}) &= \mathcal{A} \\
s_n(\mathcal{A}) &= \bigcup_{0 \le i \le n-1} s_i(\mathcal{A}) \circ s_{n-i-1}(\mathcal{A}),\ n \ge 1
\end{aligned}
$$

is the set of all terms that can be generated by exactly n collisions among elements of $\mathcal{A}$, and

$$S_n(\mathcal{A}) = \bigcup_{0 \le i \le n} s_i(\mathcal{A}),\ n \ge 0$$

is the sequence of sets of all terms produced by up to n collisions in $\mathcal{A}$.

The latter non-decreasing sequence generates the *object horizon* that is accessible from the initial set of objects.

Definition 4.8
The closure $\mathcal{A}^*$ of $\mathcal{A}$ is defined as:

$$\mathcal{A}^* = \lim_{n \to \infty} S_n(\mathcal{A}) = \bigcup_{i=0}^{\infty} s_i(\mathcal{A})$$

We are now able to prove the following result.

Proposition 4.9
Iteration of the map $\mathbf{M}$ yields the closure $\mathcal{A}^*$ of the initial set $\mathcal{A}$. That is:

$$\lim_{n \to \infty} \mathbf{M}^n(\mathcal{A}) = \mathcal{A}^*$$

Proof
Notice that:

$$S_n(\mathcal{A}) \subseteq \mathbf{M}^n(\mathcal{A}) \subseteq S_{2^n-1}(\mathcal{A})$$

With

$$\lim_{n \to \infty} S_{2^n-1}(\mathcal{A}) = \lim_{n \to \infty} S_n(\mathcal{A}) = \mathcal{A}^*$$

the result follows. $\square$

For later use we introduce the following definitions.

Definition 4.10
Any set $\mathcal{B} \subseteq \mathcal{A}^*$*, for which*

$$\lim_{n\to\infty} \mathbf{M}^n(\mathcal{B}) = \mathcal{A}^*$$

is a termed a generator *of* $\mathcal{A}^*$*.*
Any set $\mathcal{B} \subseteq \mathcal{A}^*$*, for which*

$$\lim_{n\to\infty} \mathbf{m}^n(\mathcal{B}) = \mathcal{A}^*$$

is a termed a seeding set *of* $\mathcal{A}^*$*.*

These definitions emphasize a distinction. There are subsets of the closure for which the replacement map $\mathbf{m}$ behaves effectively like the cumulative map $\mathbf{M}$. For a seeding set to generate the closure under the replacement map, that set must clearly be regenerated at some point.

Finally, we note that there can be initial sets yielding the closure $\mathcal{A}^*$ under the replacement map $\mathbf{m}$ which are not subsets of $\mathcal{A}^*$. The totality of these sets constitutes a kind of "basin of attraction" for the closure $\mathcal{A}^*$. We want to distinguish the collection of such sets from the collection of seeding sets, because the latter are associated with a kind of "stability" of $\mathcal{A}^*$ under $\mathbf{m}$ with respect to the removal of subsets. Clearly, a rigorous discussion of the range of possible dynamical behaviors of the replacement map under arbitrary initial conditions is desirable, but requires a suitable topology which is not yet available.

The iterated map framework proves to be conceptually useful, despite the fact that the base model includes an asynchronous stochastic process where the iterations undergone by an object species are frequency dependent.

4.4 Organizations generated by simple $\boldsymbol{\lambda}$-terms

In our base model, the motion in object space settles upon *self-maintaining* sets of objects. As interactions proceed in the reactor, new λ-terms are generated, while others are eliminated randomly. Depending on the initial conditions, and after many interactions have occurred, the system frequently converges on a population of λ-terms that

- maintain each other in the system by mutual production, and that
- share syntactical and algebraic regularities.

This means that the contents of the reactor have reached a particular (possibly infinite) subset of Λ that is invariant under interaction.

Syntactical regularities are made explicit by parsing λ-terms into two kinds of building blocks, called terminal elements and prefixes. Terminal elements are closed λ-terms in normal form. Prefixes are not λ-terms themselves, however prefixes form λ-terms when they precede a terminal element. The invariant subspace contains only λ-terms that are made from a characteristic set of such building blocks.

Algebraic laws are a description of the specific actions associated with each building block. These actions may, of course, depend on the context of a building block within a λ-term. The characterization of the functional relationships among building blocks yields a system of rewrite rules [2, 32]. This system can, in many cases, be exhaustively specified using Knuth-Bendix completion techniques [33, 34]. Rewrite systems which complete, permit a finite specification of all interactions among the elements of the subspace. This way they implicitly determine a grammar for its elements.

The rewrite system cast in terms of building blocks is a description of the converged reactor in which all reference to the underlying λ-calculus has been removed. In other words, the generic λ-calculus can be replaced by another formalism specific to the self-maintaining population of λ-terms in the reactor, that is, a particular *algebraic structure*.

The elements of the invariant subspace are the carrier set of the algebraic structure. Very often, but not always, that set is infinite. Although the reactor has only small capacity (up to 10000 λ-terms), the algebraic structure persists through a fluctuating, yet stably sustained, finite (multi-)set of λ-terms. This occurs whenever the connectivity of the transformation network is such that it channels the production flow towards a core set of λ-terms. This we call *kinetic confinement*.

The main conceptual result is a useful working definition of what we mean by an organization. *An organization is a kinetically self-maintaining algebraic structure*. Self-maintenance here has two aspects:

- Algebraically, a network of mutual production pathways is invariant under applicative interaction.

- Kinetically, the concentrations of the λ-terms in the network remain positive.

The former is a necessary, but not a sufficient condition for the latter. A network can be algebraically self-maintaining, i.e. every λ-term is produced in the network, but its particular connectivity may not suffice to sustain non-zero concentrations of its core components under flow-reactor conditions.

Organizations based on differing algebraic structures are obtained by varying the initial (multi-)set of λ-terms. An infinity of organizations is possible. If the initial set is sufficiently simple, we may use analytical methods to generate the respective organizations. In particular, we are interested in initial sets containing only one object species.

Methods

As a first step, we switch off the dynamical aspect of the base model by considering iterated set actions. We start with an initial set $\mathcal{A}$ consisting of a single λ-term A:

$$\mathcal{A} = \{A\}$$

We then determine the object horizon that is accessible from the initial set. As we have seen in the previous section, the closure $\mathcal{A}^*$ of the initial set $\mathcal{A}$ may be obtained by iterated application of the cumulative map $\mathbf{M}$:

$$\mathcal{A}^* = \lim_{n\to\infty} \mathbf{M}^n(\mathcal{A})$$

where

$$\mathbf{M}(\tilde{\mathcal{A}}) = (\tilde{\mathcal{A}} \circ \tilde{\mathcal{A}}) \cup \tilde{\mathcal{A}}$$

For actual calculations, we use a variant of the above iteration. First, we parse the newly generated λ-terms into building blocks, and thereby we induce a classification of λ-terms according to their building blocks. Next, we characterize the functional relationships among building blocks by a system of rewrite rules. If the rewrite system completes, it represents a finite specification of all interactions among elements of the closure. Finally, we determine the closure itself by iterated application of the rewrite rules to the initial set. The iteration converges after a finite number of steps.

To obtain the organizations generated by the initial set $\mathcal{A}$, we determine all subsets $\mathcal{B}$ of the closure $\mathcal{A}^*$ that are self-maintaining.

Definition 4.11
Let $\mathcal{B}$ be a subset of Λ. $\mathcal{B}$ is self-maintaining *if:*

$$\mathcal{B} \subseteq \mathcal{B} \circ \mathcal{B}$$

Every element of a self-maintaining set is produced by at least one interaction within the set. Still, for a self-maintaining set to represent an organization, the respective multi-set must be also kinetically self-maintaining. That is, the connectivity of the self-maintaing set with respect to mutual production must suffice to sustain non-zero concentrations of its core elements under flow-reactor conditions. A rigorous discussion of kinetic self-maintenance requires an analytical criterion, which is not yet available. Consequently, we consider the question of kinetic self-maintenance separately for each possible organization.

Results

For the sake of simplicity, our initial λ-terms A contain only one or two variables. Before we discuss both cases in detail, we introduce the following abbreviation for prefix abstractions that do not refer to a variable:

$$\lambda^k.M \equiv \lambda x_1.\cdots\lambda x_k.M \quad \text{where} \quad x_1,\ldots,x_k \notin FV(M)$$

$\lambda^k.$ is the simplest example for a prefix building block. If we apply a λ-term of the form $\lambda^k.M$ to an arbitrary λ-term N, we obtain the corresponding rewrite rule:

$$\begin{aligned}
(\lambda^k.M)N &\equiv (\lambda x_1.\cdots\lambda x_k.M)N \\
&\to \lambda x_2.\cdots\lambda x_k.M[x_1 := N] \\
&\equiv \lambda x_2.\cdots\lambda x_k.M \\
&\equiv \lambda x_1.\cdots\lambda x_{k-1}.M \\
&\equiv \lambda^{k-1}.M
\end{aligned}$$

λ-terms with one variable

We now consider the class of λ-terms that contain one variable and an arbitrary number n of abstractions.

Definition 4.12

$$A_i^n \equiv \underset{(1 \leq i \leq n)}{\lambda x_1. \cdots \lambda x_n . x_i}$$

Depending on whether the first abstraction refers to a variable, we may distinguish the following two cases:

$$\begin{array}{ll} \text{(i)} & A_i^n \equiv \lambda x_1. \cdots \lambda x_n . x_i \quad {\scriptstyle (1<i)} \\ \text{(ii)} & A_1^n \equiv \underline{\lambda x_1}. \cdots \lambda x_n . \underline{x_1} \end{array}$$

In case (i), the first $i-1$ abstractions of A_i^n do not refer to a variable. More precisely, we have the following equivalence:

$$\begin{array}{rcl} A_i^n & \equiv & \lambda x_1. \cdots \lambda x_n . x_i \\ & \equiv & \lambda x_1. \cdots \lambda x_{i-1}. \lambda x_i. \cdots \lambda x_n . x_i \\ & \equiv & \lambda^{i-1}. \lambda x_i. \cdots \lambda x_n . x_i \\ & \equiv & \lambda^{i-1}. \lambda x_1. \cdots \lambda x_{n-i+1} . x_1 \\ & \equiv & \lambda^{i-1}. A_1^{n-i+1} \end{array}$$

The applicative behavior of $A_i^n \equiv \lambda^{i-1}.A_1^{n-i+1}$ is determined by the above rewrite rule:

$$\begin{array}{rcl} (A_i^n)X & \equiv & (\lambda^{i-1}.A_1^{n-i+1})X \\ & \to & \lambda^{i-2}.A_1^{n-i+1} \\ & \equiv & A_{i-1}^{n-1} \end{array}$$

That is, A_i^n loses one prefix abstraction when applied to an arbitrary λ-term. If the result of such an application is again applied to an arbitrary λ-term, then, after a total of $i-1$ iterations, the final result A_1^{n-i+1} belongs to case (ii).

In case (ii), the first abstraction of $A_1^n \equiv \lambda x_1.\cdots\lambda x_n.x_1$ does refer to a variable. A_1^n exhibits the following applicative behavior:

$$\begin{aligned}(A_1^n)X &\equiv (\lambda x_1.\cdots\lambda x_n.x_1)X\\ &\to \lambda x_2.\cdots\lambda x_n.X\\ &\equiv \lambda^{n-1}.X\end{aligned}$$

That is, A_1^n adds $n-1$ prefix abstractions to an arbitrary λ-term X. A_1^1 is the identity function.

We are now ready to present the organizations generated by initial λ-terms with one variable. As a basis, we list the relevant rewrite rules and the resulting closure. To render the rewrite rules more comprehensible, we use the notation $X \circ Y$ for the application $(X)Y$:

$$A \equiv A_i^n \equiv \lambda^{i-1}.A_1^{n-i+1}$$

$$\begin{aligned}A_1^{n-i+1} \circ Y &\twoheadrightarrow \lambda^{n-i}.Y\\ \lambda^k.X \circ Y &\twoheadrightarrow \lambda^{k-1}.X \quad {\scriptstyle (k>0)}\end{aligned}$$

$$\mathcal{A} = \{A\}$$

$$\begin{aligned}\mathcal{A}^* &= \{\lambda^k.A_1^1 \mid {\scriptstyle 0\le k\le n-1}\} \quad {\scriptstyle (i=n)} \qquad \underline{\text{UC}}\\ \mathcal{A}^* &= \{\lambda^k.A_1^{n-i+1} \mid {\scriptstyle k\ge 0}\} \quad {\scriptstyle (i<n)} \qquad \underline{\text{PA}}\end{aligned}$$

Depending on the initial λ-term, we obtain two different closures:

<u>UC</u> (Universal Copier)

The identity function A_1^1 makes the finite closure $\mathcal{A}^* = \{\lambda^k.A_1^1 \mid {\scriptstyle 0\le k\le n-1}\}$ self-maintaining. The other elements of the closure lose one prefix abstraction with each application:

$$\begin{aligned}A_1^1 \circ Y &\twoheadrightarrow Y\\ \lambda^k.X \circ Y &\twoheadrightarrow \lambda^{k-1}.X\end{aligned}$$

Prefixes are not regenerated by any reaction. At best, the respective λ-terms are copied as a whole. Under flow-reactor conditions, the closure converges on the kinetically self-maintaining subset $\mathcal{O} = \{A_1^1\}$, the simplest organization possible.

<u>PA</u> (Prefix-Adder)

The infinite closure $\mathcal{A}^* = \{\lambda^k.A_1^{n-i+1} \mid {\scriptstyle k \geq 0}\}$ is self-maintaining. The terminal element A_1^{n-i+1} adds prefix abstractions to other λ-terms, which in turn lose one prefix with each application. Under flow-reactor conditions, finite subsets $\mathcal{O} \subset \mathcal{A}^*$ are kinetically self-maintaing.

λ-terms with two variables

We now consider the class of λ-terms that contain two variables and an arbitrary number n of abstractions.

Definition 4.13

$$A_{i,j}^{m,n} \equiv \lambda x_1.\cdots\lambda x_m.(x_i)\lambda x_{m+1}.\cdots\lambda x_n.x_j$$
$$\scriptstyle (1 \leq i \leq m \leq n,\, 1 \leq j \leq n)$$

Depending on whether the first abstraction refers to a variable, we may distinguish the following four cases:

$$\begin{array}{rll}
\text{(I)} & A_{i,j}^{m,n} \equiv \lambda x_1.\cdots\lambda x_m.(x_i)\,\lambda x_{m+1}.\cdots\lambda x_n.x_j & {\scriptstyle (1<i,\,1<j)} \\
\text{(II)} & A_{1,j}^{m,n} \equiv \underline{\lambda x_1}.\cdots\lambda x_m.(\underline{x_1})\lambda x_{m+1}.\cdots\lambda x_n.x_j & {\scriptstyle (1<j)} \\
\text{(III)} & A_{i,1}^{m,n} \equiv \underline{\lambda x_1}.\cdots\lambda x_m.(x_i)\,\lambda x_{m+1}.\cdots\lambda x_n.\underline{x_1} & {\scriptstyle (1<i)} \\
\text{(IV)} & A_{1,1}^{m,n} \equiv \underline{\lambda x_1}.\cdots\lambda x_m.(\underline{x_1})\lambda x_{m+1}.\cdots\lambda x_n.\underline{x_1} &
\end{array}$$

In case (I), the first $p = \min(i,j) - 1$ abstractions of $A_{i,j}^{m,n}$ do not refer to a variable. Hence, we may also write

$$A_{i,j}^{m,n} \equiv \lambda x_1.\cdots\lambda x_p.A \equiv \lambda^p.A$$

where the first abstraction of A does refer to a variable. In other words, A is of the form (II) – (IV).

$A_{i,j}^{m,n}$ exhibits the following applicative behavior:

$$(A_{i,j}^{m,n})X \equiv (\lambda^p.A)X \rightarrow \lambda^{p-1}.A$$

That is, $A_{i,j}^{m,n}$ loses one prefix abstraction when applied to an arbitrary λ-term. If the result of such an application is again applied to an arbitrary λ-term, then, after a total of $p-1$ iterations, the final result A belongs to one of the cases (II) – (IV).

In the following, we give an exhaustive presentation of organizations generated by initial λ-terms of the form (II) – (IV). The details of the calculations (and the consequent distinction of sub-cases) can be found in the appendix.

In case (II), we obtain organizations that are also generated by initial λ-terms with only one variable. In each sub-case, we first list the relevant rewrite rules and the resulting closure:

<u>II.2′</u>

$A \equiv A_{1,n}^{m,n} \quad {\scriptstyle (m<n)}$

$$\begin{array}{lll} A \circ A & \twoheadrightarrow & \lambda^{m+n-4}.A_1^1 \\ A \circ A_1^1 & \twoheadrightarrow & \lambda^{n-2}.A_1^1 \\ A \circ \lambda^k.X & \twoheadrightarrow & \lambda^{k+m-2}.X \quad {\scriptstyle (k>0)} \\ A_1^1 \circ X & \twoheadrightarrow & X \end{array}$$

$\mathcal{A} = \{A\}$

$\mathcal{A}^* = \{A\} \cup \{\lambda^k.A_1^1 \mid {\scriptstyle 0 \le k \le n-2}\} \quad {\scriptstyle (m\le 2)}$ <u>UC</u>

$\mathcal{A}^* = \{A\} \cup \{\lambda^k.A_1^1 \mid {\scriptstyle k \ge 0}\} \quad {\scriptstyle (m\ge 3)}$ <u>UC</u>

<u>UC</u> (Universal Copier)

Depending on the initial λ-term, the closure $\mathcal{A}^*$ may be finite or infinite. In both cases, the presence of the identity function A_1^1 makes the closure self-maintaining. Still, the initial λ-term A is not regenerated by any reaction, at best it is copied. Under flow-reactor conditions, it is removed from the reactor by the constant outflow. The remaining subset of the closure converges on the kinetically self-maintaining set $\mathcal{O} = \{A_1^1\}$.

<u>II.2″</u>

$A \equiv A_{1,j}^{m,n} \quad {\scriptstyle (m<n,\, m+1\leq j<n)}$

$$
\begin{array}{lcl}
A \circ A & \twoheadrightarrow & \lambda^{m+j-4}.A_1^{n-j+1} \\
A \circ A_1^{n-j+1} & \twoheadrightarrow & \lambda^{n-2}.A_1^{n-j+1} \\
A \circ \lambda^k.X & \twoheadrightarrow & \lambda^{k+m-2}.X \quad {\scriptstyle (k>0)} \\
A_1^{n-j+1} \circ X & \twoheadrightarrow & \lambda^{n-j}.X
\end{array}
$$

$\mathcal{A} \ = \{A\}$

$\mathcal{A}^* = \{\lambda^k.A \mid {\scriptstyle k\geq 0}\} \cup \{\lambda^k.A_1^{n-j+1} \mid {\scriptstyle k\geq 0}\}$ <u>PA</u>

<u>PA</u> (Prefix-Adder)

The infinite set $\{\lambda^k.A_1^{n-j+1} \mid {\scriptstyle k\geq 0}\}$ is self-maintaining. The terminal element A_1^{n-j+1} adds prefix abstractions to other λ-terms, which in turn lose one prefix with each application. Under flow-reactor conditions, finite subsets $\mathcal{O} \subset \{\lambda^k.A_1^{n-j+1} \mid {\scriptstyle k\geq 0}\}$ are kinetically self-maintaining, and consequently referred to as PA-organizations.

The infinite set $\{\lambda^k.A \mid {\scriptstyle k\geq 0}\}$ is self-maintaining, but not closed (as can be seen by self-application of the initial λ-term A). Under flow-reactor conditions, the set is subject to an effective drain, and consequently not kinetically self-maintaining.

<u>II.3</u>

$A \equiv A_{1,j}^{m,n} \quad {\scriptstyle (m<n,\, 1<j\leq m)}$

$$\begin{array}{lll} A \circ A & \twoheadrightarrow & \lambda^{j-2}.A_1^{m-1+n-j} \\ A \circ A_1^{n'} & \twoheadrightarrow & \lambda^{j-2}.A_1^{n'+n-j} \\ A \circ \lambda^k.X & \twoheadrightarrow & \lambda^{k+m-2}.X \quad {\scriptstyle (k>0)} \\ A_1^{n'} \circ X & \twoheadrightarrow & \lambda^{n'-1}.X \end{array}$$

$\mathcal{A} \;= \{A\}$

$\mathcal{A}^* = \{\lambda^k.A \mid {\scriptstyle k\geq 0}\} \cup \{\lambda^k.A_1^{m-1+l(n-j)} \mid {\scriptstyle k\geq 0,\, l\geq 1}\}$

<u>PAG</u>

<u>PAG</u> (Prefix-Adder Generator)

The core of the self-maintaining set $\{\lambda^k.A_1^{m-1+l(n-j)} \mid {\scriptstyle k\geq 0, l\geq 1}\}$ is formed by the infinite sequence of prefix-adders $A_1^{m-1+l(n-j)}$. Under flow-reactor conditions, we might expect a meta-organization consisting of different PA-organizations. But as a matter of fact, a fight between the PA-organizations sets in and only one of them survives.

The infinite set $\{\lambda^k.A \mid {\scriptstyle k\geq 0}\}$ is self-maintaining, but not closed (as can be seen by self-application of the initial λ-term A). Under flow-reactor conditions, the set is subject to an effective drain, and consequently not kinetically self-maintaining.

<u>II.1′</u>

$$A \equiv A^{n,n}_{1,n} \quad {\scriptstyle (1<n)}$$

$$\begin{array}{lcll} A \circ A & \twoheadrightarrow & \lambda^{n-2}.A & \\ A \circ \lambda^k.X & \twoheadrightarrow & \lambda^{k+n-2}.X & {\scriptstyle (k>0)} \end{array}$$

$$\mathcal{A} \ = \{A\}$$

$$\begin{array}{lll} \mathcal{A}^* = \{A\} & {\scriptstyle (n=2)} & \underline{\text{SC}} \\ \mathcal{A}^* = \{\lambda^k.A \,|\, {\scriptstyle k\geq 0}\} & {\scriptstyle (n\geq 3)} & \underline{\text{OPA}} \end{array}$$

Depending on the initial λ-term, we obtain two different closures:

<u>SC</u> (Self-Copier)

The closure $\mathcal{A}^*$ consists only of the initial λ-term A, which acts as a self-copier. As a consequence, the closure is also kinetically self-maintaining, and hence an organization.

<u>OPA</u> (Other Prefix-Adder)

The infinite closure $\mathcal{A}^* = \{\lambda^k.A \,|\, {\scriptstyle k\geq 0}\}$ is self-maintaining. The initial λ-term A adds prefix abstractions to other λ-terms, which in turn lose one prefix with each application. Under flow-reactor conditions, finite subsets $\mathcal{O} \subset \mathcal{A}^*$ are kinetically self-maintaining. These organizations are structurally equivalent to PA-organiztions.

<u>II.1″</u>

$$A \equiv A^{n,n}_{1,j} \quad {\scriptstyle (1<j<n)}$$

$$\begin{array}{lcl} A \circ A & \twoheadrightarrow & \lambda^{j-2}.A^{2n-j,2n-j}_{1,n} \\ A^{n',n'}_{1,j'} \circ A^{n'',n''}_{1,j''} & \twoheadrightarrow & \lambda^{j-2}.A^{n''+(n'-j'),n''+(n'-j')}_{1,j''+(n'-j')} \\ A^{n',n'}_{1,j'} \circ \lambda^k.X & \twoheadrightarrow & \lambda^{k+n'-2}.X \quad {\scriptstyle (k>0)} \end{array}$$

$$\mathcal{A} \ = \{A\}$$

$$\mathcal{A}^* = \{\lambda^k.A^{n+l(n-j),n+l(n-j)}_{1,j+l(n-j)} \mid {\scriptstyle k\geq 0,\, l\geq 0}\} \qquad \underline{\text{RQ}}$$

<u>RQ</u> (Red Queen)

The infinite sequence of λ-terms $A_l \equiv A^{n+l(n-j),n+l(n-j)}_{1,j+l(n-j)}$ forms the core of the self-maintaining closure $\mathcal{A}^*$. Hence, we may restate the above results as follows:

$$\begin{array}{lcl} A_l \circ A_{l'} & \twoheadrightarrow & \lambda^{j-2}.A_{l'+1} \\ A_l \circ \lambda^k.X & \twoheadrightarrow & \lambda^{k+n+l(n-j)-2}.X \end{array}$$

$$\mathcal{A}^* = \{\lambda^k.A_l \mid {\scriptstyle k\geq 0,\, l\geq 0}\}$$

Reactions among elements of the sequence A_l increase the index l by 1, whereas elements with lower indices are not regenerated by any reaction. Under flow-reactor conditions, none of the subsets $\mathcal{O} \subset \mathcal{A}^*$ is kinetically self-maintaining.

The ever changing population of λ-terms in the reactor rather reminds of the continuing evolution in complex ecosystems (the so-called Red Queen dynamics). Just like the Red Queen in Lewis Carroll's *Through the Looking Glass*, the RQ-"organization" is forced to keep running to stay in the same place.

Not until case (III), we obtain organizations that are structurally different from the organizations generated by initial λ-terms with one variable. Again, we first list the relevant rewrite rules and the resulting closure:

III.1

$$A \equiv A^{n,n}_{i,1} \quad {\scriptstyle (1<i)}$$
$$p \equiv \lambda x_1.\lambda^{n-i}.(x_1)$$

$$\begin{array}{lcl} A \circ X & \twoheadrightarrow & \lambda^{i-2}.pX \\ pX \circ A & \twoheadrightarrow & \lambda^{n-2}.pX \\ pX \circ \lambda^k.Y & \twoheadrightarrow & \lambda^{k+n-i-1}.Y \quad {\scriptstyle (k>0)} \\ pX \circ pY & \twoheadrightarrow & \lambda^{2n-2i}.(X \circ Y) \end{array}$$

$$\mathcal{A} \ = \{A\}$$

$$\mathcal{A}^* = \{A\} \cup \{\lambda^{k_0}.p \cdots \lambda^{k_N}.pA \,|\, {\scriptstyle 0\leq k_l \leq n-2,\, N\geq 0}\}$$
$${\scriptstyle (i=n)} \qquad \underline{\text{CPG}}$$
$$\mathcal{A}^* = \{A\} \cup \{\lambda^{k_0}.p \cdots \lambda^{k_N}.pA \,|\, {\scriptstyle k_l \geq 0,\, N\geq 0}\}$$
$${\scriptstyle (i<n)} \qquad \underline{\text{PG}}$$

The initial λ-term A adds two kinds of prefixes to other λ-terms: prefix abstractions and the newly generated prefix p. Depending on the initial λ-term, we distinguish two different closures:

CPG (Cycle from a Prefix-Generator)

The infinite closure $\mathcal{A}^*$ contains λ-terms starting with an alternating sequence of the two kinds of prefixes and ending with the terminal element pA. The sequence of prefixes may be infinite, but the number of successive prefix abstractions is finite. The initial λ-term A itself is not regenerated.

Under flow reactor conditions, the closure converges on the finite subset $\mathcal{O} = \{\lambda^k.pA \,|\, {\scriptstyle 0\leq k\leq n-2}\}$. This set represents a kinetically self-maintaining "cycle", and hence a structurally new organization.

<u>PG</u> (Prefix-Generator)

The infinite closure $\mathcal{A}^*$ contains λ-terms starting with an alternating sequence of the two kinds of prefixes and ending with the terminal element pA. Both the sequence of prefixes and the number of successive prefix abstractions may be infinite. The initial λ-term A is not regenerated.

Under flow reactor conditions, the closure converges on the infinite subset $\{\lambda^k.pA \mid {\scriptstyle k \geq 0}\}$. Finite subsets $\mathcal{O} \subset \{\lambda^k.pA \mid {\scriptstyle k \geq 0}\}$ are kinetically self-maintaining. These organizations are structurally equivalent to PA-organiztions.

<u>III.2</u>

$$A \equiv A^{m,n}_{i,1} \quad {\scriptstyle (1<i,\, m<n)}$$
$$p \equiv \lambda x_1.\lambda^{m-i}.(x_1)$$

$$\begin{array}{lll} A \circ X & \twoheadrightarrow & \lambda^{i-2}.p\lambda^{n-m}.X \\ pX \circ A & \twoheadrightarrow & \lambda^{m-2}.p\lambda^{2n-2m}.X \\ pX \circ \lambda^k.Y & \twoheadrightarrow & \lambda^{k+m-i-1}.Y \quad {\scriptstyle (k>0)} \\ pX \circ pY & \twoheadrightarrow & \lambda^{m+n-2i-1}.X \end{array}$$

$$\mathcal{A} \;= \{A\}$$

$$\mathcal{A}^* = \{\lambda^{k_0}.p\lambda^{k_1}.\cdots p\lambda^{k_N}.A \mid {\scriptstyle k_l \geq 0,\, N \geq 0}\} \qquad \underline{\text{OPG}}$$

<u>OPG</u> (Other Prefix-Generator)

The initial λ-term A adds two kinds of prefixes to other λ-terms: prefix abstractions and the newly generated prefix p. The infinite closure $\mathcal{A}^*$ consists of λ-terms starting with an alternating sequence of the two kinds of prefixes and ending with the terminal element pA. Both the sequence of prefixes and the number of successive prefix abstractions may be infinite.

Under flow-reactor conditions, finite subsets $\mathcal{O} \subset \mathcal{A}^*$ are kinetically self-maintaing. These organizations are structurally more complex than PA-organizations.

In case (IV), we obtain organizations that have already been generated in the previous cases. Again, we first list the relevant rewrite rules and the resulting closure:

$\underline{\mathrm{I} \Rightarrow \mathrm{IV.1}}$

$$A \equiv A^{n,n}_{1,1}$$
$$\tilde{A} \equiv \lambda^p.A \quad {\scriptstyle (p>0)}$$

$$A \circ A \quad \twoheadrightarrow \quad \text{elastic !}$$
$$A \circ \lambda^k.X \quad \twoheadrightarrow \quad \lambda^{k+n-2}.X \quad {\scriptstyle (k>0)}$$

$$\{A\}^* = \{A\}$$

$$\mathcal{A} = \{\tilde{A}\}$$

$$\mathcal{A}^* = \{\lambda^k.A \,|\, {\scriptstyle 0\le k\le p}\} \quad {\scriptstyle (n\le 2)} \quad \underline{\mathrm{SA}}$$
$$\mathcal{A}^* = \{\lambda^k.A \,|\, {\scriptstyle k\ge 0}\} \quad {\scriptstyle (n\ge 3)} \quad \underline{\mathrm{OPA}}$$

Since interactions between initial λ-terms A of the form (IV.1) are elastic, we consider λ-terms $\tilde{A} \equiv \lambda^p.A$ of the form (I) instead. Depending on the structure of the initial λ-term, we distinguish two different closures:

<u>SA</u> (Self-Applicator)

The finite closure $\mathcal{A}^* = \{\lambda^k.A \,|\, {\scriptstyle 0\le k\le p}\}$ is self-maintaining. Still, there is no reaction that regenerates lost prefix abstractions. Under flow-reactor conditions, the closure converges on the kinetically self-maintaining subset $\mathcal{O} = \{A\}$, where the λ-term A is not self-interacting.

<u>OPA</u> (Other Prefix-Adder)

The infinite closure $\mathcal{A}^* = \{\lambda^k.A \,|\, {\scriptstyle k\ge 0}\}$ is self-maintaining. The initial λ-term A adds prefix abstractions to other λ-terms, which in turn lose one prefix with each application. Under flow-reactor conditions, finite subsets $\mathcal{O} \subset \mathcal{A}^*$ are kinetically self-maintaining. These organizations are structurally equivalent to PA-organiztions.

<u>IV.2</u>

$$A \equiv A^{m,n}_{1,1} \quad {\scriptstyle (m<n)}$$

$$\begin{array}{lll} A \circ A & \twoheadrightarrow & \lambda^{n+m-3}.A \\ A \circ \lambda^k.X & \twoheadrightarrow & \lambda^{k+m-2}.X \quad {\scriptstyle (k>0)} \end{array}$$

$$\mathcal{A} = \{A\}$$

$$\mathcal{A}^* = \{\lambda^k.A \,|\, {\scriptstyle 0 \leq k \leq n+m-3}\} \quad {\scriptstyle (m\leq 2)} \qquad \underline{\text{CSA}}$$
$$\mathcal{A}^* = \{\lambda^k.A \,|\, {\scriptstyle k \geq 0}\} \quad {\scriptstyle (m\geq 3)} \qquad \underline{\text{OPA}}$$

Depending on the structure of the initial λ-term, we distinguish two different closures:

<u>CSA</u> (Cycle from a Self-Applicator)

The finite closure $\mathcal{A}^* = \{\lambda^k.A \,|\, {\scriptstyle 0 \leq k \leq n+m-3}\}$ is self-maintaining. The initial λ-term A adds prefix abstractions to other λ-terms, which in turn lose one prefix with each application. Under flow-reactor conditions, the closure represents another kinetically self-maintaining "cycle".

<u>OPA</u> (Other Prefix-Adder)

The infinite closure $\mathcal{A}^* = \{\lambda^k.A \,|\, {\scriptstyle k \geq 0}\}$ is self-maintaining. The initial λ-term A adds prefix abstractions to other λ-terms, which in turn lose one prefix with each application. Under flow-reactor conditions, finite subsets $\mathcal{O} \subset \mathcal{A}^*$ are kinetically self-maintaining. These organizations are structurally equivalent to PA-organiztions.

5 Conclusions

From the "object problem" ...

Whenever a particular level of analysis of Nature is populated with objects whose internal structure engenders specific action capable of changing or creating other objects, the dynamical systems methodology encounters a fundamental limit. The reason is that the formal machinery of dynamical systems is geared to handle changes in quantities, but not changes in object structure. We believe that a formal understanding of such a level of Nature requires a theory that *combines* variables that can take objects as values with the more familiar variables that hold quantitative values (such as concentrations). To convey an intuitive flavor of this, think of action as "parametrized" by structure, and imagine a "derivative" in object space giving information about the change of object action resulting from a change in object structure. Clearly, being able to meaningfully define such a thing puts stringent conditions on how structure is coupled to action. To even start thinking about this requires a powerful formal machinery capable of expressing the coupling of structure to action for the objects pertinent to a particular domain of application. This is the "object problem".

... to a "theory of chemistry"

The "object problem" is nowhere seen more crisply than in chemistry. Chemical reactions are events in which both concentrations (i.e., quantities) and objects (i.e., structures) change. The projection of a chemical reaction involving large numbers of molecules on a phase-space of concentrations is known as reaction kinetics. To set up a chemical reaction as a dynamical system in concentration space, one only requires knowledge of the proper couplings among the concentrations of reactants and products. It is sufficient if these are known as empirical facts; knowledge of the chemical identity and properties of reactants and products is not necessary. Cranking the tools of, say, infintesimal calculus yields the time evolution of reactant and product concentrations. Remove kinetics for a moment by considering just the information conveyed by a chemical reaction when it is notated on paper. We are left with a reaction arrow, "$\rightarrow$", expressing a *relation* among molecular structures. A general method capable of describing the time evolution of the contents of a reaction vessel for an *arbitray* initial mixture of molecular species would require nothing less than a formal system implicitly representing the space of molecular objects and the relation "$\rightarrow$" over them. That is a formal theory of chemistry.

Where do we get a formal theory of chemistry? The answer is crucial to our approach. To date we do not have a formal, axiomatized theory of chemistry that is useful in everyday practice, despite the fact that quantum mechanics successfully grounds chemistry in the behavior of electrons and nuclei. The problem is one of choosing the "right" level of description. With respect to both molecular biology and industrial metabolisms alike, quantum mechanics is far too fine grained, and, aside from issues of feasibility, does not convey a satisfactory understanding of "what chemistry is actually doing"; it is too close to the trees to see the forest. To put it provokingly, our understanding of life will derive in large measure from *how* we understand chemistry. It is clear, then, that identifying a coarser grain of analysis capable of hosting a formal theory of chemistry would be of tremendous practical import and by no means limited to the foundations of theoretical biology.

The stance we took in this work, is based on the intuition that at some level of description the reactive processes of chemistry are *analogous* to manipulations (rewrites) of syntactical objects. This puts us right into the domain of the computational sciences, chapter 3. In the present context, the reader is well advised to detach from an all-too narrow notion of computation as "number crunching". Much effort in the computational sciences goes into devising formal systems of syntactical constructs (we call "objects") that are interrelated by operations of transformation defined on them. It is in this sense that "computation" is the science of the construction of abstract objects with structure-specific "behavior".

What is crucial in the present context is *how* object behavior is synthesized from basic elements, as it is here that insights into fundamental mechanisms of "construction" or "interaction" are revealed, and can be compared with empirical facts. For what is desired in a theory of objects is not just a formalism and the theorems that accompany it, but a transparency of interpretation in the intended application domain. In a chemical application, one wants to capture *at least* the twin facts that (i) product molecules are lawfully constructed from substitution of parts of reactant molecules and (ii) that the same product can be produced by a diversity of different reactants. One wants, therefore, a theory of *combinational structures* and *substitution* together with the resultant theory of *equality*. Indeed, this is what drew us originally to λ-calculus. A great variety of alternative formal systems - Turing machines, Petri nets, Post systems, cellular automata, to mention but a few - allow expression of the same set of functions on the natural numbers, and, thus, may be regarded as being equivalent *in that respect*. It should be clear by now, however, that what is needed here is not merely a member of this universality class, but a member whose features are germane to the chemical problem at hand.

Identifying λ-calculus as a plausible candidate for a chemical interpretation hardly qualifies λ-terms as anything but the most metaphorical of molecules. Yet, we do not require a one-to-one mapping between some formalism expressing computational processes and real-world molecules with their chemical reactions. The map should not be confused with the territory; we do not want to "simulate" chemistry. We take the computational perspective as one enabling a different - logical - level of description of chemistry which is distinct from one that accounts for its actual physical implementation. The latter is the domain of quantum physics. For example, whether the actual protein folding process belongs to the class of computable functions is entirely irrelevant to us. For all we need to capture is the *logic* of the connection between structure and action specificity, not the physical process by which this connection is implemented. Herein lies the point of a specification language for chemistry.

... to a "theory of functional organization"

The core of our model is a theory of object construction - rather than the imitation of particular chemicals. This is what gives us the capacity to specify what the "organization" of an emerging collective of objects is in terms of a mathematical formalism. Three broad consequences follow.

(1) The abstraction of molecules in the base model as symbolic functions allows organization to be detected as a closure of interaction, manifested by invariant syntactical regularities and invariant algebraic laws characterizing the action of those objects maintained in the collective. It cannot be overemphasized that this characterization can be made

by an observer of the system who is ignorant of λ-calculus. Indeed, an organization can be specified as an algebraic rewrite system that is independent from λ-calculus, and, thus, the process by which it originated. With the theory of objects comes whole-cloth a quite different theory of the collective.

(2) A formal theory of objects makes transparent which features of the collective organization derive from the underlying theory of objects, and which features are curiosities derived from particular initial conditions, parameter settings, or a particular chemical stance. The distinction is between what has been called "digital naturalism" [20] and the claim for a theory of self-maintenance.

(3) As emphasized at the outset, an abstract theory of objects plays a role analogous to that of differential equations. The analysis of a dynamical system cast in terms of differential equations yields the characterization of manifolds in phase space that govern the set of its *possible* trajectories. Consider the base model and imagine we seeded our reactor with one λ-term known to be a basis for all λ-calculus. Imagine further that the container is itself infinite in size and the reactor would then be capable of holding all possible (normal form) expressions. When we impose a dynamics on the objects (which, in our case, is a scheme coupled to the reactor size), we sieve particular "trajectories" in object space. Recall that our kinetic scheme is designed to favor the maintenance of objects that are constructed by the extant population of objects. As a consequence "trajectory" in object space "converges" to an "attractor" - a self-maintaining organization.

This casting emphasizes the need for a theory of the "motion" in "object space" induced by the object constructors (here, functional application or logical inference) under the continously updating kinetics imposed by the extant network of objects. How might such a motion change under a different dynamics ? Or with equivalent dynamics and different object-constructors? Is there a meaningful formal concept of a "trajectory" in "object space"? What is "continuity"? Is there a useful definition of "distance" between "attractors" (in our case, algebraic structures in λ-space)? Questions like these require a theory of objects, not only to be answered, but even to be asked.

The value is apparent. Consider just one instance. A methodological imperative of the dynamical systems approach is the *a priori* choice of the pertinent entities and their functional couplings defining the system. The fact that the choice must be made *a priori* has the consequence that *the dynamical systems methodology can never be used to address the origin of that same system*, a profound limitation to the existence problem in biology [16]. It is apparent in the base model that our reactor in settling upon a self-maintaining set of objects has settled upon a fixed system of variables and functional couplings between them; thus, *particular dynamical systems appear as limiting cases (such as "fixed points") of constructive dynamical systems*. Are constructive dynamical systems "generators" of dynamical systems? If so, a formalization of the motion in such a space holds promise as a methodology to address scientific questions which include the phrase "the origin of ...".

... to a "theory of the organism"

Biology has only two claims to theories unto itself - Mendel's theory of transmission and Darwin's theory of natural selection. Fisher, Haldane, and Wright demonstrated that no conflict existed between the two theories. Still, the tools employed to show that the great theories of biology were concordant required casting the problem in a fashion that threw out the constructive aspects of biology, rendering the problem tractable as one in dynamical systems. Throwing out construction meant throwing out the organism; trying to put the organism back in is fair epitome of the intellectual history ever since. The claim is that biology requires a trinity of theories; we have two of them; we lack only a "theory of the organism". The claim we make is that the self-maintaining organizations we derive hold promise as that missing theory. Indeed, such a theory need not await a global solution to the specification language for chemistry. The fact that our organizations can be described in a formalism distinct from that in which they were generated (i.e., as abstract rewrite systems rather than λ-terms) leaves their terms (like those of any syntax) open to interpretations other than chemical. From this realization flows a diversity of potential applications.

Given a universe of self-maintaining abstract rewrite systems, the uses are limited solely by the properties of the particular rewrite system and the interpretation given to its terms. An interpretation of the terms as engineering functions in a machine might be route to a self-repair mechanism, an interpretation as a semiotic unit as a device for natural language interpretation, and interpretation of terms-as-molecules as germane to a blueprint for the design of a self-maintaining chemical manufacturing process [4], as it is to a metabolic cycle in a cell or a system of cell-cell communications defining an organ. The origin of such specifications from a research program in artifical chemistry or in experimental λ-calculus is irrelevant. It cannot be overemphasized that herein lies the significance of the characterization of our organizations in an alternative formalism.

6 Outlook

The architecture of the base model consists in joining a formal calculus with a dynamical system. The organizations generated by the base model exhibit properties – such as regeneration, structural extension, capacity for hierarchical nesting – akin to properties of living organisms. Yet both the formal calculus and the dynamical system must be improved to narrow the gap to molecular biology.

6.1 Modifying the formal system

In our minimalist abstraction of chemistry, molecules are treated as discrete structures, defined inductively. A molecule is an atom or a combination of molecules. Reactions are seen as events where such structures interact to construct new structures. The basic mechanism of a reaction is the exchange of one substructure by another, i.e. a substitution.

This abstraction of chemistry coincides with the description of a mathematical function as a rule, rather than a set. (In the former case a function is a suite of computations that generates an output when applied to an input; in the latter a function is a set of input-output pairs.) The canonical system that formalizes the notion of a function as a rule is the λ-calculus. To express rules, so-called λ-terms are defined inductively. Functions are applied to arguments, which can themselves be functions, returning a new function as a result. The evaluation of a functional application is done by repeated substitutions.

Still λ-terms are far from molecules and its organizations far from organisms. Some of the major violations of chemistry-as-we-know-it are highlighted in the following.

- **Shape**
  Molecules interact selectively, but λ-terms can act on one another indiscriminately.
- **Symmetry**
  A chemical reaction is a symmetric event, but functional application is not commutative in λ-calculus.

- **Multiple products**

  Chemical reactions can yield several molecules on the product side. This cannot occur in λ-calculus, because functions yield a single object as value.

- **Reversibility**

  A chemical reaction is a reversible event, but the application of a function is typically not invertible.

- **Mass conservation**

  Molecules are resources and are used up when they react. Moreover, atom types and numbers are conserved during a reaction event. This is violated in the dynamical systems component of the base model, because we do not use up the reactants at the time of the reaction (a function has a finite lifetime, which, however, is not linked to its reactive interactions). Conservation laws are also violated in the formal calculus component of the base model, due to terms in which a bound variable occurs more than once ("nonlinearity"). "Mass conservation" refers here to an accounting of terms. To clarify this informally, consider supplying the argument 5 to the function $f(x) = x^2 + 2x + 3$. The term 5 gets used twice; once when substituting it in x^2 and once in $2x$. If terms are treated as resources, we must account for the second instance of 5.

- **Rate constants**

  Chemical reactions proceed with different velocities leading to a separation of time scales in reaction networks. This fact is neglected in the base model where reduction to normal form of an applicative interaction is considered to occur in one "instant of time".

These limitations are substantial and motivate the improvements to which we now turn.

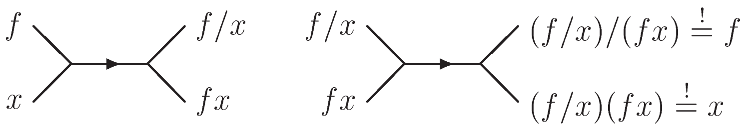
We believe that some aspects of molecular **shape** are appropriately formalized by the concept of a "type system", which is a rather natural notion in the context of λ-calculi. In typed λ-calculus a term is augmented with a notation expressing the "kind" of terms on which it can act, as well as the "kind" of terms resulting from this action. For example, a term denoting a function acting on terms of type A and returning terms of type B is itself of type $A \to B$. Such a function is prevented from acting on objects other than of type A. This may also prove useful in addressing the issue of **symmetry** by preventing one of the possible applications, fg or gf, between two terms f and g from occurring. Type theory is a very active field of research in computer science, and a number of useful type systems have appeared in the literature (for a survey see [6]). We plan to study whether certain constraints of molecular interaction can be modeled by using System F [39] possibly augmented with union types and subtyping.

Moreover, typed λ-calculus is a proof-theory of intuitionistic logic in natural deduction style [23]. This connection extends our previous view of chemistry to one in which molecular shapes are seen as propositions and physical molecules as proofs of propositions. A chemical reaction then literally becomes an inference with the reactants as premisses and the products as conclusions.

To address the issue of **multiple products** requires either the modification of the evaluation process in λ-calculus, or the definition of a new calculus.

In the first case, we note that in our model a chemical reaction starts with a collision joining (two) terms into a new ("transition") term which is subsequently rewritten to its normal form. We may think of the "splitting" of a term as an event that reverses a joining collision. More specifically, the collision of two molecules is represented by the formation of a reducible expression (redex) upon application of two λ-terms. A splitting occurs when an intermediate term arises during normalization whose form is that of a "collision redex", $(\lambda x.A)B$. At this point the two well-defined subterms $(\lambda x.A)$ and B may separate into two products. (Upon continuing normalization, each of these terms might undergo further splitting yielding an overall outcome of more than two products).

The second case is more audacious. In response to our work, Jonathan Rees (MIT) has recently devised a quite intriguing and elegant approach to multiple products [47]. He starts with the issue of **reversibility**, and notes that reversibility of the application of a function is weaker than invertibility of a function. Rees *defines* an interaction between a term f and an argument x to produce both the value, fx, and a so-called "residue", written f/x, which is a term such that its interaction with the argument fx yields x as the value and f as its residue. Nothing is said about the interaction of f/x with terms other than fx. In Rees' notation the definition of interaction looks like



and yields two equations which are taken as an implicit definition of the "residue operator", /. Rees has begun to construct a modification of combinatory logic (a computational system related to λ-calculus) whose rules comply with this definition of interaction. It is worth noting that the rules cannot be made unambigous unless one considers **mass conservation**. In Rees' version of the calculus the nonlinear S-combinator (whose λ-translation contains two occurrences of a bound variable) has to be replaced by two linear versions.

Although still unexplored, the elegance of the Rees-system raises an interesting point: it may be that the issue concerning multiple products may not be solved in a theoretically appealing way without simultaneously addressing the issues of mass conservation (e.g., "linearity", see also [22]) and reversibility. The calculus proposed by Rees deserves attention, and we plan to join his efforts in understanding this system and evaluating its utility for our research agenda.

A proxy for differential **reaction rates** may be obtained by taking into account the number of rewrite steps needed to normalize an interaction term. In other words, the rate constant of an application reflects the time required to compute it.

Yet even if all these issues were adequately addressed, it still remains unclear to what degree a faithful translation from real molecules and their actions into the resultant formal system would be possible. Part of the problem is understanding to which extent attributes of molecules are "irreducibly physical" and to which extent they possess an abstract logical counterpart. For example, what *role* does energy play in this context? Could it play a role analogous to an order relation among terms of a formal system?[5]

6.2 Modifying the dynamical system

The particular population dynamics imposed on the formal calculus in the base model leaves ample room for improvement. Physical aspects, such as spatial extension, may simply be added to current logical formalisms.

[5] Incidentally, a rewrite relation on terms that is confluent and terminating requires the existence of an order relation, where rewriting lowers the order of a term [2].

From a dynamical viewpoint the base model was (functionally) closed, since no functions were supplied from the outside. We can must now turn our attention to the construction of organizations under a **true flow** provided by a controlled supply of some initial set of functions and the withdrawal of others. Moreover, progress with the "multiple products" issue of the previous section will enable us to study other kinetic schemes which do not fix the total number of particles, but where the system size is determined by the organization itself.

A further natural improvement of our base model is to consider the formation of organizations in a **spatially extended** system. Space is obviously an important constraint on the collision frequency of function-particles. To explore the effect of differential diffusion rates we could, for example, make the diffusion coefficient of a term dependent on its length. The Santa Fe Institute's SWARM group is developing a public domain platform for the simulation of agent-based models across research domains. This software seems well suited for implementing a spatial version of our flow reactor.

The self-maintaining organizations generated in the base model are organized only in the algebraic sense of production pathways that have attained closure. They lack, however, a causal structure involving **coordinated sequences of action**, since it only matters who produces what by interaction with whom without regard to timing. Sequences of actions cannot be stably maintained in a flow reactor with functional application as the only form of interaction. Additional modes of interaction, such as functional composition, need to be considered. However, associations of actions into sequences (first apply function A, then function B, then...) must also be given the possibility of disassembling again, to permit the exploration of various patterns of coordination.

6.3 Analyzing organizations

An exploration of the proposed modifications requires computer simulations. Aside from the software that must be devloped to implement the proposed goals, there remains the difficult and tedious problem of analyzing the organizations arising in any future model of this kind. Up to now the resulting organizations were "parsed" and analyzed largely by hand, sifting through screens full of computer generated data. Software is needed for assisting in elucidating the grammar and algebraic structure of the organizations. The extent to which these analyses can be made online determines the complexity of the phenomena that we can detect. Consider, for example, computer simulations in which many independently generated organizations are brought into interaction. It is virtually impossible to keep track of the “glue” and of the fate of each individual organization by hand.

6.4 Finding a canonical example

By adequately addressing each of the mentioned issues we hope to move closer to an abstraction of chemistry-as-we-know-it (and that subsequent steps would suggest themselves more readily as they become more constrained by previous choices). Yet, even if the desired theory of chemistry does not exist, explorations like these are necessary. They provide insights into a functional mode of organization that cannot be attained by traditional dynamical systems alone; they surely will expose the generic and robust properties of constructive dynamical systems, a class to which molecular systems belong.

The proposed model extensions will generate a variety of organizations, probably similar in flavor to those encountered in the base model. We have obviously no way to explore them all, and we are still far from real chemistry. Hence many organizations are invariably “nonsense” from the chemical point of view. However, suppose that we could represent some section of metabolism in an appropriate formal calculus, and that this representation would indeed constitute a self-maintaining fixed point under a flow reactor dynamics. Finding such a canonical example would be highly desirable, since we were able to demonstrate that a real molecular self-maintaining system is at least within the domain of discourse of our model universe, even if we may not know the boundary conditions needed to actually generate it.

Appendix

In section 4.4, we determine the closure of initial sets consisting of a single λ-term. For this purpose, we require the rewrite rules that specify the interactions among elements of the closure.

In this appendix, we present the respective calculations for initial λ-terms A with two variables. After recapitulating the relevant definitions, we first calculate the result A' of the self-application $(A)A$. Thereby, we introduce a distinction of cases according to the structure of A.

Next, we parse A' into building blocks, which induces a classification of λ-terms according to their building blocks. Finally, we characterize the mutual applications among classes of λ-terms by a system of rewrite rules.

Definitions

$$A_i^n \equiv \lambda x_1.\cdots\lambda x_n.x_i$$
$$\scriptstyle (1\leq i\leq n)$$

$$\boxed{\begin{array}{rl} \text{(i)} & A_i^n \equiv \lambda x_1.\cdots\lambda x_n.x_i \quad {\scriptstyle (1<i)} \\ \text{(ii)} & A_1^n \equiv \underline{\lambda x_1}.\cdots\lambda x_n.\underline{x_1} \end{array}}$$

$$A_{i,j}^{m,n} \equiv \lambda x_1.\cdots\lambda x_m.(x_i)\lambda x_{m+1}.\cdots\lambda x_n.x_j$$
$$\scriptstyle (1\leq i\leq m\leq n,\, 1\leq j\leq n)$$

$$\boxed{\begin{array}{rl} \text{(I)} & A_{i,j}^{m,n} \equiv \lambda x_1.\cdots\lambda x_m.(x_i)\,\lambda x_{m+1}.\cdots\lambda x_n.x_j \quad {\scriptstyle (1<i,\,1<j)} \\ \text{(II)} & A_{1,j}^{m,n} \equiv \underline{\lambda x_1}.\cdots\lambda x_m.(\underline{x_1})\lambda x_{m+1}.\cdots\lambda x_n.x_j \quad {\scriptstyle (1<j)} \\ \text{(III)} & A_{i,1}^{m,n} \equiv \underline{\lambda x_1}.\cdots\lambda x_m.(x_i)\,\lambda x_{m+1}.\cdots\lambda x_n.\underline{x_1} \quad {\scriptstyle (1<i)} \\ \text{(IV)} & A_{1,1}^{m,n} \equiv \underline{\lambda x_1}.\cdots\lambda x_m.(\underline{x_1})\lambda x_{m+1}.\cdots\lambda x_n.\underline{x_1} \end{array}}$$

Calculation of $(A)A \twoheadrightarrow A'$ for the cases (II) – (IV)

$\boxed{\text{II}}$

$$A \equiv \underset{(1<j)}{A^{m,n}_{1,j}} \equiv \lambda x_1.\cdots\lambda x_m.(x_1)\lambda x_{m+1}.\cdots\lambda x_n.x_j$$

$$\begin{aligned}
(A)A &\equiv (\lambda x_1.\cdots\lambda x_m.(x_1)\lambda x_{m+1}.\cdots\lambda x_n.x_j)A \\
&\to \lambda x_2.\cdots\lambda x_m.(A)\lambda x_{m+1}.\cdots\lambda x_n.x_j \\
&\equiv \lambda x_2.\cdots\lambda x_m.(\lambda y_1.\cdots\lambda y_m.(y_1)\lambda y_{m+1}.\cdots\lambda y_n.y_j)\lambda x_{m+1}.\cdots\lambda x_n.x_j \\
&\to \lambda x_2.\cdots\lambda x_m.\lambda y_2.\cdots\lambda y_m.(\lambda x_{m+1}.\cdots\lambda x_n.x_j)\lambda y_{m+1}.\cdots\lambda y_n.y_j
\end{aligned}$$

<u>II.1</u>

$(1<j, m=n)$

$$\begin{aligned}
(A)A &\twoheadrightarrow \lambda x_2.\cdots\lambda x_m.\lambda y_2.\cdots\lambda y_m.(\lambda x_{m+1}.\cdots\lambda x_n.x_j)\lambda y_{m+1}.\cdots\lambda y_n.y_j \\
&\equiv \lambda x_2.\cdots\lambda x_n.\lambda y_2.\cdots\lambda y_n.(x_j)y_j \\
&\equiv \lambda x_2.\cdots\lambda x_{j-1}.\underline{\lambda x_j.\cdots\lambda x_n.\lambda y_2.\cdots\lambda y_{j-1}.\lambda y_j.\cdots\lambda y_n.(x_j)y_j} \\
&\equiv \lambda^{j-2}.\underline{\lambda x_1.\cdots\lambda x_{n-j+1}.\lambda^{j-2}.\lambda y_1.\cdots\lambda y_{n-j+1}.(x_1)y_1} \\
&\equiv \lambda^{j-2}.A^{2n-j,2n-j}_{1,n}
\end{aligned}$$

<u>II.2</u>

$(1<j, m<n, m+1\leq j)$

$$\begin{aligned}
(A)A &\twoheadrightarrow \lambda x_2.\cdots\lambda x_m.\lambda y_2.\cdots\lambda y_m.(\lambda x_{m+1}.\cdots\lambda x_n.x_j)\lambda y_{m+1}.\cdots\lambda y_n.y_j \\
&\to \lambda x_2.\cdots\lambda x_m.\lambda y_2.\cdots\lambda y_m.\lambda x_{m+2}.\cdots\lambda x_n.x_j \\
&\equiv \lambda x_2.\cdots\lambda x_m.\lambda y_2.\cdots\lambda y_m.\lambda x_{m+2}.\cdots\lambda x_{j-1}.\underline{\lambda x_j.\cdots\lambda x_n.x_j} \\
&\equiv \lambda^{m+j-4}.\underline{\lambda x_1.\cdots\lambda x_{n-j+1}.x_1} \\
&\equiv \lambda^{m+j-4}.A^{n-j+1}_1
\end{aligned}$$

<u>II.3</u>

$(1<j, m<n, j\leq m)$

$$\begin{aligned}
(A)A &\twoheadrightarrow \lambda x_2.\cdots\lambda x_m.\lambda y_2.\cdots\lambda y_m.(\lambda x_{m+1}.\cdots\lambda x_n.x_j)\lambda y_{m+1}.\cdots\lambda y_n.y_j \\
&\to \lambda x_2.\cdots\lambda x_m.\lambda y_2.\cdots\lambda y_m.\lambda x_{m+2}.\cdots\lambda x_n.x_j \\
&\equiv \lambda x_2.\cdots\lambda x_{j-1}.\underline{\lambda x_j.\cdots\lambda x_m.\lambda y_2.\cdots\lambda y_m.\lambda x_{m+2}.\cdots\lambda x_n.x_j} \\
&\equiv \lambda^{j-2}.\underline{\lambda x_1.\cdots\lambda x_{m+n-j-1}.x_1} \\
&\equiv \lambda^{j-2}.A^{m+n-j-1}_1
\end{aligned}$$

$\boxed{\text{III}}$

$$\underset{(1<i)}{A} \equiv A^{m,n}_{i,1} \equiv \lambda x_1.\cdots\lambda x_m.(x_i)\lambda x_{m+1}.\cdots\lambda x_n.x_1$$

$$\begin{aligned}
(A)A &\equiv (\lambda x_1.\cdots\lambda x_m.(x_i)\lambda x_{m+1}.\cdots\lambda x_n.x_1)A \\
&\to \lambda x_2.\cdots\lambda x_m.(x_i)\lambda x_{m+1}.\cdots\lambda x_n.A \\
&\equiv \lambda x_2.\cdots\lambda x_{i-1}.\underline{\lambda x_i.\cdots\lambda x_m.(x_i)}\lambda x_{m+1}.\cdots\lambda x_n.A \\
&\equiv \lambda^{i-2}.\underline{\lambda x_1.\cdots\lambda x_{m-i+1}.(x_1)}\lambda^{n-m}.A \\
&\equiv \lambda^{i-2}.\lambda x_1.\lambda^{m-i}.(x_1)\lambda^{n-m}.A
\end{aligned}$$

$\boxed{\text{IV}}$

$$A \equiv A^{m,n}_{1,1} \equiv \lambda x_1.\cdots\lambda x_m.(x_1)\lambda x_{m+1}.\cdots\lambda x_n.x_1$$

$$\begin{aligned}
(A)A &\equiv (\lambda x_1.\cdots\lambda x_m.(x_1)\lambda x_{m+1}.\cdots\lambda x_n.x_1)A \\
&\to \lambda x_2.\cdots\lambda x_m.(A)\lambda x_{m+1}.\cdots\lambda x_n.A \\
&\equiv \lambda^{m-1}.(A)\lambda^{n-m}.A
\end{aligned}$$

<u>IV.1</u>
$(m=n)$

$$\begin{aligned}
(A)A &\twoheadrightarrow \lambda^{m-1}.(A)\lambda^{n-m}.A \\
&\equiv \lambda^{n-1}.(A)A \quad \text{(no nf)}
\end{aligned}$$

<u>IV.2</u>
$(m<n)$

$$\begin{aligned}
(A)A &\twoheadrightarrow \lambda^{m-1}.(A)\lambda^{n-m}.A \\
&\equiv \lambda^{m-1}.(\lambda x_1.\cdots\lambda x_m.(x_1)\lambda x_{m+1}.\cdots\lambda x_n.x_1)\lambda^{n-m}.A \\
&\to \lambda^{m-1}.\lambda x_2.\cdots\lambda x_m.(\lambda^{n-m}.A)\lambda x_{m+1}.\cdots\lambda x_n.\lambda^{n-m}.A \\
&\equiv \lambda^{m-1}.\lambda^{m-1}.(\lambda^{n-m}.A)\lambda^{n-m}.\lambda^{n-m}.A \\
&\to \lambda^{m-1}.\lambda^{m-1}.\lambda^{n-m-1}.A \\
&\equiv \lambda^{m+n-3}.A
\end{aligned}$$

Calculation of $(A)A', \ldots$ for the cases (II) – (IV)

$\boxed{\text{II.1}}$

$A \equiv \underset{(1<j)}{A^{n,n}_{1,j}} \equiv \lambda x_1.\cdots\lambda x_n.(x_1)x_j$

$(A)A \twoheadrightarrow A' \equiv \lambda^{j-2}.A^{2n-j,2n-j}_{1,n}$

$\underset{(j=n)}{\underline{\text{II.1}'}}$

$A \equiv \lambda x_1.\cdots\lambda x_n.(x_1)x_n$

$(A)A \twoheadrightarrow A' \equiv \lambda^{n-2}.A$

$$\begin{array}{rcl} (A)\lambda^k.A & \equiv & (\lambda x_1.\cdots\lambda x_n.(x_1)x_n)\lambda^k.A \quad {\scriptstyle (k>0)} \\ & \to & \lambda x_2.\cdots\lambda x_n.(\lambda^k.A)x_n \\ & \to & \lambda x_2.\cdots\lambda x_n.\lambda^{k-1}.A \\ & \equiv & \lambda^{n-1}.\lambda^{k-1}.A \\ & \equiv & \lambda^{k+n-2}.A \end{array}$$

$\underset{(j<n)}{\underline{\text{II.1}''}}$

$A \equiv A^{n,n}_{1,j} \equiv \lambda x_1.\cdots\lambda x_n.(x_1)x_j$

$(A)A \twoheadrightarrow A' \equiv \lambda^{j-2}.A^{2n-j,2n-j}_{1,n}$

$$\begin{array}{rcl} (A^{n,n}_{1,j})\lambda^k.X & \equiv & (\lambda x_1.\cdots\lambda x_n.(x_1)x_j)\lambda^k.X \quad {\scriptstyle (k>0)} \\ & \to & \lambda x_2.\cdots\lambda x_n.(\lambda^k.X)x_j \\ & \to & \lambda x_2.\cdots\lambda x_n.\lambda^{k-1}.X \\ & \equiv & \lambda^{n-1}.\lambda^{k-1}.X \\ & \equiv & \lambda^{k+n-2}.X \end{array}$$

$$
\begin{aligned}
(A^{n,n}_{1,j})A^{n',n'}_{1,j'} &\equiv (\lambda x_1.\cdots\lambda x_n.(x_1)x_j)A^{n',n'}_{1,j'} \\
&\to \lambda x_2.\cdots\lambda x_n.(A^{n',n'}_{1,j'})x_j \\
&\equiv \lambda x_2.\cdots\lambda x_n.(\lambda y_1.\cdots\lambda y_{n'}.(y_1)y_{j'})x_j \\
&\to \lambda x_2.\cdots\lambda x_n.\lambda y_2.\cdots\lambda y_{n'}.(x_j)y_{j'} \\
&\equiv \lambda x_2.\cdots\lambda x_{j-1}.\underline{\lambda x_j.\cdots\lambda x_n.\lambda y_2.\cdots\lambda y_{j'-1}.\lambda y_{j'}.\cdots\lambda y_{n'}.(x_j)y_{j'}} \\
&\equiv \lambda^{j-2}.\underline{\lambda x_1.\cdots\lambda x_{n-j+1}.\lambda^{j'-2}.\lambda y_1.\cdots\lambda y_{n'-j'+1}.(x_1)y_1} \\
&\equiv \lambda^{j-2}.A^{n'+n-j,n'+n-j}_{1,j'+n-j}
\end{aligned}
$$

$\boxed{\text{II.2}}$

$A \equiv A^{m,n}_{1,j} \equiv \lambda x_1.\cdots\lambda x_m.(x_1)\lambda x_{m+1}.\cdots\lambda x_n.x_j$
$(m<n, m+1\leq j)$

$A \equiv \lambda x_1.\lambda^{m-1}.(x_1)\lambda^{j-m-1}.A^{n-j+1}_1$

$(A)A \twoheadrightarrow A' \equiv \lambda^{m+j-4}.A^{n-j+1}_1$

<u>II.2′</u>
$(j=n)$

$A \equiv \lambda x_1.\lambda^{m-1}.(x_1)\lambda^{n-m-1}.A^1_1$

$(A)A \twoheadrightarrow A' \equiv \lambda^{m+n-4}.A^1_1$

$$
\begin{aligned}
(A)\lambda^k.X &\equiv (\lambda x_1.\lambda^{m-1}.(x_1)\lambda^{n-m-1}.A^1_1)\lambda^k.X \quad {\scriptstyle (k>0)} \\
&\to \lambda^{m-1}.(\lambda^k.X)\lambda^{n-m-1}.A^1_1 \\
&\to \lambda^{m-1}.\lambda^{k-1}.X \\
&\equiv \lambda^{k+m-2}.X
\end{aligned}
$$

$$
\begin{aligned}
(A)A^1_1 &\equiv (\lambda x_1.\lambda^{m-1}.(x_1)\lambda^{n-m-1}.A^1_1)A^1_1 \\
&\to \lambda^{m-1}.(A^1_1)\lambda^{n-m-1}.A^1_1 \\
&\to \lambda^{m-1}.\lambda^{n-m-1}.A^1_1 \\
&\equiv \lambda^{n-2}.A^1_1
\end{aligned}
$$

$\underline{\text{II.2}''}$

$(j<n)$

$A \equiv \lambda x_1.\lambda^{m-1}.(x_1)\lambda^{j-m-1}.A_1^{n-j+1}$

$(A)A \twoheadrightarrow A' \equiv \lambda^{m+j-4}.A_1^{n-j+1}$

$$\begin{aligned}(A)\lambda^k.X &\equiv (\lambda x_1.\lambda^{m-1}.(x_1)\lambda^{j-m-1}.A_1^{n-j+1})\lambda^k.X \quad {\scriptstyle (k>0)}\\ &\to \lambda^{m-1}.(\lambda^k.X)\lambda^{j-m-1}.A_1^{n-j+1}\\ &\to \lambda^{m-1}.\lambda^{k-1}.X\\ &\equiv \lambda^{k+m-2}.X\end{aligned}$$

$$\begin{aligned}(A)A_1^{n-j+1} &\equiv (\lambda x_1.\lambda^{m-1}.(x_1)\lambda^{j-m-1}.A_1^{n-j+1})A_1^{n-j+1}\\ &\to \lambda^{m-1}.(A_1^{n-j+1})\lambda^{j-m-1}.A_1^{n-j+1}\\ &\to \lambda^{m-1}.\lambda^{n-j}.\lambda^{j-m-1}.A_1^{n-j+1}\\ &\equiv \lambda^{n-2}.A_1^{n-j+1}\end{aligned}$$

$\boxed{\text{II.3}}$

$A \equiv A_{1,j}^{m,n} \equiv \lambda x_1.\cdots\lambda x_m.(x_1)\lambda x_{m+1}.\cdots\lambda x_n.x_j$

$(m<n, 1<j\leq m)$

$A \equiv \lambda x_1.\cdots\lambda x_m.(x_1)\lambda^{n-m}.x_j$

$(A)A \twoheadrightarrow A' \equiv \lambda^{j-2}.A_1^{m+n-j-1}$

$$\begin{aligned}(A)\lambda^k.X &\equiv (\lambda x_1.\cdots\lambda x_m.(x_1)\lambda^{n-m}.x_j)\lambda^k.X \quad {\scriptstyle (k>0)}\\ &\to \lambda x_2.\cdots\lambda x_m.(\lambda^k.X)\lambda^{n-m}.x_j\\ &\to \lambda x_2.\cdots\lambda x_m.\lambda^{k-1}.X\\ &\equiv \lambda^{m-1}.\lambda^{k-1}.X\\ &\equiv \lambda^{k+m-2}.X\end{aligned}$$

$$\begin{aligned}(A)A_1^{n'} &\equiv (\lambda x_1.\cdots\lambda x_m.(x_1)\lambda^{n-m}.x_j)A_1^{n'}\\ &\to \lambda x_2.\cdots\lambda x_m.(A_1^{n'})\lambda^{n-m}.x_j\\ &\to \lambda x_2.\cdots\lambda x_m.\lambda^{n'-1}.\lambda^{n-m}.x_j\\ &\equiv \lambda x_2.\cdots\lambda x_{j-1}.\underline{\lambda x_j.\cdots\lambda x_m.\lambda^{n'-1}.\lambda^{n-m}.x_j}\\ &\equiv \lambda^{j-2}.\underline{\lambda x_1.\cdots\lambda x_{m-j+1}.\lambda^{n'-1}.\lambda^{n-m}.x_1}\\ &\equiv \lambda^{j-2}.A_1^{n'+n-j}\end{aligned}$$

$\boxed{\text{III}}$

$A \equiv A^{m,n}_{i,1} \equiv \lambda x_1.\cdots\lambda x_m.(x_i)\lambda x_{m+1}.\cdots\lambda x_n.x_1$
$_{(1<i)}$

$A \equiv \lambda x_1.\cdots\lambda x_m.(x_i)\lambda^{n-m}.x_1$

$(A)A \twoheadrightarrow A' \equiv \lambda^{i-2}.\lambda x_1.\lambda^{m-i}.(x_1)\lambda^{n-m}.A$

<u>III.1</u>
$_{(m=n)}$

$A \equiv \lambda x_1.\cdots\lambda x_n.(x_i)x_1$

$(A)A \twoheadrightarrow A' \equiv \lambda^{i-2}.\lambda x_1.\lambda^{n-i}.(x_1)A$

$$\begin{aligned}
(A)X &\equiv (\lambda x_1.\cdots\lambda x_n.(x_i)x_1)X, \quad X \equiv A, A', \ldots\\
&\to \lambda x_2.\cdots\lambda x_n.(x_i)X\\
&\equiv \lambda x_2.\cdots\lambda x_{i-1}.\lambda x_i.\cdots\lambda x_n.(x_i)X\\
&\equiv \lambda^{i-2}.\lambda x_1.\cdots\lambda x_{n-i+1}.(x_1)X\\
&\equiv \lambda^{i-2}.\lambda x_1.\lambda^{n-i}.(x_1)X
\end{aligned}$$

$\tilde{X} \equiv \lambda x_1.\lambda^{n-i}.(x_1)X$

$$\begin{aligned}
(\tilde{X})A &\equiv (\lambda x_1.\lambda^{n-i}.(x_1)X)A\\
&\to \lambda^{n-i}.(A)X\\
&\to \lambda^{n-i}.\lambda^{i-2}.\lambda x_1.\lambda^{n-i}.(x_1)X\\
&\equiv \lambda^{n-2}.\lambda x_1.\lambda^{n-i}.(x_1)X
\end{aligned}$$

$$\begin{aligned}
(\tilde{X})\lambda^k.Y &\equiv (\lambda x_1.\lambda^{n-i}.(x_1)X)\lambda^k.Y \quad {\scriptstyle (k>0)}\\
&\to \lambda^{n-i}.(\lambda^k.Y)X\\
&\to \lambda^{n-i}.\lambda^{k-1}.Y\\
&\equiv \lambda^{k+n-i-1}.Y
\end{aligned}$$

$$\begin{aligned}
(\tilde{X})\tilde{Y} &\equiv (\lambda x_1.\lambda^{n-i}.(x_1)X)\tilde{Y}\\
&\to \lambda^{n-i}.(\tilde{Y})X\\
&\equiv \lambda^{n-i}.(\lambda x_1.\lambda^{n-i}.(x_1)Y)X\\
&\to \lambda^{n-i}.\lambda^{n-i}.(X)Y\\
&\equiv \lambda^{2n-2i}.(X)Y \twoheadrightarrow \ldots
\end{aligned}$$

III.2

$(m<n)$

$A \equiv \lambda x_1.\cdots\lambda x_m.(x_i)\lambda^{n-m}.x_1$

$(A)A \twoheadrightarrow A' \equiv \lambda^{i-2}.\lambda x_1.\lambda^{m-i}.(x_1)\lambda^{n-m}.A$

$$\begin{array}{lcl}
(A)X & \equiv & (\lambda x_1.\cdots\lambda x_m.(x_i)\lambda^{n-m}.x_1)X, \quad X \equiv A, A', \ldots \\
 & \to & \lambda x_2.\cdots\lambda x_m.(x_i)\lambda^{n-m}.X \\
 & \equiv & \lambda x_2.\cdots\lambda x_{i-1}.\lambda x_i.\cdots\lambda x_m.(x_i)\lambda^{n-m}.X \\
 & \equiv & \lambda^{i-2}.\lambda x_1.\cdots\lambda x_{m-i+1}.(x_1)\lambda^{n-m}.X \\
 & \equiv & \lambda^{i-2}.\lambda x_1.\lambda^{m-i}.(x_1)\lambda^{n-m}.X
\end{array}$$

$\tilde{X} \equiv \lambda x_1.\lambda^{m-i}.(x_1)\lambda^{n-m}.X$

$$\begin{array}{lcl}
(\tilde{X})A & \equiv & (\lambda x_1.\lambda^{m-i}.(x_1)\lambda^{n-m}.X)A \\
 & \to & \lambda^{m-i}.(A)\lambda^{n-m}.X \\
 & \to & \lambda^{m-i}.\lambda^{i-2}.\lambda x_1.\lambda^{m-i}.(x_1)\lambda^{n-m}.\lambda^{n-m}.X \\
 & \equiv & \lambda^{m-2}.\lambda x_1.\lambda^{m-i}.(x_1)\lambda^{2n-2m}.X
\end{array}$$

$$\begin{array}{lcl}
(\tilde{X})\lambda^k.Y & \equiv & (\lambda x_1.\lambda^{m-i}.(x_1)\lambda^{n-m}.X)\lambda^k.Y \quad {\scriptstyle (k>0)} \\
 & \to & \lambda^{m-i}.(\lambda^k.Y)\lambda^{n-m}.X \\
 & \to & \lambda^{m-i}.\lambda^{k-1}.Y \\
 & \equiv & \lambda^{k+m-i-1}.Y
\end{array}$$

$$\begin{array}{lcl}
(\tilde{X})\tilde{Y} & \equiv & (\lambda x_1.\lambda^{m-i}.(x_1)\lambda^{n-m}.X)\tilde{Y} \\
 & \to & \lambda^{m-i}.(\tilde{Y})\lambda^{n-m}.X \\
 & \equiv & \lambda^{m-i}.(\lambda x_1.\lambda^{m-i}.(x_1)\lambda^{n-m}.Y)\lambda^{n-m}.X \\
 & \to & \lambda^{m-i}.\lambda^{m-i}.(\lambda^{n-m}.X)\lambda^{n-m}.Y \\
 & \to & \lambda^{m-i}.\lambda^{m-i}.\lambda^{n-m-1}.X \\
 & \equiv & \lambda^{m+n-2i-1}.X
\end{array}$$

$\boxed{\text{IV.1}}$

$A \equiv A^{n,n}_{1,1} \equiv \lambda x_1. \cdots \lambda x_n.(x_1)x_1$

$A \equiv \lambda x_1.\lambda^{n-1}.(x_1)x_1$

$(A)A \twoheadrightarrow A' \equiv \lambda^{n-1}.(A)A \quad \text{(no nf)}$

$\underline{\text{I} \Rightarrow \text{IV.1}}$

$\tilde{A} \equiv \lambda^p.A \quad {\scriptstyle (p>0)}$

$A \equiv A^{n,n}_{1,1} \equiv \lambda x_1.\lambda^{n-1}.(x_1)x_1$

$(A)A \twoheadrightarrow \lambda^{n-1}.(A)A \quad \text{(no nf)}$

$$\begin{array}{rcl}(A)\lambda^k.A & \equiv & (\lambda x_1.\lambda^{n-1}.(x_1)x_1)\lambda^k.A \quad {\scriptstyle (k>0)} \\ & \to & \lambda^{n-1}.(\lambda^k.A)\lambda^k.A \\ & \to & \lambda^{n-1}.\lambda^{k-1}.A \\ & \equiv & \lambda^{k+n-2}.A\end{array}$$

$\boxed{\text{IV.2}}$

$A \equiv A^{m,n}_{1,1} \equiv \lambda x_1. \cdots \lambda x_m.(x_1)\lambda x_{m+1}. \cdots \lambda x_n.x_1$
${\scriptstyle (m<n)}$

$A \equiv \lambda x_1.\lambda^{m-1}.(x_1)\lambda^{n-m}.x_1$

$(A)A \twoheadrightarrow A' \equiv \lambda^{m+n-3}.A$

$$\begin{array}{rcl}(A)\lambda^k.A & \equiv & (\lambda x_1.\lambda^{m-1}.(x_1)\lambda^{n-m}.x_1)\lambda^k.A \quad {\scriptstyle (k>0)} \\ & \to & \lambda^{m-1}.(\lambda^k.A)\lambda^{n-m}.\lambda^k.A \\ & \to & \lambda^{m-1}.\lambda^{k-1}.A \\ & \equiv & \lambda^{k+m-2}.A\end{array}$$

References

[1] P. Aczel. Frege structures and the notions of proposition, truth and set. In J. Barwise, H. J. Keisler, and K. Kunen, editors, *The Kleene Symposium*, Studies in Logic 101, pages 31–60. North-Holland, Amsterdam, 1980.

[2] J. Avenhaus. *Reduktionssysteme. Rechnen und Schließen in gleichungsdefinierten Strukturen.* Springer Verlag, Berlin, 1995.

[3] H. G. Barendregt. *The Lambda Calculus. Its Syntax and Semantics.* Studies in Logic 103. North-Holland, Amsterdam, second revised edition, 1984.

[4] P. Bro. Artificial life in real chemical reaction systems. Book manuscript in preparation (contact address: P. Bro, Santa Fe Institute, 1399 Hyde Park Road, Santa Fe NM 87501), 1995.

[5] W. Burge. *Recursive Programming Techniques.* Addison-Wesley, Reading, MA, 1978.

[6] L. Cardelli. Type systems. In A. B. Tucker Jr., editor, *The Handbook of Computer Science and Engineering*, pages 2208–2236. CRC Press, 1997.

[7] A. Church. A set of postulates for the foundation of logic. *Annals of Math. (2)*, 33:346–366, 34:839–864, 1932.

[8] A. Church. *The Calculi of Lambda Conversion.* Princeton University Press, Princeton, 1941.

[9] J. N. Crossley, editor. *Algebra and Logic.* Lecture Notes in Mathematics 450. Springer Verlag, Berlin, 1975.

[10] J. P. Crutchfield and M. Mitchell. The evolution of emergent computation. *Proc. Natl. Acad. Sci. USA*, 92:10742–10746, 1995.

[11] H. B. Curry. Grundlagen der kombinatorischen Logik. *Amer. J. Math.*, 52:509–536, 789–834, 1930.

[12] H. B. Curry and R. Feys. *Combinatory Logic, Vol. I.* North-Holland, Amsterdam, 1958.

[13] H. B. Curry, J. R. Hindley, and J. P. Seldin. *Combinatory Logic, Vol. II*. Studies in Logic 65. North-Holland, Amsterdam, 1972.

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

[15] R. A. Fisher. *The Genetical Theory of Natural Selection*. Clarendon Press, Oxford, 1930.

[16] W. Fontana and L. W. Buss. 'The arrival of the fittest': Toward a theory of biological organization. *Bull. Math. Biol.*, 56:1–64, 1994.

[17] W. Fontana and L. W. Buss. What would be conserved 'if the tape were played twice'. *Proc. Natl. Acad. Sci. USA*, 91:757–761, 1994.

[18] W. Fontana and L. W. Buss. The barrier of objects: From dynamical systems to bounded organizations. In J. Casti and A. Karlqvist, editors, *Barriers and Boundaries*, pages 56–116. Addison-Wesley, Reading, MA, 1996.

[19] W. Fontana and P. Schuster. Continuity in evolution: On the nature of transitions. *Science*, 280:1451–1455, 1998.

[20] W. Fontana, G. Wagner, and L. W. Buss. Beyond digital naturalism. *Artificial Life*, 1:211–227, 1994.

[21] G. Frege. *Grundgesetze der Arithmetik. Begriffsschriftlich abgeleitet.* Pohle, Jena, 1893/1903.

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

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

[24] M. J. C. Gordon. *Evaluation and Denotation of Pure LISP. A Worked Example in Semantics.* PhD thesis, University of Edinburgh, 1973.

[25] M. J. C. Gordon. *The Denotational Description of Programming Languages.* Springer Verlag, Berlin, 1979.

[26] J. B. S. Haldane. *The Causes of Evolution*. Longmans Green, New York, 1932.

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

[28] J. H. Holland, K. J. Holyoak, and R. E. Nisbett. *Induction : Processes of Inference, Learning and Discovery*. MIT Press, reprinted 1989.

[29] S. C. Kleene. λ-definability and recursiveness. *Duke Math. J.*, 2:340–353, 1936.

[30] S. C. Kleene and J. B. Rosser. The inconsistency of certain formal logics. *Annals of Math. (2)*, 36:630–636, 1935.

[31] J. W. Klop. *Combinatory Reduction Systems*. CWI Report, Amsterdam, 1980.

[32] J. W. Klop. Term rewriting systems. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 1–116. Clarendon Press, Oxford, 1992.

[33] J. W. Klop and A. Middeldorp. An introduction to Knuth-Bendix completion. *CWI Quarterly*, 1, 1988.

[34] D. E. Knuth and P. E. Bendix. Simple word problems in universal algebra. In J. Leech, editor, *Computational Problems in Abstract Algebra*. Pergamon Press, New York, 1970.

[35] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, Mass., 1992.

[36] P. J. Landin. A correspondence between ALGOL 60 and Church's lambda notation. *Comm. Assoc. Comput. Mach.*, 8:89–101, 158–165, 1965.

[37] K. Lindgren. Evolutionary phenomena in simple dynamics. In C. G. Langton, C. Taylor, J. D. Farmer, and S. Rasmussen, editors, *Artificial Life II*, Santa Fe Institute Studies in the Sciences of Complexity, pages 295–312, Redwood City, 1992. Addison-Wesley.

[38] R. E. Milne and C. Strachey. *A Theory of Programming Language Semantics*. 2 vols. Chapman and Hall, London; Wiley, New York, 1976.

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

[40] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1996.

[41] J. Monod. *Chance and Necessity*. page 119, emphasis by the author. Alfred Knopf, New York, 1971.

[42] J.-H. Morris. *Lambda Calculus Models of Programming Languages*. PhD thesis, MIT, 1968.

[43] M. J. O'Donnell. *Computing in Systems Described by Equations*. Lecture Notes in Computer Science 58. Springer Verlag, Berlin, 1977.

[44] A. Ollongren. *A Definition of Programming Languages by Interpreting Automata*. Academic Press, New York and London, 1975.

[45] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theor. Comput. Sci.*, 1:125–159, 1975.

[46] T. S. Ray. Evolution and optimization of digital organisms. In K. R. Billingsley, E. Derohanes, and H. Brown III, editors, *Scientific Excellence in Supercomputing: The IBM 1990 Contest Prize Papers*, pages 489–531, Athens, GA, 1991. The Baldwin Press.

[47] J. Rees. A Reversible Object Interaction Calculus. Santa Fe Institute Working Paper, 1998.

[48] G. E. Revesz. *Lambda Calculus, Combinators and Functional Programming*. Cambridge Tracts in Theoretical Computer Science 4. Cambridge University Press, Cambridge, 1988.

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

[50] D. S. Scott. Combinators and classes. In C. Böhm, editor, *λ-calculus and Computer Science Theory*, Lecture Notes in Computer Science 37, pages 1–26. Springer Verlag, Berlin, 1975.

[51] D. S. Scott and C. Strachey. Toward a mathematical semantics for computer languages. *Proc. Symp. on Computers and Automata, Polytechnic Institute of Brooklyn*, 21:19–46, 1971.

[52] J. P. Seldin. Recent advances in Curry's program. *Rend. Sem. Mat. Univers. Politecn. Torino*, 35:77–88, 1976.

[53] J. E. Stoy. *Denotational Semantics. The Scott-Strachey Approach to Programming Languages*. MIT Press, Cambridge, 1977.

[54] A. M. Turing. On computable numbers with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 42:230–265, 1936.

[55] A. M. Turing. Computability and λ-definability. *J. Symbolic Logic*, 2:153–163, 1937.

[56] S. Wright. Evolution in mendelian populations. *Genetics*, 16:97–159, 1931.

Curriculum vitae

Stefan Müller

*8. 10. 1969
Wels, O.Ö.

Schulbildung

1976 – 1980	Volksschule Gunskirchen
1980 – 1988	Realistisches Gymnasium, BRG Wels Reifeprüfung mit ausgezeichnetem Erfolg bestanden

Studium

1989 – 1995	Technische Physik, TU Wien Diplomprüfungen mit Auszeichnung bestanden Diplomarbeit am Atominstitut der öst. Universitäten *Bau eines Meßgeräts zur photoplethysmographischen Untersuchung der peripheren Blutpulsdynamik*
1991 – 1995	Biologie, Universität Wien
1995 – 1996	Betriebs-, Rechts- und Wirtschaftswissenschaften TU Wien
1996 – 1999	Doktorat Chemie, Universität Wien Dissertation am Institut für Theoretische Chemie *Functional Organization in Molecular Systems and the λ-calculus*
Juni 1997	Complex Systems Summer School Santa Fe Institute, Santa Fe, New Mexico, USA

Forschungstätigkeit

1998	Wissenschaftlicher Mitarbeiter *The Study of Adaptation and Self-organization* International Institute for Applied Systems Analysis Laxenburg