The Book Review Column[1]

by William Gasarch
Department of Computer Science
University of Maryland at College Park
College Park, MD, 20742
email: `gasarch@cs.umd.edu`

In this column we review the following books.

1. **Number Theory for Computing** by Song Y. Yan. Review by Prodromos Saridis. This is a book on Number Theory which looks at computational issues and crytography.
2. **Type-Logical Semantics** by B. Carpenter. Review by Riccardo Pucella and Stephen Chong. This is a book about applying logic and type theory to semantics in linguistics.
3. **The π-calculus: A Theory of Mobile Processes** by D. Sangiorgi and D. Walker. Review by Riccardo Pucella. This is a book about a way to formalize reasoning about mobile systems.

Books I want Reviewed

If you want a FREE copy of one of these books in exchange for a review, then email me at gasarch@cs.umd.edu
Reviews need to be in LaTeX, LaTeX2e, or Plaintext.

Books on Algorithms, Combinatorics, and Related Fields

1. *Diophantine Equations and Power Integral Bases* by Gaal.
2. *An Introduction to Data Structures and Algorithms* by Storer.
3. *Computational Line Geometry* by Pottmann and Wallner.
4. *Linear Optimization and Extensions: Problems and Solutions* by Alevras and Padberg.

Books on Cryptography and Books on Learning

1. *Cryptography: Theory and Practice* (Second Edition) by Stimpson.
2. *Cryptanalysis of Number Theoretic Ciphers* by Wagstaff.
3. *Learning with Kernels (Support Vector Machines, Regularization, Optimization, and Beyond)* Bernard Scholkopf and Alexander Smola.
4. *Learning Kernel Classifiers* by Herbrich.

Books on Complexity and Logic

1. *Logic and Language Models for Computer Science* by Hamburger and Richards.
2. *Automata Theory and its Applications* by Khoussainov and Nerode.
3. *Introduction to Language, Machines, and Logic* by Parkes

[1]© William Gasarch, 2003.

Review[2] of
Number Theory for Computing
Author of Book: Song Y. Yan
Publisher: Springer (381 pages)
Author of Review: Prodromos Saridis

1 Overview

The theory of numbers, in mathematics, is primarily the theory of the properties of integers. The subject has long been considered as the *purest* branch of mathematics, with very few applications to other areas. However recent years have seen considerable increase in interest in several central topics of number theory, precisely because of their importance and applications in other areas, particularly in computing and information technology.

This book takes the reader from elementary number theory in computer science, via algorithmic number theory, to applied number theory in computer science. The book follows the style "Definition–Theorem–Algorithm–Example" rather than the "Definition–Theorem–Proof" style. For a reader who has at least high-school math background the derivations in the book will not be difficult to follow.

2 Summary of Contents

The book is divided into three parts, each consisting of multiple chapters.

2.1 Part I. Elementary Number Theory

(Introduction / Theory of Divisibility / Diophantine Equations / Arithmetic Functions / Distribution of Prime Numbers / Theory of Congruences / Arithmetic of Elliptic Curves / Bibliographic Notes and Further Reading.)

These preliminary chapters introduce the fundamental concepts and results of divisibility theory, congruence theory, continued fractions, Diophantine equations and elliptic curves. The reader becomes familiar with the terminology. A novel feature is that a whole chapter is devoted to Elliptic Curves, which is not normally provided by an elementary number theory book. This part carefully and concisely defines all the tools (e.g. theorems, concepts etc) in order to build the essential intuition which is so necessary for the later chapters.

2.2 Part II. Algorithmic Number Theory

(Introduction / Algorithms for Primality Testing / Algorithms for Integer Factorization / Algorithms for Discrete Logarithms / Quantum Number–Theoretic Algorithms / Miscellaneous Algorithms in Number Theory.)

The second part of the book briefly introduces the concepts of algorithms and complexity and presents some important and widely used algorithms in computational number theory. The author recommends readers to implement all the algorithms introduced here.

[2] ©Prodromos Saridis , 2003

2.3 Part III. Applied Number Theory

(Why Applied Number Theory / Computer Systems Design / Cryptography and Information Security.)

While the earlier chapters covered the broad spectrum of number theory basics, within this part, the book shifts gears into more specialized applications of number theory. Here is a taste of the contents: Residue Computers / Hashing Functions / Error Detection / Random Number Generation / Secret-Key and Public-Key Cryptography / DES/AES / Various Cryptosystems / Digital Signatures / Database Security / Internet, Web Security and e-Commerce / Steganography / Quantum Cryptography.

3 Opinion

Number Theory for Computing by Yan, is a cohesive introduction to a field whose importance has increased in the last years. I enjoyed reading this book, not only because it is well written and easy to read but also because there are almost 80 mini-biographies of researchers, scientists, and number theorists who played (and some of the still playing) great role to the development of number theory. E.g. it is easier for the student to remember a theorem proved by a number theorist, if the student knows more about the life and the research interests of that particular number theorist.

I believe however, that especially in the 1st part of the book, the breadth of coverage makes it difficult for an undergraduate to learn number theory from this book. This is mainly because of the style of the book (Definition-Theorem-Example) and the scarcity of exercises. On the other hand, the first part of the book is an excellent reference for researchers. It provides almost allthe necessary theorems without unnecessary details.

Although the coverage of the materials is not exhaustive the author achieves to present the topics in an attractive way to the reader. The citation references are sufficient and range from old references to fairly modern. Also the selection of the applications presented is quite modern. Overall, the book is well written and easy to understand.

Review[3] of
Type-Logical Semantics
Author of Book: B. Carpenter
Publisher: MIT Press (585 pages)
Author of Review: Pucella and Chong, Dept of CS, Cornell Univ

4 Introduction

One of the many roles of linguistics is to address the semantics of natural languages, that is, the meaning of sentences in natural languages. An important part of the meaning of sentences can be characterized by stating the conditions that need to hold for the sentence to be true. Necessarily, this approach, called truth-conditional semantics, disregards some relevant aspects of meaning, but has been very useful in the analysis of natural languages. Structuralist views of language (the kind held by Saussure, for instance, and later Chomsky) have typically focused on phonology, morphology, and syntax. Little progress, however, has been shown towards the structure of meaning, or content.

A common tool for the study of content, and structure in general for that matter, has been logic. During most of the 20th century, an important role of logic has been to study the structure of content of mathematical languages. Many logicians have moved on to apply the techniques developed to the analysis of natural languages—Frege, Russell, Carnap, Reichenbach, and Montague. An early introduction to such classical approaches can be found in [2].

As an illustration of the kind of problems that need to be addressed, consider the following examples. The following two sentences assert the existence of a man that both walks and talks:

Some man that walks talks
Some man that talks walks

The situations with respect to which these two sentences are true are the same, and hence a truth-conditional semantics needs to assign the same meaning to such sentences. Ambiguities arise easily in natural languages:

Every man loves a woman

There are at least two distinct readings of this sentence. One says that for every man, there exists a woman that he loves, and the other says that there exists a woman that every man loves. Other problems are harder to qualify. Consider the following two sentences:

Tarzan likes Jane
Tarzan wants a girlfriend

The first sentence must be false if there is no Jane. On the other hand, the second sentence can be true even if no woman exists.

Those examples are extremely simple, some might even say naive, but they exemplify the issues for which a theory of natural language semantics must account. A guiding principle, apocryphally due to Frege, in the study of semantics is the so-called Fregean principle. Essentially, it can be stated as "the meaning of a complex expression should be a function of the meaning of its parts." Such a principle seems required to explain how natural languages can be learned. Since there is no arbitrary limit on both the length and the number of new sentences human beings can understand, some general principle such as the above must be at play. Moreover, since it would not be helpful to require an infinite number of functions to derive the meaning of the whole from the meaning of the parts, a notion such as recursion must be at play as well.

[3]©Riccardo Pucella and Stephen Chong, 2003

That there is a recursive principle *à la Frege* at play both in syntax and semantics is hardly contested. What is contested is the interplay between the two. The classic work by Chomsky [6] advocated essentially the autonomy of syntax with respect to semantics. Chomsky's grammars are transformational: they transform the "surface" syntax of a sentence to extract its so-called deep structure. The semantics is then derived from the deep structure of the sentence. Some accounts of the Chomsky theory allows for a Fregean principle to apply at the level of the deep structure, while more recent accounts slightly complicate the picture. A different approach is to advocate a close correspondence between syntax and semantics. Essentially, the syntax can be seen as a map showing how the meaning of the parts are to be combined into the meaning of the whole.

The latter approach to semantics relies on two distinct developments. First, it is based on a kind of semantic analysis of language originating mainly with the work of Montague [9]. His was the first work that developed a large scale semantic description of natural languages by translation into a logical language that can be given a semantics using traditional techniques. The second development emerged from a particular syntactic analysis of language. During his analysis of logic, which led to development of the λ-calculus, Curry noticed that the types he was assigning to λ-terms could also be used to denote English word classes [7]. For example, in *John snores loudly*, the word *John* has type n, *snores* has type $n \Rightarrow s$, and *loudly* has type $s \Rightarrow s$. Independently, Lambek introduced a calculus of syntactic types, distinguishing two kinds of implication, reflecting the non-commutativity of concatenation [8]. The idea was to push all the grammar into the dictionary, assigning to each English word one or more types, and using the calculus to decide whether a string of words is a grammatically well-formed sentence. This work derived in part from earlier work by Ajdukiewicz [1] and Bar-Hillel [4].

This book, "Type-Logical Semantics" by Carpenter, explores this particular approach. Essentially, it relies on techniques from type theory: we assign a type (or more than one) to every word in the language, and we can check that a sentence is well-formed by performing what amounts to type-checking. In fact, it turns out that we can take the type-checking derivation proving that a sentence has the right type, and use the derivation to derive the semantics of the sentence. In the next sections, we will introduce the framework, and give simple examples to highlight the ideas. Carpenter pushes these ideas quite far, as we shall see when we cover the table of contents. We conclude with some opinions on the book.

5 To semantics...

The first problem we need to address is how to describe the semantics of language. We will follow in the truth-conditional tradition and model-theoretic ideas and we start with first-order logic. Roughly speaking, first-order logic provides one with constants denoting individuals, and predicates over such individuals. Simple example should illustrate this. Consider the sentence *Tarzan likes Jane*. Assuming constants $\mathbf{tarzan}$ and $\mathbf{jane}$, and a predicate $\mathbf{like}$, this sentence corresponds to the first-order logic formula $\mathbf{like}(\mathbf{tarzan}, \mathbf{jane})$. The sentence *Everyone likes Jane* can be expressed as $\forall x.\mathbf{like}(x, \mathbf{jane})$. This approach of using first-order logic to give semantics is quite straightforward. Unfortunately, for our purposes, it is also quite deficient. Let us see two reasons why that is. First, recall that we want a compositional principle at work in semantics. In other words, we want to be able to derive the meaning of *Tarzan likes Jane* from the meaning of *Tarzan* and *Jane*, and the meaning of *likes*. This sounds straightforward. However, the same principle should apply to the sentence *Tarzan and Kala like Jane*, corresponding to the first-order formula $\mathbf{like}(\mathbf{tarzan}, \mathbf{jane}) \wedge \mathbf{like}(\mathbf{kala}, \mathbf{jane})$. Giving a compositional semantics seems to require giving a semantics to the extract *like Jane*. What is the semantics of such a part of speech? First-order logic cannot answer this easily. Informally, *like Jane* should have as semantics something that expects an individual (say x) and gives back the formula $\mathbf{like}(x, \mathbf{jane})$. A second problem is that the grammatical structure of a sentence can be lost during translation. This can lead to wild differences in semantics for similar sentences. For instance, consider the following sentences:

Tarzan likes Jane.
An apeman likes Jane.
Every apeman likes Jane.
No apeman likes Jane.

These sentences can be formalized as such in first-order logic, respectively:

$\mathbf{like}(\mathbf{tarzan}, \mathbf{jane})$
$(\exists x)(\mathbf{apeman}(x) \wedge \mathbf{like}(x, \mathbf{jane}))$
$(\forall x)(\mathbf{apeman}(x) \Rightarrow \mathbf{like}(x, \mathbf{jane}))$
$\neg(\exists x)(\mathbf{apeman}(x) \wedge \mathbf{like}(x, \mathbf{jane}))$, or equivalently, $(\forall x)(\mathbf{apeman}(x) \Rightarrow \neg\mathbf{like}(x, \mathbf{jane}))$

There seems to be a discrepancy among the logical contributions of the subjects in the above sentences. There is a distinction in the first-order logic translation of these sentences that is not expressed by their grammatical form.

It turns out that there is a way to solve those problems, by looking at an extension of first-order logic, known as higher-order logic [3]. Let us give enough theory of higher-order logic to see how it can be used to assign semantics to (a subset of) a natural language. This presentation presupposes a familiarity with both first-order logic and the λ-calculus [5].[4] There is a slight difference in our approach to first-order logic and our approach to higher-order logic. In the former, formulas, which represents properties of the world and its individuals, are the basic units of the logic. In higher-order logics, terms are the basic units, including constants and functions, with formulas explicitly represented as boolean-valued functions.

We start by defining a set of types that will be used to characterize the well-formedness of formulas, as well as derive the models. We assume a set of basic types $\mathsf{BasTyp} = \{\mathit{Bool}, \mathit{Ind}\}$, where *Bool* is the type of boolean values, and *Ind* is the type of individuals. (In first-order logic, the type *Bool* is not made explicit.) The set of types Typ is the smallest set such that $\mathsf{BasTyp} \subseteq \mathsf{Typ}$, and $(\sigma \rightarrow \tau) \in \mathsf{Typ}$ if $\sigma, \tau \in \mathsf{Typ}$. A type of the form $\sigma \rightarrow \tau$ is a functional (or higher-order) type, the elements of which map objects of type σ to objects of type τ.

The syntax of higher-order logic is defined as follows. Assume for each type $\tau \in \mathsf{Typ}$ a set Var_τ of variables and a set Con_τ of constants of that type. The set Term_τ of terms of type τ is defined as the smallest set such that:

$\mathsf{Var}_\tau \subseteq \mathsf{Term}_\tau$,
$\mathsf{Con}_\tau \subseteq \mathsf{Term}_\tau$,
$\alpha\beta \in \mathsf{Term}_\tau$ if $\alpha \in \mathsf{Term}_{\sigma\rightarrow\tau}$ and $\beta \in \mathsf{Term}_\sigma$, and
$\lambda x.\alpha \in \mathsf{Term}_\tau$ if $\tau = \sigma \rightarrow \rho$, $x \in \mathsf{Var}_\sigma$, and $\alpha \in \mathsf{Term}_\rho$.

What are we doing here? We are defining a term language. First-order logic introduces special syntax for its logical connectives ($\wedge$, $\vee$, $\neg$, $\Rightarrow$). It turns out, for higher-order logic, that we do not need to do that, we can simply define constants for those operators. (We will call these logical constants, because they will have the same interpretation in all models.) We will assume the following constants, at the following types: a constant $\mathbf{not}$ of type $\mathit{Bool} \rightarrow \mathit{Bool}$ and a constant $\mathbf{and}$ of type $\mathit{Bool} \rightarrow \mathit{Bool} \rightarrow \mathit{Bool}$, the interpretation of which should be clear, a family of constants $\mathbf{eq}_\tau$ each of type $\tau \rightarrow \tau \rightarrow \mathit{Bool}$, which checks for equality of two elements of type τ, and a family of constants $\mathbf{every}_\tau$ each of type $(\tau \rightarrow \mathit{Bool}) \rightarrow \mathit{Bool}$, used to capture universal quantification. The idea is that $\mathbf{every}_\tau$ quantifies over objects of type τ. In first-order

[4] Higher-order logic is interesting in that it can either be viewed as a generalization of first-order logic, or as a particular instance of the simply-typed λ-calculus.

logic, $\forall x.\varphi$ is true if for every possible individual i, replacing x by i in φ yields a true formula. Note that $\forall x$ binds the variable x in first-order logic. In higher-order logic, where there is only λ as a binder, we write the above as $\mathbf{every}_\tau(\lambda x.\varphi)$, which is true if φ is true for all objects of type τ.

This defines the syntax of higher-order logic. The models of higher-order logic are generalizations of the relational structures used to model first-order logic. A (standard) frame for higher-order logic is specified by giving for each basic type $\tau \in \mathsf{BasTyp}$ a domain Dom_τ of values of that type. These extend to functional types inductively: for a type $(\sigma \to \tau) \in \mathsf{Typ}$, $\mathsf{Dom}_{(\sigma\to\tau)} = \{f \mid f : \mathsf{Dom}_\sigma \to \mathsf{Dom}_\tau\}$, that is, the set of all functions from elements of Dom_σ to elements of Dom_τ. Let $\mathsf{Dom} = \bigcup_{\tau\in\mathsf{Typ}} \mathsf{Dom}_\tau$. We also need to given an interpretation for all the constants, via a function $[\![-]\!]_\tau : \mathsf{Con}_\tau \to \mathsf{Dom}_\tau$ assigning to every constant of type τ an object of type τ. (We simply write $[\![-]\!]$ when the type is clear from the context.) Hence, a model for higher-order logic is of the form $M = (\mathsf{Dom}, [\![-]\!])$. We extend the interpretation $[\![-]\!]$ to all the terms of the language. To deal with variables, we define an assignment to be a function $\theta : \mathsf{Var} \to \mathsf{Dom}$ such that $\theta(x) \in \mathsf{Dom}_\tau$ if $x \in \mathsf{Var}_\tau$. We denote $\theta[x := a]$ the assignment that maps x to a and $y \neq x$ to $\theta(y)$. We define the denotation $[\![\alpha]\!]^\theta_M$ of the term α with respect to the model $M = \langle \mathsf{Dom}, [\![-]\!]\rangle$ and assignment θ as:

$[\![x]\!]^\theta_M = \theta(x)$ if $x \in \mathsf{Var}$,
$[\![c]\!]^\theta_M = [\![c]\!]$ if $c \in \mathsf{Con}$,
$[\![\alpha(\beta)]\!]^\theta_M = [\![\alpha]\!]^\theta_M([\![\beta]\!]^\theta_M)$, and
$[\![\lambda x.\alpha]\!]^\theta_M = f$ such that $f(a) = [\![\alpha]\!]^{\theta[x:=a]}_M$.

Standard frames are subject to restrictions. For instance, the domain corresponding to boolean values must be a two-element domain, such as $\mathsf{Dom}_{Bool} = \{\mathbf{true}, \mathbf{false}\}$. Moreover, they must give a fixed interpretation to the logical constants (i.e., the conjunction operator should actually behave as a conjunction operator). Hence, we require:

$$[\![\mathbf{not}]\!](b) = \begin{cases} \mathbf{false} & \text{if } b = \mathbf{true} \\ \mathbf{true} & \text{if } b = \mathbf{false} \end{cases}$$
$$[\![\mathbf{and}]\!](b_1)(b_2) = \begin{cases} \mathbf{true} & \text{if } b_1 = \mathbf{true} \text{ and } b_2 = \mathbf{true} \\ \mathbf{false} & \text{otherwise} \end{cases}$$
$$[\![\mathbf{eq}_\tau]\!](v_1)(v_2) = \begin{cases} \mathbf{true} & \text{if } v_1 = v_2 \\ \mathbf{false} & \text{otherwise} \end{cases}$$
$$[\![\mathbf{every}_\tau]\!](f) = \begin{cases} \mathbf{true} & \text{if } f(x) = \mathbf{true} \text{ for all } x \in \mathsf{Dom}_\tau \\ \mathbf{false} & \text{otherwise} \end{cases}$$

One can check that if we define $\mathbf{some}_\tau$ as $\lambda P.\mathbf{not}(\mathbf{every}_\tau(\lambda x.\mathbf{not}(P(x))))$, it has the expected interpretation. Note that we will often use the abbreviations $\varphi \wedge \psi$ for $\mathbf{and}(\varphi,\psi)$, and $\neg\varphi$ for $\mathbf{not}(\varphi)$.

A formula of higher-order logic is a term of type $Bool$. We say that a model M satisfies a formula φ if $[\![\varphi]\!]_M = \mathbf{true}$ in the model. Two terms are said to be logically equivalent if they have the same interpretation in all models. One can check, for instance, that $\lambda x.r(x)$ and r are logically equivalent, as are $(\lambda x.\alpha)\beta$ and $\alpha\{\beta/x\}$ (that is, α where every occurrence of x is replaced by β).

For example, consider the following simple three individual model M, with constants $\mathbf{tarzan}, \mathbf{jane}, \mathbf{kala}$ and $\mathbf{like}$.

$$\mathsf{Dom}_{Ind} = \{t, j, k\}$$

$$[\![\mathbf{tarzan}]\!] = t \quad [\![\mathbf{jane}]\!] = j \quad [\![\mathbf{kala}]\!] = k$$

$$[\![\mathbf{like}]\!] = \left[\begin{array}{ccc} t & \mapsto & \left[\begin{array}{ccc} t & \mapsto & \textbf{false} \\ j & \mapsto & \textbf{true} \\ k & \mapsto & \textbf{true} \end{array}\right] \\ j & \mapsto & \left[\begin{array}{ccc} t & \mapsto & \textbf{true} \\ j & \mapsto & \textbf{false} \\ k & \mapsto & \textbf{false} \end{array}\right] \\ k & \mapsto & \left[\begin{array}{ccc} t & \mapsto & \textbf{true} \\ j & \mapsto & \textbf{false} \\ k & \mapsto & \textbf{true} \end{array}\right] \end{array}\right]$$

This model M satisfies the term $\mathbf{like}(\mathbf{kala})(\mathbf{tarzan})$ (*Kala likes Tarzan*) as $[\![\mathbf{like}(\mathbf{kala})(\mathbf{tarzan})]\!] = [\![\mathbf{like}]\!](k)(t) = \textbf{true}$. It also satisfies the term $\mathbf{some}_{Ind}(\mathbf{like}(\mathbf{jane}))$ (*There is someone Jane likes*). It does not satisfy the term $\mathbf{every}_{Ind}(\lambda x.\mathbf{like}(x)(x))$ (*Everyone likes himself/herself*).

We will use higher-order logic to express our semantics. The idea is to associate with every sentence (or part of speech) a higher-order logic term. We can then use the semantics of higher-order logic to derive the truth value of the sentence. Consider the examples at the beginning of the section. We assume constants $\mathbf{tarzan}$, $\mathbf{kala}$ and $\mathbf{jane}$ of type *Ind*, and a constant $\mathbf{like}$ of type $Ind \to Ind \to Bool$. We can translate the sentence *Tarzan likes Jane* as $\mathbf{like}(\mathbf{tarzan})(\mathbf{jane})$, as in first-order logic. But now we can also translate the part of speech *like Jane* independently as $\lambda x.\mathbf{like}(x)(\mathbf{jane})$.

For a more interesting example, consider the treatment of noun phrases as given at the beginning of the section. The solution to the problem of losing the grammatical structure was solved by Russell by treating all noun phrases as though they were functions over their verb phrases. This is analogous to what is already happening with the definition of $\mathbf{every}_{Ind}$ in higher-order logic, which has type $(Ind \to Bool) \to Bool$. Such generalized quantifier takes a property of an individual (a property has type $Ind \to Bool$) and produces a truth value—in the case of $\mathbf{every}$, the truth value is true if every individual has the supplied property. A similar abstraction can be applied to a noun position. We define a generalized determiner as a function taking a property stating a restriction on the quantified individuals, and returning a generalized quantifier obeying that restriction. Hence, a generalized determiner has type $(Ind \to Bool) \to (Ind \to Bool) \to Bool$. Consider the following generalized determiners, used above:

$$\begin{array}{l} \mathbf{some}^2 = \lambda P.\lambda Q.\mathbf{some}(\lambda x.P(x) \wedge Q(x)) \\ \mathbf{every}^2 = \lambda P.\lambda Q.\mathbf{every}(\lambda x.P(x) \Rightarrow Q(x)) \\ \mathbf{no}^2 = \lambda P.\lambda Q.\neg\mathbf{some}(\lambda x.P(x) \wedge Q(x)) \end{array}$$

One can check that the sentence *An apeman likes Jane* becomes $\mathbf{some}^2(\mathbf{apeman})(\lambda x.\mathbf{like}(x)(\mathbf{jane}))$, that *Every apeman likes Jane* becomes $\mathbf{every}^2(\mathbf{apeman})(\lambda x.\mathbf{like}(x)(\mathbf{jane}))$, and that *No apeman likes Jane* becomes $\mathbf{no}^2(\mathbf{apeman})(\lambda x.\mathbf{like}(x)(\mathbf{jane}))$. The subject is interpreted as $\mathbf{some}^2(\mathbf{apeman})$, $\mathbf{every}^2(\mathbf{apeman})$ and $\mathbf{no}^2(\mathbf{apeman})$ respectively. The verb phrase *likes Jane* is given the expected semantics $\lambda x.\mathbf{like}(x)(\mathbf{jane})$. What about the original sentence *Tarzan likes Jane*. According to the above, we should be able to give a semantics to *Tarzan* (when used as a subject) with a type $(Ind \to Bool) \to Bool$. One can check that if we interpret *Tarzan* as $\lambda P.P(\mathbf{tarzan})$, we indeed get the required behavior. Hence, we see that the noun phrase can be given the uniform type $(Ind \to Bool) \to Bool$, and that higher-order logic can be used to derive a uniform, compositional semantics.

6 ... from syntax

We have seen in the previous section how we can associate to sentences a semantics in higher-order logic. More importantly, we have seen how we can assign a semantics to sentence extracts, in a way that does capture the intuitive meaning of the sentences. The question at this point is how to derive the higher-order logic term corresponding to a given sentence or sentence extract.

The grammatical theory we use to achieve this is *categorial grammars*, originally developed by Ajdukiewicz [1] and later Bar-Hillel [4]. In fact, we will use a generalization of their approach due to Lambek [8]. The idea behind categorial grammars is simple. We start with a set of *categories*, each category representing a grammatical function. For instance, we can start with the simple categories *np* representing noun phrases, *n* representing nouns, and *s* representing sentences. Given categories A and B, we can form the *functor* categories A/B and $B\backslash A$. The category A/B represents the category of syntactic units that take a syntactic unit of category B to their right to form a syntactic unit of category A. Similarly, the category $B\backslash A$ represents the category of syntactic units that take a syntactic unit of category B to their left to form a syntactic unit of category A. Consider some examples. The category n/n represents the category of prenominal modifiers, such as adjectives: they take a noun on their right and form a noun. The category $n\backslash n$ represents the category of postnominal modifiers. The category $np\backslash s$ is the category of intransitive verbs: they take a noun phrase on their left to form a sentence. Similarly, the category $(np\backslash s)/np$ represents the category of transitive verbs: they take a noun phrase on their right to then expect a noun phrase on their left to form a sentence.

Before deriving semantics, let's first discuss well-formedness, as this was the original goal for such grammars. The idea was to associate to every word (or complex sequence of words that constitute a single lexical entry) one or more categories. We will call this the dictionary, or lexicon. The approach described by Lambek [8] is to prescribe a calculus of categories so that if a sequence of words can be assigned a category A according to the rules, then the sequence of words is deemed a well-formed syntactic unit of category A. Hence, a sequence of words is a well-formed sentence if it can be shown in the calculus that it has category s. As an example of reduction, we see that if σ_1 has category A and σ_2 has category $A\backslash B$, then $\sigma_1\ \sigma_2$ has category B. Schematically, $A, A\backslash B \Rightarrow B$. Moreover, this goes both ways, that is, if $\sigma_1\ \sigma_2$ has category B and σ_1 can be shown to have category A, then we can derive that σ_2 has category $A\backslash B$.

It was the realization of van Benthem [12] that this calculus could be used to assign a semantics to terms and use the derivation of categories to derive the semantics. The semantic will be given in some higher-order logic as we saw above. We assume that to every basic category corresponds a higher-order logic type. Such a type assignment T can be extended to functor categories by putting $T(A/B) = T(B\backslash A) = T(B) \to T(A)$. We extend the dictionary so that we associate with every word one or more categories, and a corresponding term of higher-order logic. We stipulate that the term α corresponding to a word in category A should have a type corresponding to the category, i.e. $\alpha \in \mathsf{Term}_{T(A)}$.

We will use the following notation (called a sequent) $\alpha_1 : A_1, \dots, \alpha_n : A_n \Rightarrow \alpha : A$ to mean that expressions $\alpha_1, \dots, \alpha_n$ of categories $A_1, \dots, A_n$ can be concatenated to form an expression α of category A. We will use capital Greek letters (Γ, Δ,...) to represent sequences of expressions and categories. We now give rules that allow us to derive new sequents from other sequents:

$$\frac{}{\alpha : A \Rightarrow \alpha : A} \qquad \frac{\Gamma_2 \Rightarrow \beta : B \quad \Gamma_1, \beta : B, \Gamma_3 \Rightarrow \alpha : A}{\Gamma_1, \Gamma_2, \Gamma_3 \Rightarrow \alpha : A}$$

In other words, if Γ_2 can concatenate into an expression β with category B, and if $\Gamma_1, \beta : B, \Gamma_3$ can concatenate into an expression α with category A, then $\Gamma_1, \Gamma_2, \Gamma_3$ can concatenate into α with category A.

$$\frac{\Delta \Rightarrow \beta : B \quad \Gamma_1, \alpha(\beta) : A, \Gamma_2 \Rightarrow \gamma : C}{\Gamma_1, \alpha : A/B, \Delta, \Gamma_2 \Rightarrow \gamma : C} \qquad \frac{\Delta \Rightarrow \beta : B \quad \Gamma_1, \alpha(\beta) : A, \Gamma_2 \Rightarrow \gamma : C}{\Gamma_1, \Delta, \alpha : B\backslash A, \Gamma_2 \Rightarrow \gamma : C}$$

$$\frac{\Gamma, x : A \Rightarrow \alpha : B}{\Gamma \Rightarrow \lambda x.\alpha : B/A} \qquad \frac{x : A, \Gamma \Rightarrow \alpha : B}{\Gamma \Rightarrow \lambda x.\alpha : A\backslash B}$$

For example, the following is a derivation of *Tarzan likes Jane*.

$$\frac{\overline{\mathbf{tarzan} : np \Rightarrow \mathbf{tarzan} : np} \quad \dfrac{\overline{\mathbf{jane} : np \Rightarrow \mathbf{jane} : np} \quad \overline{\mathbf{like}(\mathbf{tarzan})(\mathbf{jane}) : s \Rightarrow \mathbf{like}(\mathbf{tarzan})(\mathbf{jane}) : s}}{\mathbf{like}(\mathbf{tarzan}) : s/np, \mathbf{jane} : np \Rightarrow \mathbf{like}(\mathbf{tarzan})(\mathbf{jane}) : s}}{\mathbf{tarzan} : np, \mathbf{like} : np\backslash s/np, \mathbf{jane} : np \Rightarrow \mathbf{like}(\mathbf{tarzan})(\mathbf{jane}) : s}$$

For example, the following derivation of the sentence fragment *Tarzan likes* shows that it is of the type s/np—it is an expression that expects a noun phrase to the right to form a complete sentence.

$$\frac{\dfrac{\overline{\mathbf{tarzan} : np \Rightarrow \mathbf{tarzan} : np} \quad \dfrac{\overline{x : np \Rightarrow x : np} \quad \overline{\mathbf{like}(\mathbf{tarzan})(x) : s \Rightarrow \mathbf{like}(\mathbf{tarzan})(x) : s}}{\mathbf{like}(\mathbf{tarzan}) : s/np, x : np \Rightarrow \mathbf{like}(\mathbf{tarzan})(x) : s}}{\mathbf{tarzan} : np, \mathbf{like} : np\backslash s/np, x : np \Rightarrow \mathbf{like}(\mathbf{tarzan})(x) : s}}{\mathbf{tarzan} : np, \mathbf{like} : np\backslash s/np \Rightarrow \lambda x.\mathbf{like}(\mathbf{tarzan})(x) : s/np}$$

A look at the theory underlying type-logical approaches to linguistics reveals some fairly deep mathematics at work. The fact that we can derive the semantics in parallel with a derivation of the categories associated with the sequence of words is not an accident. In fact, it is a phenomenon known as a Curry-Howard isomorphism. The original Curry-Howard isomorphism was a correspondence between intuitionistic propositional logic and the simply-typed λ-calculus: every valid formula of intuitionistic propositional logic corresponds to a type in the simply-typed λ-calculus, in such a way that a proof of the formula corresponds to a λ-term of the corresponding type. Such a correspondence exists between the Lambek calculus (which can be seen as a substructural logic, namely intuitionistic bilinear logic) and an appropriate instance of the λ-calculus, namely higher-order logic.

We have in this review merely sketched the basics of the type-logic approach, a merciless summary of the first few chapters of the book. Carpenter investigates more advanced linguistic phenomena by extending the Lambek calculus with more categorial constructions, and deriving the corresponding semantics. For instance, he deals with generalized quantifiers, deriving the semantics we hinted at earlier through a syntactic derivation, as well as plural forms, and modalities such as belief. The latter requires a move to a modal form of higher-order logic known as intensional logic.

7 The book

The book naturally divides in three parts. The first part, the first five chapters (as well as an appendix on mathematical preliminaries), introduces the technical machinery required to deal with linguistic issues, namely the higher-order logic used to express the semantics, and the Lambek calculus to derive the semantics.

Chapter 1, **Introduction**, provides an outline of the role of semantics in linguistic theory. Carpenter discusses the central notions of truth and reference, the latter telling us how linguistic expressions can be linked to objects in the world. He gives a survey of topics that linguistic theories need to address, including synonymy, contradiction, presupposition, ambiguity, vagueness. He also surveys topics in pragmatics, the branch of linguistic concerned with aspects of meaning that involve more than literal interpretation of utterances. Finally, he argues for the methodology of the book, in terms of originality, compositionality, model theory and grammar fragments. Some caveats apply: he studies models of natural language itself, not models of our knowledge or ability to use language; furthermore, these models are not intended to have any metaphysical interpretation, but are only a description and approximation of natural language.

Chapter 2, **Simply Typed λ-Calculus**, lays out the basic theory of the simply typed λ-calculus. The simply typed λ-calculus provides an elegant solution to the problem of giving a denotation for the basic expressions of a language in a compositional manner, as explained in Chapter 3. This chapter concentrates on the basic theory, describing the language of the simply typed λ-calculus, along with a model theory and a proof theory for the logical language, that formalizes whether two λ-calculus expressions are equal (have the same denotation in all models). The standard λ-calculus notions of reductions, normal forms, strong normalization, the Church-Rosser theorem, and combinators are discussed. An extension of the simply typed λ-calculus with sums and products is described.

Chapter 3, **Higher-Order Logic**, introduces a generalization of first-order logic where quantification and abstraction occurs over all the entities of the language, including relations and functions. Higher-order logic is defined as a specific instance of the simply typed λ-calculus, with types capturing both individuals and truth values, and logical constants such as conjunction, negation, and universal quantification. The usefulness of the resulting logic is demonstrated by showing how it can handle quantifiers in natural languages in a uniform way. The proof theory of higher-order logic is discussed.

Chapter 4, **Applicative Categorial Grammar**, is an introduction to the syntactic theory from which the denotation of natural language terms is derived, that of categorial grammars. Categorial grammars are based on the notion of categories representing syntactic functionality, and describe how to syntactically combine entities in different categories to form combined entities in new categories. The framework described in this chapter is the simplest form of applicative categorial grammar, which will be extended in later chapters. After introducing the basic categories, the chapter shows how to assign semantic domains to categories, and how to associate with every basic syntactic entity a term in the corresponding domain, creating a lexicon. The basics of how to derive the semantic meaning of a composition of basic syntactic entities based on the derivation of categories is explored. Finally, a discussion of some of the consequences of this way of assigning semantic meaning is given; mainly, it focuses on ambiguity and vagueness, corresponding respectively to expressions with multiple meanings, and expressions with a single undetermined meaning.

Chapter 5, **The Lambek Calculus**, introduces a logical system that extends the applicative categorial grammar framework of the previous chapter. The Lambek calculus allows for a more flexible description of the possible ways of putting together entities in different categories. The Lambek calculus is presented both in sequent form and in natural deduction form, the former appropriate for automatic derivations, the latter more palatable for humans. The Lambek calculus is decidable (i.e., the problem of determining whether the calculus can show a given sentence grammatical is decidable). The correspondence between the Lambek calculus and a variant of linear logic is established.

The following four chapters show how to apply the machinery of the first part to different aspects of linguistic analysis.

Chapter 6, **Coordination and Unbounded Dependencies**, studies two well-known linguistic applications of categorial grammars. The first, coordination, corresponds to the use of *and* in sentences. Such a coordination operator can occur on many levels, coordinating two nouns (*Joe and Victoria*), two adjectives (*black and blue*), two sentences, etc. Coordination at any level is achieved by lifting the coordination to the level of sentences, via the introduction of a polymorphic coordination operator in the semantic framework. This operator can be handled in the Lambek calculus via type lifting. The resulting system remains decidable. An extension of the Lambek calculus with conjunction and disjunction is considered, to account for coordinating, for example, unlike complements of a category, such as in *Jack is a good cook and always improving*. The second well-known use of categorial grammars is to account for unbounded dependencies, that is, relationships between distant expressions within an expression, the distance potentially unbounded.

This is handled by introducing a new categorial combinator $A \uparrow B$, an element of which can be analyzed as an A with a B missing somewhere within it. The appropriate derivation rules can be added to the Lambek calculus.

Chapter 7, **Quantifiers and Scope**, studies the contribution of quantified noun phrases to the meaning of phrases in which they occur. Such generalized quantifiers, such as *every kid*, or *some toy*, are traditionally problematic because they take semantic scope around an arbitrary amount of material. For instance, *every kid played with some toy* has two readings, depending on the scope of the quantifiers *every* and *some* (is there a single toy with which every kid plays, or does every kid play with a possibly different toy?) Accounting for such readings is the aim of this chapter. Two historically significant approaches to quantifiers are surveyed: Montague's quantifying in approach, and Cooper's storage mechanism. Then, the type-logical solution of Moortgat is described. The idea is to introduce a new category $B \Uparrow A$ of expressions that act locally as B's but take their semantic scope over an embedding expression of category A. Generalized quantifiers are given category $np \Uparrow s$, since they act like a noun phrase (category np) *in situ*, but scope semantically to an embedding sentence (category s). The Lambek calculus is extended with appropriate derivation rules. The issues of quantifier coordination, quantifiers within quantifiers, and the interaction with negation are discussed. Other topics related to quantifiers and determiners in general, such as definite descriptions, possessives (*every kid's toy*), indefinites (*some student*), generics (*italians*), comparatives (*as tall as*), and expletives (*it*, *there*) are analyzed within that context.

Chapter 8, **Plurals**, provides a type-logical account of plurality. First, the notion of group is added to the syntax and semantics. The type $Group$ is considered to be a subtype of the type Ind and thus the domain of $Group$ is a subset of the domain of Ind. A relation linking a group to the property that defines membership in the group is defined, and restrictions are imposed to ensure that every group has a unique property that defines membership of that group. With this interpretation, categories for plural noun phrases and plural nouns are studied. The notions of distributors (to view a group as a set of individuals) and collectors (to view a set of individuals as a group) are defined, to handle, for example, verbs that apply only to individuals or only to groups. The issues of coordination and negation are examined in the context of plurals. Further topics examined include plural quantificatives and, more generally, partitives (*each*, *all*, *most*, or numerical partitives such as *three of*, etc.), nonboolean coordination with *and*, comitative (the use of *with* in *Tarzan climbed the tree with Cheetah*), and mass terms such as *snow* and *water*.

Chapter 9, **Pronouns and Dependency**, analyses the use of non-indexical pronouns such as *him*, *she*, *itself*, especially the dependent use of such pronouns. Dependent pronouns are characterized as having their interpretation depend on the interpretation of some other expression (the antecedent). For example, *he* in *Jody believes he will be famous*. A popular interpretation of pronouns in type-logical frameworks is as variables, although the treatment is subtle, at least for non-reflexive pronouns such as the *he* above. (Admittedly, this topic is an outstanding problem for type-logical grammars.) Reflexive pronouns, such as *himself* in *Everyone likes himself*, can be handled as quantifiers. Topics related to pronomial forms are examined, such as reciprocals (the *each other* in *The three kids like each other*), pied piping (the *which* in *the table the leg of which Jody broke*), ambiguous verb-phrases ellipses (*Jody likes himself and Brett does too*), and interrogatives.

The final part, the last three chapters, extend the framework with modalities to account for intensional aspects of natural languages.

Chapter 10, **Modal Logic**, introduces the logical tools required to deal with intensionality, tense and aspect. The key concept is that of a modal logic, where operators are used to qualify the truth of a statement. The chapter presents both a model theory (Kripke frames) and a proof theory for S5, a particular modal logic of necessity. A brief discussion of how the techniques of modal logic can be used to model indexicality precedes the presentation of a general modal model. First-order tense logics, which extend first order logics with modal operators about the truth of statements in the past and future, are presented in some depth, as they are able to provide a model of tenses in natural language. Time can be regarded as a collection of moments, or as a collection of possibly overlapping intervals. Higher order logic is extended to include modal operators by taking the domains of worlds and time to be basic types, on the same level as the domains of individuals and truth values, yielding a framework referred to as intensional logic. This approach avoids a number of problems associated with simply abstracting the model for higher order logic over possible worlds.

Chapter 11, **Intensionality**, uses modal logic to extend the type-logical framework to cover intensional constructions. In particular, $World$ is added as a new basic type, and the assignment of types to basic categories is modified, replacing $Bool$ with $World \to Bool$, i.e. truth values may be different at different worlds. This change facilitates the inclusion of many constructs, such as propositional attitudes (*Frank believes Brooke cheated*), modal adverbs (*possibly*), modal auxiliaries (*should*, *might*), and so-called control verbs (*persuaded*, *promised*), although some constructs remain problematic. The “individual concepts” approach is considered, where the type of a noun phrase is $World \to Ind$ instead of Ind, i.e. the referent of a noun phrase may differ from world to world. Other approaches to intensionality, which do not involve possible worlds, are explained briefly. Finally, the last section returns to the issue of giving a categorization of control verbs, and gives some problematic examples showing the need for more work in this area.

Chapter 12, **Tense and Aspect**, extends the grammar and semantics with a theory of tense. It presents Reichenbach’s approach to simple and perfect tenses, how this applies to discourse, and Vendler’s verb classes—a semantic classification of verbs that is correlated with their syntactic use. The approach Carpenter adopts for tense and aspect is based on insights derived from these works, and on further development of these works by other authors. To extend the grammar, verbs are subcategorized by classifying them based on whether they are finite or non-finite, and whether they involve simple or perfect tense, resulting in several different categories for sentences. A new basic type is introduced, representing time periods, and all of the sentence categories are assigned the same type: functions from time periods to truth values. The temporal argument always corresponds to the time of the event being reported. (This is essentially similar to the intensional approach of the previous chapter, but here we distinguish time periods from possible worlds.) From this beginning, the grammar is developed to encompass many of the English constructs involving tense and aspect. Many of these constructs are very complex in their usage and generally there do not seem to be simple and complete solutions to incorporating them into the grammar.

8 Opinion

There are some typos (potentially confusing, as they sometimes occur in the types for functions), as well as some glossing over central topics (such as the discussion of groups in Chapter 8). Carpenter doesn’t generally delve into syntactic explanations, that is, explaining why the theory of syntax he develops does or does not permit certain sentences. Moreover, for linguists, it may be important to note that Carpenter does not develop a theory of morphology (the structure of words at the level of morphemes).

This book fills a sorely void niche in the field of semantics of natural languages via type-logical approaches. There are some books on the subject, but the most accessible are severely limited in their development [13], while the others are typically highly mathematical and focus on the metatheory of the type-logical approach [10].

Carpenter’s book is a reasonable blend of mathematical theory and linguistic applications. Its great strength is an excellent survey of type-logical approaches applied to a great variety of linguistic phenomena. On the other hand, the preliminary chapters presenting the underlying mathematical theory are slightly confusing—not necessarily surprising considering the amount of formalism needed to account for all the linguistic phenomena studied. A background or at least exposure to ideas from both logic and programming language semantics is extremely helpful. In this sense, this book seems slightly more suited, at least as an introductory book, to mathematicians and computer scientists interested in linguistic applications, than to linguists interested in learning about applicability of type-logical approaches. (Although this book could nicely follow a book such as [13], or any other introductory text on type-logical grammars that focuses more on the "big picture" than on the underlying mathematical formalisms.) People that are not linguists will most likely find chapters 9 and on hard to follow, as they assume more and more knowledge of linguistic phenomena.

This book points to interesting areas of ongoing research. In particular, the later sections of the book on aspects of intensionality highlight areas where the semantics of natural languages are not clear. (This is hardly a surprise, as intensional concepts have always been problematic, leading philosophers to develop many flavors of modal logics to attempt to explain such concepts.) Another avenue of research that is worth pointing out, although not discussed in this book, is the current attempt to base semantics of natural languages not on higher-order logic as presented in this book, but rather on Martin-Löf constructive type theory, via categorial techniques [11].

References

[1] K. Ajdukiewicz. Die syntaktische Konnexität. *Studia Philosophica*, 1:1–27, 1935.

[2] J. Allwood, L.-G. Andersson, and O. Dahl. *Logic in Linguistics*. Cambridge Textbooks in Linguistics. Cambridge University Press, 1977.

[3] P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth through Proof*. Academic Press, 1986.

[4] Y. Bar-Hillel. A quasi-arithmetical notation for syntactic description. *Language*, 29:47–58, 1953.

[5] H. P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. Studies in Logic. North-Holland, Amsterdam, 1981.

[6] N. Chomsky. *Syntactic Structures*. Mouton and Co., 1957.

[7] H. B. Curry. Some logical aspects of grammatical structure. In *American Mathematical Society Proceedings of the Symposia on Applied Mathematics 12*, pages 56–68, 1961.

[8] J. Lambek. The mathematics of sentence structure. *The American Mathematical Monthly*, 65:154–170, 1958.

[9] R. Montague. *Formal Philosophy: Selected Papers of Richard Montague*. Yale University Press, 1974.

[10] G. V. Morrill. *Type Logical Grammar: Categorial Logic of Signs*. Kluwer Academic Publishers, 1994.

[11] A. Ranta. *Type-Theoretical Grammar*. Oxford University Press, 1994.

[12] J. van Benthem. The semantics of variety in categorial grammar. In W. Buszkowski, J. van Benthem, and W. Marciszewski, editors, *Categorial Grammar*, number 25 in Linguistics and Literary Studies in Eastern Europe, pages 37–55. John Benjamins, 1986. Previously appeared as Report 83-29, Department of Mathematics, Simon Fraser University (1983).

[13] M. M. Wood. *Categorial Grammars*. Routledge, 1993.

Review of
The π-calculus: A Theory of Mobile Processes
Author of Book: D. Sangiorgi and D. Walker
Author of Review: Riccardo Pucella, Dept of CS, Cornell
Cambridge University Press, 585 Pages

9 Introduction

With the rise of computer networks in the past decades, the spread of distributed applications with components across multiple machines, and with new notions such as mobile code, there has been a need for formal methods to model and reason about concurrency and mobility. The study of sequential computations has been based on notions such as Turing machines, recursive functions, the λ-calculus, all equivalent formalisms capturing the essence of sequential computations. Unfortunately, for concurrent programs, theories for sequential computation are not enough. Many programs are not simply programs that compute a result and return it to the user, but rather interact with other programs, and even move from machine to machine.

Process calculi are an attempt at getting a formal foundation based on such ideas. They emerged from the work of Hoare [4] and Milner [6] on models of concurrency. These calculi are meant to model systems made up of processes communicating by exchanging values across channels. They allow for the dynamic creation and removal of processes, allowing the modelling of dynamic systems. A typical process calculus in that vein is CCS [6, 7]. The π-calculus extends CCS with the ability to create and remove communication links between processes, a new form of dynamic behaviour. By allowing links to be created and deleted, it is possible to model a form of *mobility*, by identifying the position of a process by its communication links.

This book, "The π-calculus: A Theory of Mobile Processes", by Davide Sangiorgi and David Walker, is a in-depth study of the properties of the π-calculus and its variants. In a sense, it is the logical followup to the recent introduction to concurrency and the π-calculus by Milner [8], reviewed in SIGACT News, 31(4), December 2000.

What follows is a whirlwind introduction to CCS and the π-calculus. It is meant as a way to introduce the notions discussed in much more depth by the book under review. Let us start with the basics. CCS provides a syntax for writing processes. The syntax is minimalist, in the grand tradition of foundational calculi such as the λ-calculus. Processes perform actions, which can be of three forms: the sending of a message over channel x (written $\overline{x}$), the receiving of a message over channel x (written x), and internal actions (written τ), the details of which are unobservable. Send and receive actions are called *synchronization* actions, since communication occurs when the corresponding processes synchronize. Let α stand for actions, including the internal action τ, while we reserve $\lambda, \mu, \ldots$ for synchronization actions.[5] Processes are written using the following syntax:

$$P \quad ::= \quad A\langle x_1, \ldots, x_k\rangle \mid \sum_{i \in I} \alpha_i.P_i \mid P_1|P_2 \mid \nu x.P$$

We write 0 for the empty summation (when $I = \emptyset$). The idea behind process expressions is simple. The process 0 represents the process that does nothing and simply terminates. A process of the form $\lambda.P$ awaits to synchronize with a process of the form $\overline{\lambda}.Q$, after which the processes continue as process P and Q respectively. A generalization of such processes is $\sum_{i \in I} \alpha_i.P_i$, which nondeterministically synchronizes via one of its α_i only. We will write $\sum_{i \in I} \alpha_i.P_i$ as $\alpha_1.P_1 + \cdots + \alpha_n.P_n$ when the set I is $\{1, \ldots, n\}$.

[5]In the literature, the actions of CCS are often given a much more abstract interpretation, as simply names and co-names. The send/receive interpretation is useful when one moves to the π-calculus.

To combine processes, the parallel composition $P_1|P_2$ is used. Note the difference between summation and parallel composition: a summation offers a choice, so only one of the summands can synchronize and proceed, while a parallel composition allows all its component processes to proceed. (This will be made clear when we get to the transition rules describing how processes execute.) The process expression $\nu x.P$ defines a local channel name x to be used within process P. This name is guaranteed to be unique to P (possibly through consistent renaming). Finally, we allow process definitions, where we assume that every identifier A is associated with a process definition of the form $A(x_1, \ldots, x_k) = P_A$ where $x_1, \ldots, x_k$ are free in P_A. To instantiate the process A to values $y_1, \ldots, y_k$, you write $A\langle y_1, \ldots, y_k\rangle$.

As an example, consider the process $(x.y.0 + \overline{x}.z.0)|\overline{x}.0|\overline{y}.0$. Intuitively, it consists of three processes running in parallel: the first offers of choice of either receiving over channel x, or sending over channel x, the second sends over channel x, and the third sends over channel y. Depending on which choice the first process performs (as we will see, this depends on the actions the other process can perform), it can continue in one of two ways: if it chooses to receive on channel x (i.e., the $x.y.0$ summand is chosen), it can then receive on channel y, while if it chooses to send on x (i.e., the $\overline{x}.z.0$ summand is chosen), it can then receive on channel z.

To represent the execution of a process expression, we define the notion of a transition. Intuitively, the transition relation tells us how to perform one step of execution of the process. Note that since there can be many ways in which a process executes, the transition is fundamentally nondeterministic. The transition of a process P into a process Q by performing an action α is indicated $P \xrightarrow{\alpha} Q$. The action α is the observation of the transition. (We will sometimes simply use $\longrightarrow$ when the observation is unimportant.) The transition relation is defined by the following inference rules:

$$\frac{}{\sum_{i\in I} \alpha_i.P_i \xrightarrow{\alpha_j} P_j} \text{ for } j \in I \qquad \frac{P \xrightarrow{\lambda} P' \quad Q \xrightarrow{\overline{\lambda}} Q'}{P|Q \xrightarrow{\tau} P'|Q'}$$

$$\frac{P \xrightarrow{\alpha} P'}{\nu x.P \xrightarrow{\alpha} \nu x.P'} \text{ if } \alpha \notin \{x, \overline{x}\} \qquad \frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q}$$

$$\frac{\{\vec{y}/\vec{x}\}P_A \xrightarrow{\alpha} P'}{A\langle\vec{y}\rangle \xrightarrow{\alpha} P'} \text{ if } A(\vec{x}) = P_A \qquad \frac{Q \xrightarrow{\alpha} Q'}{P|Q \xrightarrow{\alpha} P|Q'}$$

For example, consider the transitions of the example process above, $(x.y.0 + \overline{x}.z.0)|\overline{x}.0|\overline{y}.0$. A possible first transition (the first step of the execution, if you wish), can be derived as follows:

$$\frac{\dfrac{x.y.0 + \overline{x}.z.0 \xrightarrow{x} y.0 \quad \overline{x}.0 \xrightarrow{\overline{x}} 0}{(x.y.0 + \overline{x}.z.0)|\overline{x}.0 \xrightarrow{\tau} y.0|0}}{(x.y.0 + \overline{x}.z.0)|\overline{x}.0|\overline{y}.0 \xrightarrow{\tau} y.0|0|\overline{y}.0}$$

That is, the process reduces to $y.0|0|\overline{y}.0$ in one step that does not provide outside information, since it appears as an internal action. Note that the 0 can be removed from the resulting process, as it does not contribute further to the execution of the process. The resulting process $y.0|\overline{y}.0$ can then perform a further transition, derived as follows:

$$\frac{y.0 \xrightarrow{y} 0 \quad \overline{y}.0 \xrightarrow{\overline{y}} 0}{y.0|\overline{y}.0 \xrightarrow{\tau} 0|0}$$

In summary, a possible sequence of transitions for the original process is the two-step sequence

$$(x.y.0 + \overline{x}.z.0)|\overline{x}.0|\overline{y}.0 \xrightarrow{\tau} y.0|\overline{y}.0 \xrightarrow{\tau} 0.$$

A central concept in the study of processes is that of equivalence of processes. We have in fact implicitly used a notion of equivalence in the example above, when we removed processes of the form 0 from parallel processes. Many notions of equivalence can be defined, capturing the various intuitions that lead us to think of two processes as equivalent. A standard notion of equivalence is strong bisimulation. A strong bisimulation is a relation $\mathcal{R}$ such that whenever $P\mathcal{R}Q$, if $P \xrightarrow{\alpha} P'$, then there exists Q' such that $Q \xrightarrow{\alpha} Q'$ and $P'\mathcal{R}Q'$, and if $Q \xrightarrow{\alpha} Q'$, then there exists P' such that $P \xrightarrow{\alpha} P'$ and $P'\mathcal{R}Q'$. We say P and Q are strongly bisimilar if there exists a strong bisimulation $\mathcal{R}$ such that $P\mathcal{R}Q$. In other words, if P and Q are strongly bisimilar, then whatever transition P can take, Q can match it with one of its own that retains all of P's options, and vice versa.

Strong bisimulation is a very fine equivalence relation—not many processes end up being equivalent. More worryingly, strong bisimulation does not handle internal actions very well. Intuitively, process equivalence should really only involve observable actions. Two processes that only perform internal actions should be considered equivalent. For instance, the processes $\tau.\tau.0$ and $\tau.0$ should really be considered equivalent, as they really do nothing after performing some internal (and hence really unobservable) actions. Unfortunately, it is easy to check that these two processes are not strongly bisimilar. To capture this intuition, we define a weaker notion of equivalence, aptly called weak bisimulation.

Let $P \Longrightarrow Q$ denote that P can take any number of transitions before turning into Q. In other words, $P \Longrightarrow Q$ holds if $P \longrightarrow \cdots \longrightarrow Q$, i.e., $\Longrightarrow$ is the reflexive transitive closure of $\longrightarrow$. We write $P \overset{\alpha}{\Longrightarrow} Q$ if $P \Longrightarrow P' \xrightarrow{\alpha} Q' \Longrightarrow Q$, i.e., if P can take any number of transitions before and after doing an α-transition. A weak bisimulation is a relation $\mathcal{R}$ such that whenever $P\mathcal{R}Q$, if $P \xrightarrow{\tau} P'$ then there exists Q' such that $Q \Longrightarrow Q'$, while if $P \xrightarrow{\lambda} P'$ then there exists Q' such that $Q \overset{\lambda}{\Longrightarrow} Q'$, and vice versa for Q. We say that P and Q are weakly bisimilar if there exists a weak bisimulation $\mathcal{R}$ such that $P\mathcal{R}Q$. If P and Q are weakly bisimilar, an internal transition by P can be matched by zero or more transitions by Q, while an α-transition by P can be matched by one or more transition by Q as long as one such is an α-transition, and vice versa. One can check that a strong bisimulation is a weak bisimulation, but the converse does not hold: weak bisimilarity is a coarser equivalence relation.

In the framework above, we cannot communicate any value at synchronization time. It is not difficult to extend the calculus to allow for the exchange of values such as integers over the channel during a synchronization. Doing so does not fundamentally change the character of the calculus. A variation on this, however, does change the calculus in a highly nontrivial way, and that is to allow for the communication of channel names during synchronization. This yields the π-calculus. To see why such an extension might be useful, consider the following scenario. It shows that passing channel names around can be used to model process mobility. Intuitively, a process is characterized by the channels it exposes to the world, that can be used to communicate with it. These channels act as an interface to the process. Hence, the process $P = (x.y.0|\overline{x}.z.0)$ provides the channel x as an interface. A process that sends x to another process in some sense sends the capability to access P to that process. This captures the mobility of process P, although more accurately it captures the mobility of the capability to access P. It was Sangiorgi's original contribution to the theory of the π-calculus to show that capability mobility could indeed express in a precise sense process mobility [9]. (This particular topic is covered in Part V of the book.)

The necessary modifications to the calculus that capture the above intuition are straightforward. Syntactically, we need to change the kind of guards that can appear in summands. Sends now must carry a value, and receives must accept an identifier to be bound to the received value. The only values that can be exchanged are channel names. We define a prefix π to be of the following form:

$$\pi \quad ::= \quad \overline{x}\langle y\rangle \mid x(y) \mid \tau$$

Here, x and y are channel names, and τ is the internal action. Processes are described by the following syntax, where the only difference from CCS is in the summations:

$$P \quad ::= \quad A\langle x_1,\ldots,x_k\rangle \mid \sum_{i\in I}\pi_i.P_i \mid P_1|P_2 \mid \nu x.P$$

The intuition behind the new prefixes should be rather clear: as before, a process of the form $\tau.P$ performs an internal action τ before becoming P; a process of the form $\overline{x}\langle y\rangle.P$ is ready to send the channel name y onto channel x, and when this process synchronizes it behaves as P; a process of the form $x(y).P$ is ready to receive a channel name from channel x, and when this process synchronizes, it binds the received channel to the identifier y in P before continuing as the (modified) P. Note that the calculus in Sangiorgi and Walker's book is slightly different than the one presented here, which has been kept simple for reasons of exposition.

As an example of a process in the π-calculus, consider the process $x(y).\overline{y}\langle z\rangle.0|\overline{x}\langle w\rangle$. Intuitively, the second process sends channel name w via channel x to the first process, who binds it to name y. The first process then sends channel name z over this channel. Hence, the above process "reduces" to the process $\overline{w}\langle z\rangle$. Although the intuition underlying passing channel names over channels should be clear, it turns out that formalizing this in a nice way is difficult. Already describing the transition relation (in the form we gave above) is complicated by the fact that we cannot simply consider channel names, but also need to take into account the information exchanged at synchronization time. Similarly, describing the appropriate notion of bisimulation in such a setting needs to account for the information exchanged. Rather than describing these, I will defer to Sangiorgi and Walker's book, since in a sense this is exactly where the book picks up.

10 The book

The book splits into seven parts. Each parts comprises between two and three chapters, and deals with a particular aspect of the π-calculus.

Part I, **The π-calculus**, is made up of two chapters. Chapter 1 introduces the basic concepts of the π-calculus, starting from the syntax, and defines two notion of system behaviour. The first notion, called reduction, can be understood as a simplified account of what we described above. Essentially, reduction tells you how a term rewrites upon execution, without keeping track of the actions performed by the process. This notion of behaviour has the advantage of being simple and intuitive. The second notion of behaviour, in terms of labelled transition, follows the account I gave in the introduction. The relationship between these two notions is made explicit.

Behavioural equivalence of processes occupies a central part in the theory, and an initial take on the problem is given in Chapter 2. Many notion of equivalence can be defined for the π-calculus, and the fundamental ones are studied in this chapter. The main form of equivalence, barbed congruence, is defined naturally by specifying that no difference between equivalent processes can be observed by placing them into an arbitrary π-calculus context. This natural notion of equivalence turns out to be awkward to work with, and definitions based on ideas similar to bisimilarity given above can be introduced. It turns out that strong bisimilarity defined in the most natural way is equivalent to barbed congruence for the π-calculus. (This equivalence however fails when extensions to the π-calculus are considered in Parts IV and V, where barbed congruence remains the natural notion of equivalence.)

Part II, **Variations of the π-calculus**, is made up of three chapters. Chapter 3 studies various simple modifications to the basic π-calculus framework given in Part I. A variant of the π-calculus, the polyadic π-calculus, is introduced, where tuples of names can be passed at synchronization time. Adding tuples in such a way forces the introduction of sorts, which intuitively ensure that synchronizing terms agree as to the number of names that are being exchanged. This can be viewed as a primitive form of typing for the π-calculus. The second variation considers the addition of recursive definitions to the π-calculus. These variations are minor in the sense that they do not add expressive power: anything that can be expressed using tuples or recursive definitions can already be expressed in the basic π-calculus of Part I.

Chapter 4 returns to behavioural equivalence. New notions of equivalence are defined, such as ground, late, and open bisimilarity. Roughly speaking, these new bisimilarity definitions ease the demand on the processes to mimic one another's input actions. The main advantage of these equivalences is that they are easier to establish than the more natural notions of equivalence, a recurring concern, especially in the context of automatic tools for reasoning about processes. After studying these new equivalence relations, the question of axiomatizing these equivalences is addressed. An axiomatization for an equivalence relation on processes is a set of equational axiom on process expressions that, together with the rules of equational reasoning (i.e., reflexivity, transitivity, etc.), suffice for proving exactly the valid equations between process expressions, with respect to the equivalence under consideration.

While the variants of the π-calculus examined in Chapter 3 are extensions that do not change the expressive power of the calculus, Chapter 5 studies restrictions to the calculus that lend insight into various phenomena of interaction and mobility. The asynchronous π-calculus restricts terms that perform a send action to be of the form $\overline{x}\langle y\rangle.0$, that is, to become the null process after synchronization. This can be used to capture a form of asynchrony, and the resulting calculus is provably less expressive than the full π-calculus. The localized π-calculus has essentially the restriction that a name received by a process cannot be used as an input channel—it must be either used for sending, or sent to another process. Hence, all input channels are localized in the process in which they are defined. Finally, the private π-calculus imposes the restriction on the π-calculus that local names cannot be exported, that is, they cannot be sent to a process outside of the scope of the ν where the channel is defined. For all these subcalculi, notions of equivalences are studied.

Part III, **Typed π-calculi**, explores the issue of assigning types to π-calculus expressions. This part is mostly concerned with defining type systems to detect errors statically, or to enforce properties statically. (The following part focuses on types as an aid for reasoning about the behaviour of processes.) In Chapter 6, the foundations of type systems for the π-calculus are laid. The Base-π calculus is introduced, essentially CCS extended with the capability of passing values at synchronization time. The types are associated with those values. Channels are also given a type, stating the type of value they carry. Processes themselves are also given a type, all processes getting the same type. The simply-typed π-calculus is obtained by adding channel names to the values that can be exchanged, and modifying the type system accordingly.

Chapter 7 extends the basic type system of the simply-typed π-calculus with the notion of subtyping. In order for this to be nontrivial, the calculus is refined to differentiate, in the type of channels, whether or not the channel is an input channel (used for receiving values) or an output channel (used for sending values). We can then refine the type system to account for subtyping: if we have an output channel that can send values of type S, then clearly we can also use the channel to send values of any subtype of S. Various properties of subtyping are examined.

The type systems described above are fairly standard. In Chapter 8, more advanced type systems are investigated. These are meant to capture various properties of processes that we may want to enforce. For instance, it is possible to deal with linearity constraints in the type system, for example, that a given channel name can only be used once for input or output. Another property is that of receptiveness, that is, that a given channel name is always ready to process some input, or equivalently that sending a value on that channel will never deadlock. Another property is polymorphism, namely the fact that some processes don't really care about the actual type of the value they process. An example of such a process is one that simply forwards a value from one channel to another. Type systems to capture these properties are defined and studied.

Part IV, **Reasoning about processes using types**, explores another advantage of type systems beyond static enforcement, that of helping reasoning about behavioural properties of processes. The three chapters in this part mirror those of Part III. Specifically, Chapter 9 discusses how types can help in reasoning. The example examined in the chapter is that of a security property, specifically that a given name always remains private to a given process. Behavioural equivalences on typed processes are investigated. In Chapter 10, reasoning about typed processes in the context of subtyping with input and output channels (as introduced in Chapter 7) is investigated, with an application towards the asynchronous π-calculus. Process equivalence in the presence of input and output channel types is then studied. In Chapter 11, a similar development is done for type systems capturing linearity, responsiveness, and polymorphism.

Part V, **The higher-order paradigm**, provides an in-depth study of the notion of mobility. As we saw in the introduction, mobility in the π-calculus is modeled by allowing channel names to be sent and received via channels. This is also called name-passing (or first-order). A more concrete approach to mobility is to allow the ability for processes to send and receive entire processes, an approach called process-passing (or higher-order). Mathematically, name-passing is much simpler than process-passing. On the other hand, process-passing is a more intuitive way to model mobility. It turns out that allowing process-passing does not add to the expressive power of the π-calculus, that is, anything expressible using process-passing is already expressible using name-passing. To make this formal, Chapter 12 introduces a higher-order typed π-calculus, HOπ, and develops its basic theory. The idea is to define the notion of a process abstraction that can be communicated across channels. In Chapter 13, it is shown how to translate HOπ into the π-calculus in a satisfactory way. Intuitively, the communication of a process abstraction translates into the communication of access to that abstraction. The translation is such that it reflects and preserves the equivalence of processes: two terms in the higher-order language are equivalent if and only their respective translations are equivalent, using the appropriate notions of equivalence.

The last two parts of the book address the relevance of the π-calculus to the theory of programming languages. Part VI, **Functions as processes**, explores the relation between the λ-calculus, a standard calculus for modeling sequential computations, and the π-calculus. It turns out that it is possible to encode the λ-calculus in the π-calculus, essentially turning the functions of the λ-calculus into processes that accept a value on a preselected input channel (the parameter channel), and returns a value on a preselected output channel (the result channel). In Chapter 14, the relevant theory of the λ-calculus is reviewed. The reader is expected to have had prior exposure to this topic, as the treatment is fast. In Chapter 15, the basic encoding is presented, which essentially amounts to a transformation of λ-terms into continuation-passing style, where each function takes an extra functional argument (the continuation) to which the result of the function call is passed. Different encodings can be given, corresponding to the different reductions strategies possible for the λ-calculus (call-by-name, call-by-value, etc...). Chapter 16 does the same for the typed λ-calculus. Chapters 17 and 18 explore properties of a particular encoding, that of the untyped call-by-name λ-calculus. The property of interest is that of the equality relation induced on λ-terms when their respective encodings are behaviourally equivalent as processes (according to different notions of process equivalences, but mostly barbed congruence).

Part VII, **Objects and π-calculus**, develops the relationship between the π-calculus and object-oriented programming. The intuitive similarities between these may not be completely clear at first glance, until one describes object-oriented systems as made up of objects that interact by invoking each other's methods, a procedure akin to sending messages. In Chapter 19, this is made manifest by introducing a simple object-oriented programming language, OOL, and by showing how to give it a semantics by translation to the π-calculus. In Chapter 20, some formal properties of OOL are examined, illustrating the use of π-calculus techniques in reasoning about objects. For example, one can check the correctness of certain program transformations, or the implementation of objects by separating code shared between instances of an object and data private to each instance.

11 Opinion

The basic recommendation I have is that anyone with a technical interest in the π-calculus needs this book. It brings together much of what is know about the calculus, information that for the most part can only be found in technical research articles. As such, it should remain the *de facto* reference work on the π-calculus for a great many years to come. I will even go as far as predicting that it will play the same role for the π-calculus that Barendregt's seminal book [1] plays for the λ-calculus.

It should be emphasized, however, that this is a reference book, not a textbook. It requires a good level of mathematical maturity, and furthermore, it requires prior exposure to the problems intrinsic to modeling concurrent systems, as well as the motivation underlying the process calculi approaches to those problems. Because of this, prior to reading the book, the neophyte should really first read Milner's own introduction to the π-calculus (in fact, to process calculi in general, and CCS in particular). Milner's original book on concurrency [7] is also suitable as an introduction to the material.

The need for a good reference for the basic theory of the π-calculus is clear when one looks at current work based on process calculi. Systems based on or inspired by the π-calculus are being used to study various aspects of security, for instance. The Ambient Calculus of Cardelli and Gordon [2] aims at modeling and reasoning about the notion of a locale in which computations execute, and in and out of which computations can move; the Spi-calculus of Gordon and Abadi [3] aims at modeling and reasoning about cryptographic protocols; the type system for processes developed by Honda et al. [5] aims at restricting statically the kind of information flowing from processes deemed high-security to processes deemed low-security.

References

[1] H. P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. Studies in Logic. North-Holland, Amsterdam, 1981.

[2] L. Cardelli and A. D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000.

[3] A. D. Gordon and M. Abadi. A calculus for cryptographic protocols: The Spi calculus. In *4th ACM Conference on Computer and Communications Security*, 1997.

[4] C. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[5] K. Honda, V. Vasconcelos, and N. Yoshida. Secure information flow as typed process behaviour. In *European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 180–199. Springer-Verlag, 2000.

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

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

[8] R. Milner. *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, 1999.

[9] D. Sangiorgi. *Expressing mobility in process algebras: first-order and higher-order paradigms*. PhD thesis, Department of Computer Science, University of Edinburgh, 1993. CST-99-93, also published as ECS-LFCS-93-266.