1

Introduction

In this book, we are interested in systems that consist of programmable components and encounter probabilistic impact. We find such systems in many application areas, i.e., whenever a software-intensive system operates in a dynamic, vague environment: control systems, production systems, logistics systems, socio-technical systems of any kind. Also, the components of these systems themselves may show probabilistic behavior. However, probabilistic programs are interesting also in their own right, i.e., even if the probabilism is not a circumstance that we need to deal with, but is generated for the sake of probabilistic programming itself. This is the field of randomized algorithms. Randomized algorithms become ever more important in practice, in particular, in the field of cryptographic systems. In this book, we are interested in the semantics of probabilistic programs. And we are interested in systematic reasoning about probabilistic programs.

We take a foundational, completely reductionist approach. We narrow our investigation to a maximally reductionist programming language, the typed lambda calculus with recursion. We enrich this lambda calculus by a single programming construct for probabilistic choice, which yields the probabilistic typed lambda calculus, which we also often call just the probabilistic lambda calculus for short. We investigate both the operational semantics and the denotational semantics of the resulting calculus. First, we will delve into the operational semantics. On top of its basic operational semantics, we systematically give a Markov chain semantics to the probabilistic lambda calculus. This way we unlock the whole machinery of probability theory, in general, and Markov chains, in particular, for reasoning about probabilistic program systems. We use this mathematical machinery to systematically investigate the termination behavior of probabilistic programming systems. We will come up with a broadened notion of termination, so-called path stoppability. Path stoppable programs have a finite term cover, therefore, linear algebra can be exploited to determine the probability with which program outcomes are reached. Our investigation yields a precise infinitesimal understanding of the termination degree of a program.

D. Draheim, *Semantics of the Probabilistic Typed Lambda Calculus*,
DOI 10.1007/978-3-642-55198-7_1

Next, we will define a denotational semantics of the probabilistic lambda calculus. Denotational semantics is the Scott-Strachey approach to the semantics of programming languages [243]. The mathematical beauty of denotational semantics stems from its compositionality resp. de-compositionality. A denotational semantics is given inductively along the abstract syntax of a programming language. It establishes a correspondence of syntactical and semantical constructors. This way, a denotational semantics achieves implementation independency [238], i.e., it can be considered the specification of a programming language. Denotational semantics has developed into the de facto standard for the investigation of programming language semantics, see [192, 253, 224, 117, 103, 118]. Moreover, denotational semantics gave rise to domain theory, see [239, 240, 105, 116, 106] and also [224, 117, 3]. Domain theory provides and investigates appropriate mathematical structures for programming language semantics. However, domain theory is not only important when teamed together with denotational semantics, rather, it yielded important results in its own right. Most importantly, in [240] Dana Scott found a model for the untyped lambda calculus. In [242], Dana Scott incorporates probabilism into such a model. This way, the semantics for untyped probabilistic lambda calculi, called stochastic lambda calculi in [242], has been achieved.

In this book we deal with the typed probabilistic lambda calculus. And also with respect to denotational semantics we only deal with the typed version of the probabilistic lambda calculus. We will define a denotational semantics for the probabilistic lambda calculus based on ω-cpo*s* (ω-complete partial orders). We use pure probability pre-distributions as ω-cpo*s*. In the case of call-by-name semantics the probabilistic choices at higher types can be flattened denotationally to probabilistic choices of ground type. This way, in case of call-by-name we need distributions only in the construction of the base domains but no nested constructions as higher types. Also, we prove the basic corresponce between the denotational semantics and our operational Markov chain semantics.

In the upcoming sections we will give a more detailed overview of the motivation and the contributions as well as pointers to important literature. In Sect. 1.1 we motivate the book's investigations from the perspective of system simulation and system analytics as well as from the perspective of randomized algorithms. In Sect. 1.2 we outline the probabilistic lambda calculus and its Markov chain semantics. In Sect. 1.3 we explain what will be achieved with respect to the analysis of termination behavior of probabilistic programs. In Sect. 1.4 we give an outline of our denotational semantics. In Sect. 1.5 we give further remarks on the book's content and provide a chapter outline of the book.

1.1 Motivation

In the IT sector, we have many systems that are mere data-processing systems. Their task is to capture some data, maybe transactional, and to process and keep them for us [83, 82, 34, 89, 86, 79, 78]. We may find such systems in enterprise computing [85, 84, 75, 10], e.g., enterprise resource-planning systems, customer relationships management systems, any kind of master data management systems, e.g., identity management systems, any kind of data repositories, optical archives, information bases and so forth. Of course, all of these systems are no silos and offer interfaces to their environments. They are used by humans and also other information systems. However, the interaction with such systems is usually confined to data access.

On the other hand, we see many software-intensive systems that are highly engaged, i.e., actively engaged, with their environment. They react to external triggers. They adapt to the changing state of their environment. This is, in a sense, the realm of agent-oriented systems [261, 132, 133]. Examples stem from the field of control systems, robotics, manufacturing execution systems, production planning systems or logistics systems. The degree of interaction, e.g., in terms of frequency, reaction time or criticality may greatly vary. Also, the frontiers to the data-processing systems mentioned above vanish more and more in today's system landscapes: business process management systems [74, 7, 10, 73] become adaptive [167, 229, 260, 76, 135, 88, 8, 80], decision support systems become reactive [120, 256], etc. In general, IT systems become ever more adaptive. With cloud computing [100, 77], elasticity of IT infrastructure has become mainstream [188].

Also, we might want to deal with internal components of the system that show probabilistic behavior, i.e., we need to deal not only with external but also internal probabilistic events.

The described systems are all highly relevant and there are two very important and huge communities that deal with them, i.e., the community of *model checking* and the community of *model-based design*. We will give some further remarks on model checking and model-based design together with hints to the respective literature in due course in Sect. 1.5. On the programming side, reaction to probabilistic events always gives rise to non-determinism. If we know the statistical distribution of the relevant external events this opens the opportunity of probabilistic reasoning about the overall system. If we have a rigorous model of how probabilities propagate through the programmed systems, we can make assumptions on program outcomes and overall system behavior. Given the Six Sigma approach [125, 126] the potential of such statistically founded reasoning should be immediately clear for the field of numerical control and manufacturing execution systems. Consider an example from a higher-level use case, i.e., from the domain of logistics. Consider a stock management system. A stock management system would react to events concerning the amount of incoming goods, overrunning or too scarce stockpiles and maybe other variable resources such as availability of employees and

so forth. Based on that it would automatically order goods or re-distribute goods. Now, as a mature option, the dataflow through a single storage could be modeled with queueing theory [101, 115, 268, 31], also, the supply chain consisting of a network of several storages could be modeled as a net of queues. Other options to model the storages and their network exist. For example, we could try to exploit stochastic extensions of Petri nets [119, 64, 13] or stochastic extensions of process algebra [107, 134, 50] for this endeavor. Anyhow, our reasoning about the programmed system in the single stores should fit into the overall probabilistic model to enable seamless reasoning. Therefore, we need a rigorous semantics of the programmed system involving probabilities. Our approach is reductionist. We will choose the lambda calculus as a candidate for our investigations. We will extend it by a probabilistic choice and give a Markov chain semantics to it. A probabilistic choice can be thought of as an input channel, yielding a reductionist, binary external information "left" or "right", i.e., the decision where to move next. This approach can be considered a first, fundamental step in the direction of an overall target – the switch from simulation of systems to a systematic analytical approach.

By the way, against the background of the above storage example, it is worth noting that there exist several business-process-modeling tools that offer features for process simulation. Simulation is exactly about forecasting the behavior of the modeled system like the flow of goods through a net of storages as described above. However, to our best knowledge, we do not know of a single business-process-modeling tool that incorporates queueing theory to support an analytical approach. This means, although with queueing theory we have a powerful tool to analyze systems, queueing theory is not yet consumable, i.e., it is not yet brought to the end-user, not yet brought to the average domain expert working in the field. Now, in computer science, or, to be more precise, in the field of software engineering, we actually have a long tradition in approaching simulation systematically. The original motivation of the first object-oriented programming language Simula [212] was to create a framework and language for system simulation. So, approaching simulation systematically was at the root of the early object-oriented paradigm. Only later, object-oriented programming turned into a paradigm of reusable [104], self-responsible software components that was particularly well-suited to serve the needs of upcoming object-oriented user interfaces. The original motivation towards system simulation has been a little bit forgotten in the object-oriented programming languages and design community. Instead, we can recognize the simulation approach as a basis of the agent-oriented system and modeling paradigm [261]. Nevertheless, in a sense, the original vision of Simula simply has been turned into reality. Today's object-oriented programming languages are mature candidates to specify executable system models – it's just about programming. What we need is to raise the level to system analytics. And there is a need to do this systematically, by building analytical features into existing programming and modeling environments. Even better, we should

think about systematically applying automatic or semi-automatic reasoning platforms [40, 39, 209] in the field of software-intensive systems.

So far, we have considered probabilistic programming languages as embedded into probabilistic environments. Here, the probabilistic programming language is used to program components that have to react to probabilistic events. In this perspective, our reductionist probabilistic choice models an input channel that delivers information about events from the environment to the programmed component. However, probabilistic programming is important in its own right, in order to program algorithms stand-alone, without relationship to an environment. Here, the probabilistic choice is fed by a random generator, i.e., it is a programming element instead of representing an information channel to an outside system. This is the field of randomized algorithms. Randomized algorithms can speed up the solution of problems dramatically, at the price of yielding an erroneous result occasionally. For a seminal work on randomized algorithms see [226] by Michael O. Rabin, see also the work of Andrew C. Yao in [264] as well as Andrew C. Yao and F. Frances Yao in [265, 266]. The classical example of randomized algorithms is about primality testing, approached by Solovay and Strassen in [248] and Michael O. Rabin in [227]. For an overview on the topic of randomized algorithms see [205] and also [148]. For a thorough treatment of randomized algorithms including many use cases, see [204]. An introduction to the complexity theory of randomized algorithms can be found in the standard text book on automata theory [137].

Now, a formal semantics of probabilistic programming languages also allows for reasoning about randomized algorithms. Reductionist models of randomized algorithms have been given as probabilistic automata in [68] and [225]. For a formal treatment of complexity of randomized algorithms see the seminal text book on automata and language theory by Hopcroft, Motwani and Ullman [137], which encompasses the definition of complexity classes on the basis of the probabilistic Turing machine. An important source concerning the complexity theory of randomized algorithms can be found in the quantum computing literature. See [208] for a comprehensive text book on quantum computing and [25] for a survey of quantum computing complexity theory that also clarifies the relationship between the complexity of randomized algorithms and the complexity of quantum computing algorithms.

1.2 The Probabilistic Lambda Calculus

The typed lambda calculus with recursion can be considered a most reductionist functional programming language. It can be considered the essence of functional programming languages like ML [190, 111, 193] and Haskell [138], compare also with [24, 23, 102]. Furthermore, the typed lambda calculus with recursion has been subject to intensive investigation in the semantics of programming language community, where it is also called PCF (Programming

Language for Computable Functions) [223] on many occasions. Therefore, we have chosen it as our candidate for investigating the semantics of probabilistic computation. It is the task of this section to explain how the probabilistic lambda calculus emerges as an extension of the typed lambda calculus with recursion, see Figs. 1.1 and 1.2. We have preferred the probabilistic lambda calculus over other options such as the probabilistic Turing machine [68, 225]. As opposed to the Turing machine, the lambda calculus comes with a particular 3GL (third generation language) or even 4GL (fourth-generation language) look and feel, albeit in a most reductionist form. Programs of the probabilistic lambda calculus are particularly intuitive and easy to read, because of their high similarity to the mathematical notation of recursive functions. Similarly, the probabilistic lambda calculus is particularly amenable to a denotational treatment that we will also provide in this book.

1.2.1 The Typed Lambda Calculus with Recursion

The lambda calculus in its original form, as introduced by Alonzo Church [47, 48], is an untyped language. If the lambda calculus is enriched by a type system, it is also called the simply typed lambda calculus [19]. In order to gain a Turing-complete typed calculus, an explicit recursion construct μ is added to the simply typed lambda calculus. Then, the resulting calculus is usually called the typed lambda calculus with recursion. Henceforth, we will also refer to this calculus simply as the lambda calculus or λ-calculus if it is clear from the context which calculus we mean.

Basically, the lambda calculus is introduced as a syntax and its operational semantics – see Fig. 1.1. The syntax is defined as context-free syntax plus a type system which specifies the notion of well-typed term. The operational semantics is introduced in two stages. First, the so-called immediate reduction relation is defined. This specifies which single steps are possible between terms. Actually, the immediate reduction relation is a partial function from terms to terms. The constants are considered the result values of programs. The constants and all of those terms that are outermost abstractions are the so-called values of the lambda calculus. It is not possible to do a further step from a value. Values stop computations. For each other term M there exists exactly one successor term N to which a next step is possible, which is denoted by $M \to N$. This right-uniqueness of the immediate reduction relation makes the considered lambda calculus a deterministic calculus. The considered lambda calculus is deterministic, because a concrete reduction strategy, in our case call-by-name, is fixed for it.

The immediate reduction relation forms the first stage of the lambda calculus' operational semantics. As the second stage the reduction relation is defined as the transitive, reflexive closure of the immediate reduction relation. We say that a term M reduces to another term N if it is connected to it – in the correct direction – via the reduction relation, which is denoted by

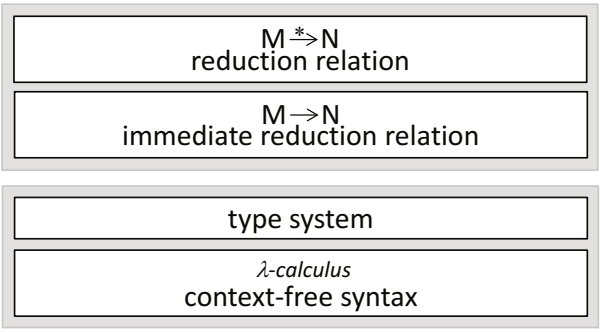

Fig. 1.1. Syntax and semantics of the λ-calculus

$M \xrightarrow{\star} N$. We then also say that N is reachable from M. It is very interesting to recognize that the specification of the immediate reduction relation is already sufficient for an implementation of the lambda calculus as a programming language. It might be a bit unfair, but we could say that the reduction relation does not add anything else but some terminology, i.e., the notion of reachability of terms. It is actually unfair, because semantics is, first, about the agreement of what we actually intend with a formal language and, second, a means to get access to further semantical tools that can be used in reasoning about the program. Obviously, we need to agree upon the notion of program results and, furthermore, an understanding of the reduction relation as a transitive closure will be needed in formal argumentations on program behavior. Nevertheless, the definition of the immediate reduction relation is not sufficient for all investigations of program behavior that we might want to conduct. Further notions, such as program reduction trees, are usually defined on top of it to gain appropriate mathematical tools for program reasoning. Anyhow, it is worth noting that with respect to the behavior of programs, the immediate reduction relation is already a complete specification. This viewpoint will also be useful in the comparison with the probabilistic lambda calculus in due course.

In general, it is not necessary to fix a reduction strategy for a lambda calculus, so that a lambda calculus may encounter some level of non-determinism. However, such non-determinism does no harm, or, to say it better, makes no difference with respect to the program outcomes. This is so due to the Church-Rosser property [49]. If a program is able to reduce to a constant, i.e., if it is a terminating program, then this constant is uniquely given. Furthermore, we can reduce to it from any reachable intermediate term. However, the latter is not so important for us here. What interests us here is the uniqueness of terminating program outcomes. This actually guarantees a certain level of determinism. With respect to terminating program results, each lambda calculus is deterministic, independent of the chosen reduction strategy. And here also lies the difference between the non-determinism encountered in lambda

calculi in general as opposed to the probabilistic lambda calculus in this book. In the probabilistic lambda calculus, a program may terminate with different constants.

1.2.2 The Probabilistic Lambda Calculus Compared

Now, let us turn to the probabilistic lambda calculus, see Fig. 1.2. Syntactically, the probabilistic lambda calculus is just the typed lambda calculus with recursion plus a program construct for probabilistic choice. Of course also the type system is adjusted to the new terms. Now, for any two terms M and N, the term $M|N$ denotes the probabilistic choice of M and N. Given a program $M|N$, this program executes with fifty per cent probability as M and, equally, with fifty per cent as N. Again, we give the semantics in two stages. First we define a one-step semantics. However, this time the one-step semantics is a total function that assigns to each pair of terms M and N the probability i with which the program may move from M to N, which is then denoted by $M \xrightarrow{i} N$.

Of course, all terms of the plain lambda calculus are also terms of the probabilistic lambda calculus. Each immediate reduction $M \to N$ can be found in the probabilistic lambda calculus as step $M \xrightarrow{1} N$. Of course, the crucial difference lies in terms of the form $M|N$. In case that $M \neq N$ we specify two possible reductions, i.e., a reduction $M|N \xrightarrow{0.5} M$ and a reduction $M|N \xrightarrow{0.5} N$. For each term P that is different from both M and N, we define an immediate one step reduction $M|N \xrightarrow{0} P$ to ensure that the one-step reduction becomes a total function. In case of terms $M|M$ we will introduce a possible reduction $M|M \xrightarrow{1} M$. Again, for each term P different from M we define a one-step reduction $M|M \xrightarrow{0} P$.

On top of the one-step semantics we then define the Markov chain semantics of the probabilistic lambda calculus, see Fig. 1.2. We take closed terms of the probabilistic lambda calculus as the states of a Markov chain S. We take the one-step semantics as the transition matrix of this Markov chain. Now, we define the reduction semantics of the probabilistic lambda calculus via hitting probabilities of the Markov chain. The probability to reduce from a term M to a term N, denoted by $M \Rightarrow N$, is defined as the Markov chain's probability of starting in M and ever hitting N. With the Markov chain semantics we inherit all results from probability theory and Markov chains for reasoning about program behavior of probabilistic programs. For example, we can determine reduction probabilities as least solutions of linear equation systems.

In this book we consider the lambda calculus under the call-by-name strategy. This is a very common choice: note that also PCF [223] is considered with the call-by-name strategy. The choice of call-by-name is convenient. Actually, the results on termination behavior in Chap. 4 are independent of the call-by-name strategy, i.e., they would also hold under a concrete call-by-value

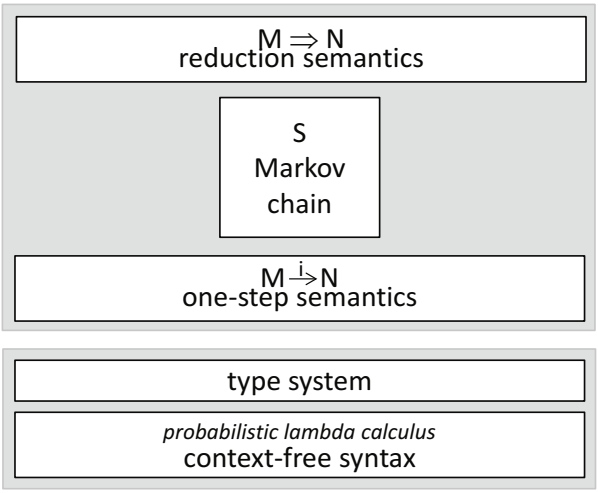

Fig. 1.2. Syntax and semantics of the probabilistic lambda calculus

strategy. However, the choice of call-by-name is crucial for the denotational semantics in Chap. 5. Here, the choice of call-by-value would result in different domains than the chosen ones.

1.3 Termination Behavior of Probabilistic Programs

In the deterministic lambda calculus a program either terminates or does not. Each program corresponds to exactly one program run. This is not so any more with the probabilistic lambda calculus. Here, a program results, in general, in one out of many possible program runs. It is the single program run that may terminate or not. Now, a non-deterministic program may have some terminating program runs plus some non-terminating program runs. However, we will define the notion of termination degree for probabilistic programs. The termination degree of a program is the probability that it will ever reach one of the constant values. Or, to say it differently, the termination degree of a program is the accumulated probability of all of its terminating program runs.

Let us coin the somehow artificial term of strictly terminating program. We say that a program strictly terminates if all of its program runs terminate. Actually, with respect to terminology we are in a bit of a dilemma. We would like to avoid calling a non-deterministic program a terminating program. The problem is that we have the interesting class of programs with termination degree one. In this class, there are also programs that may have a non-terminating program run. For example, the program $M = \mu\lambda x.(x|0)$ has a termination degree of one, however, it also has a non-terminating program run. The program M will reach the constant 0 with a hundred per

cent probability. On the other hand M has a non-terminating program run $M \xrightarrow{1} \lambda x.(x|0)\mu\lambda x.(x|0) \xrightarrow{1} (M|x) \xrightarrow{0.5} M \xrightarrow{1} \cdots$ which is kept in an endless loop back to the starting term M. However, this terminating program run has a probability of zero. Now, it would be fair to say that M is always terminating, or just terminating for short. We prefer to say that M has a termination degree of one and that a program that has no non-terminating program runs is strictly terminating. All this has to do with the original *explicatio* of probability theory given by Kolmogorov in [163] as a model of experimental data, and with how we usually speak about events that have zero per cent probabilities as impossible events. The termination degree of M equals the termination degrees of the programs $\mu\lambda x.0$ and 0 in our semantics, i.e., they are *infinitesimally* "equal".

Based on the Markov chain semantics we will be able to identify a broadened notion of termination, so-called path stoppability, see Fig. 1.3. Path stoppability is a notion of program analysis. It allows us to stop some non-terminating programs and to determine their termination degree. Let us walk through the example programs given in Fig. 1.3. The program $(\lambda x.x)(0|1)$ is a strictly terminating program. All of its program runs terminate. This is not so for the program $M = \mu\lambda x.(x|0)$. We have just discussed that this program has a non-terminating program run. We will design an algorithm, let us call it the path-stopping algorithm, that dovetails a given non-deterministic program and detects and stops all of its endlessly looping program runs. We will show that this algorithm terminates for all of those programs that have a finite cover. The path-stopping algorithm implements a program analysis. Now, program M has a finite cover, and therefore it is stoppable by the path-stopping algorithm. It falls into the class of path stoppable programs. This is not so for the program $(\mu\lambda x.(+1(x) \mid 0))$. The cover of this program contains infinitely many terms, e.g., all terms of the form $+1^n(0)$ for each number n.

The notion of path stoppability also applies to deterministic lambda programs. We have given the corresponding examples in Fig. 1.3. The program $(\lambda x.x)0$ serves as an example for strictly terminating programs, the most simple non-terminating program $\mu\lambda x.x$ as an example for path stoppable programs and the program $\mu\lambda x. + 1(x)$ as an example for programs that are not path stoppable.

For path stoppable programs, it is possible to compute their termination degrees. As a consequence, it is possible to compute all reduction probabilities for path stoppable programs. We will also speak of path computability or p-computability of reduction probabilities. In order to investigate the termination behavior of probabilistic programs as just outlined, we will need results from graph theory. Therefore, the Markov chain semantics is teamed together with the notion of reduction graph and the notion of reduction tree – see Fig. 1.4. We will interpret the one-step semantics as a graph, the so-called reduction graph R. Based on that we will precisely define program executions, program runs and term covers. We will tightly integrate the graph seman-

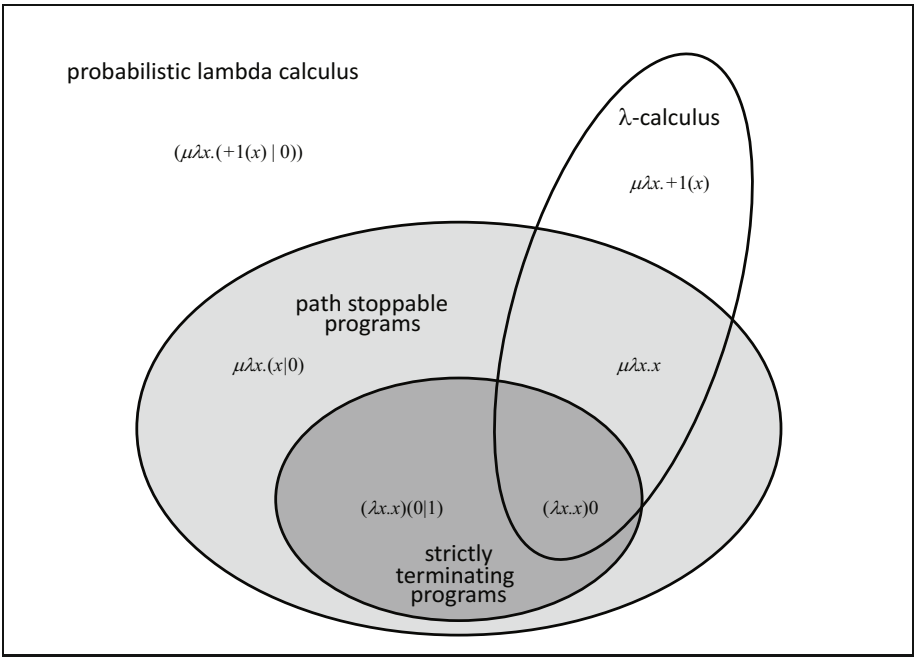

Fig. 1.3. The λ-calculus and the probabilistic lambda calculus compared

tics with the Markov chain semantics. This way we unlock graph theoretical results for reasoning about programs and their behavior. For example, we will exploit König's lemma in the investigation of termination behavior of programs. Similarly, we define a tree semantics on top of the one-step semantics. For each program M we define the tree $\tau[M]$ of program runs starting in M. Reduction trees are defined on the basis of the reduction graph R. They provide a tree-specific viewpoint on the reduction graph and unlock tree-specific graph theoretical results for program reasoning. For example we will use Beth's Tree Theorem, which is an instance of König's Lemma, in the investigation of bounded program termination.

1.4 Denotational Semantics

Denotational semantics is the Scott-Strachey approach to the semantics of programming languages [243]. The denotational semantics of a programming language is given directly in terms of the mathematical objects that are computed by the programs of a programming language. A denotational semantics targets implementation independency [238], i.e., it can be considered the specification of a programming language as opposed to the several possible implementations of this programming language. A major characteristic of denotational semantics is compositionality. This means that a denotational semantics

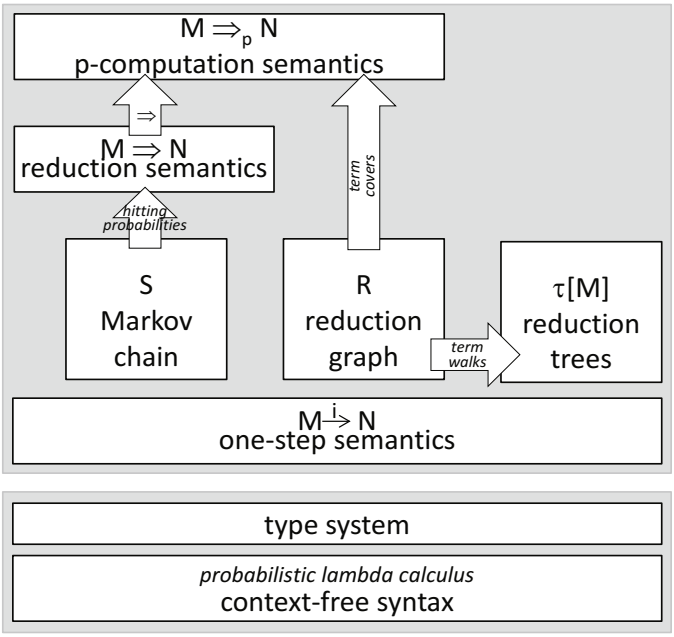

Fig. 1.4. Auxiliary operational concepts for the probabilistic lambda calculus

introduces an appropriate semantic constructor for each of the syntactic constructors of the programming language. Then, the denotational semantics is given inductively along the abstract syntax of the programming language and the correspondence of syntactical and semantical constructors. Given this inductiveness, a denotational semantics also reveals operational semantics, i.e., it can be considered a program of a recursive "meta" programming language. The point is that it does not have to rely on an operational semantics of this pre-assumed meta programming language, it itself receives its semantics from first principles, i.e., the notion of inductive definitions. For more on inductive definitions see also our primer on this topic in this book in Sect. 2.4.

Against the background of implementation independence, a denotational semantics can be considered as coming first. Then, it can be considered a specification in a programming language engineering process. However, also the opposite perspective is admissible. Here, the denotational semantics is defined for an existing programming language for the purpose of clarifying its semantics and, even more important, for unlocking the mathematical toolkit for reasoning about program behavior. At least when we treat reductionist programming languages like the lambda calculus, denotational semantics sometimes rather has this flavor of coming second, after the operational semantics.

The reductionist programming language and reasoning system Edinburgh LCF (Logics for Computable Functions) [237, 190, 191, 111], also just LCF for short, has been given a denotational semantics in [192]. The reductionist programming language PCF, which is also based on LCF, has been given a denotational semantics in [223]. A major task in denotational semantics is to establish the appropriate domains from which the semantical objects are drawn. A major challenge is to provide domains for recursively defined data types which naturally arise in programming languages. Dana Scott showed how to solve this problem of finding solutions of recursive domain equations even if function spaces are involved. In [240] he was able to construct a model of the untyped lambda calculus. A model of the untyped lambda calculus amounts to a function space that is isomorphic to itself, i.e., is a solution to the domain equation $D \cong D \to D$. In general, i.e., in the case of sets, it is impossible to find a solution to the domain equation $D \cong D \to D$, because the cardinality of $D \to D$ is larger than the cardinality of D for any set D. In [240] such a function space has been established on the basis of complete lattices.

The construction of domains for denotational semantics evolved into a discipline in its own right, i.e., domain theory. The foundational challenge addressed by domain theory is to tighten the correspondence between the denotational semantics and the operational semantics of programming languages and to provide optimized domains for this purpose. Orthogonal to this challenge, domain theory provides specialized domains to address various programming language phenomena such as non-determinism or parallelism. For texts on domain theory, see [116, 224, 3]. For comprehensive texts on denotational semantics in general, see, e.g., [253, 235, 117, 203, 118, 263].

We will elaborate a denotational semantics for the probabilistic lambda calculus. The chosen approach is straightforward and standard from the literature; compare with the work of Saheb-Djaromi in [232, 233]. Our denotational semantics is based on ω-complete partial orders, i.e., ω-cpo*s*. We have that ω-cpo*s* are, among other domains such as, e.g., complete lattices, directed complete partial orders, a well-known choice for denotational semantics. They have been used very early for this purpose, i.e., in the definition of a denotational semantics of Edinburgh LCF, see [192]. As basic mathematical objects of our denotational semantics, we will use vectors of real numbers in $[0, 1]$. These vectors assign probability values to data points. We call these vector probability pre-distributions, because they play the same role as distributions. There is a crucial difference to distributions. Their total mass, i.e., the sum of all the values of all possible data points, does not necessarily have to be a hundred per cent in case of pre-distributions. And this makes sense. A total mass of less than a hundred per cent stands for the existence of some non-terminating program runs. Actually, we will call the total mass of a pre-distribution the degree of termination later. As a result, we work without explicit bottom elements representing non-termination. Bottom elements arise implicitly as distributions that assign a zero per cent probability to each

data point. Furthermore, we will see that probabilistic choices at higher types can be flattened completely to probabilistic choices of ground type. This way, we need distributions only in the construction of the base domains. All the domains of higher type that we introduce in our semantics turn out to be vector spaces, which greatly eases our argumentations and formal proofs.

Given a program M of the probabilistic lambda calculus, we will denote, as usual, the semantical object assigned to it as $[\![M]\!]$. Given a number constant n_i, its semantics is given as the (pre-)distribution that assigns a hundred per cent probability to the so-called data point i, i.e., $[\![n_i]\!](i) = 1$ and $[\![n_i]\!](j) = 0$ for all $j \neq i$. Next, we also say that n_i represents the data point i and use $[n_i]$ to denote it, i.e., $[n_i] = i$. Similar definitions can be given for the other ground type of the probabilistic lambda calculus, i.e., Boolean. Once we have introduced the denotational semantics for the probabilistic lambda calculus, we will investigate its relationship to the operational semantics. We will prove the one-to-one correspondence at the base element level. More concretely, this correspondence means that the semantics of a program M, applied to a data point $[c]$, equals the probability with which M reduces to that constant c that represents $[c]$, which can expressed in the notation that we have introduced so far as follows: $([\![M]\!][c]) = (M \Rightarrow c)$.

1.5 Chapter Outline and Further Remarks

Chapter 2 is a preparatory chapter. It recaps and comments on basic mathematical tools needed throughout the book, in particular, from the fields of probability theory, Markov chains, graph theory and domain theory. Also, it delves into the topic of inductive definitions, because they form the foundation of rigorous specification of semantics. In Chap. 3 we define the syntax and establish the Markov chain semantics of the probabilistic lambda calculus. To improve comparability we also introduce the operational semantics of the plain, deterministic lambda calculus. In Chap. 4 we investigate the termination behavior of the probabilistic lambda calculus. We define the notion of termination degree. Furthermore, we define the notion of bounded termination, which is about programs that do not exceed an upper bound of steps whenever they terminate. We define the notion of path stoppability that we have described above as a broadened notion of termination. Then, we systematically investigate the mutual dependencies between path stoppability, bounded termination and termination degrees of a hundred per cent. To achieve all this, the chapter establishes the graph semantics as well as the tree semantics of the probabilistic lambda calculus. Also, it shows some needed graph cover lemmas, i.e., the fact that the cover of a graph is already completely determined by the cover of its paths and, second, that the finite cover of a graph is computable for all k-ary graphs that have a finite cover and a computable edge relation. Finally, in Chap. 5 we define a denotational semantics of the probabilistic lambda calculus, based on functionals

over probability distributions as domains. As the basic semantic correspondence, we show the correspondence of the denotational semantics with respect to the Markov chain semantics.

The main focus of this book is the Markov chain semantics for the probabilistic higher-order typed lambda calculus. It is a natural idea to treat probabilistic computation with Markov chains, compare, e.g., to the textbook [204] of Motwani and Raghavan on randomized algorithms and, again, to the work on the probabilistic lambda calculus by Saheb-Djahromi in [232, 233]. Compare also with, e.g., [128, 174, 38, 94, 110, 175, 72] to name a few. The book's aim is to fully elaborate the Markov chain semantics, i.e., to elaborate it to a level that systematically unlocks results from Markov chain theory to be used in reasoning about program semantics and, in particular, establishing further formal results. This can be understood best by looking at how we exploit the established semantics to investigate termination behavior of probabilistic programs and come up with and further investigate notions like path stoppability, path computability and bounded termination.

The focus of the book is narrow in several ways, i.e., with respect to the probabilistic programming phenomena it delves into, the chosen programming language paradigm, the programming language semantic approaches it exploits, the level of investigation and the motivation it stresses for its investigations. All of these aspects are mutually dependent. First, we deal with the functional programming language paradigm and here we have chosen the most reductionist language, i.e., the typed lambda calculus also known as Plotkin's PCF [223]. A thorough treatment of probabilistic imperative programming is provided by the book [185] by McIver and Morgan.

We delve into operational semantics. Also, we work with denotational semantics. We treat denotational semantics only as far as we feel is needed to bridge our treatment into the extremely mature body of knowledge that is established by denotational semantics and domain theory for probabilistic computation, see Sect. 5.4 on selected important readings in the field. We do not look into axiomatic semantics [136] of probabilistic computing. Once more, we want to recommend the book [185] by McIver and Morgan as an authoritative reference. The important field of axiomatic semantics is so rich with respect to probabilistic computation, we do not even attempt to give a literature overview, again, the reader might want to use [185] as a good starting point.

The field of model checking is extremely mature with respect to probabilistic systems. The overview article [175] of Legay, Delahaye and Bensalem provides a good entry point into the subject matter. The reason why the field of model checking is particularly important is because it does not stop at defining and investigating formal logics and reasoning systems, but actually provides concrete implementations of model-checking platforms or model checkers. Important probabilistic model checkers are ETMCC [131], Prism [172], Ymer [267], Vesta [244], and MRMC [149]. The list does not aim to be complete, nor does it express preference or priority. A rigorous comparison of these

model checkers is provided in [140] by Jansen et al. A typical model checker for probabilistic systems allows us to model systems as discrete- and continuous Markov chains. Based on these models, it allows for system simulation and automated or semi-automated verification of system properties. Probabilistic versions of the temporal logics CTL and CTL*, i.e., pCTL and pCTL*, have been introduced by Hansson and Jonsson in [121]. In [11, 12], Aziz et al. introduce the logic CSL (Continuous Stochastic Logic), which allows for reasoning about continuous-time Markov chains, compare also with [14], which treats, in more depth, model checking aspects of CSL. As we said, we consider model checking based on Markov chain models as typical, albeit other important approaches exist such as, e.g., the application [30] of uniform continuous-time Markov decision processes [15] to statecharts [122, 123], the GreatSPN tool [64, 13] with respect to Petri nets, or the PEPA workbench [107, 134] with respect to stochastic process algebras.

It is also important to mention the industrial-strength modeling environment Simulink [257] and, in particular, also its module SimEvents [114, 44]. Simulink is based on Matlab. It supports model-based design, a combination of modeling, simulation, code generation for and verification of dynamic systems consisting of both discrete and continuous switching blocks. Its original domain is the domain of control systems and it is applied at several levels and instances thereof ranging from manufacturing execution systems over embedded systems to circuit design. The module SimEvents extends Simulink by queueing-system building blocks.

Programming languages such as IBAL [218], Church [110] and Venture [180] are programming languages that contain a probabilistic programming primitive. The purpose of these languages is to express stochastic models, i.e., to generate stochastic models. They allow for querying distribution outcomes against the traces of a probabilistic program. This way, these programming languages become decision support tools. They are called probabilistic programming languages by Stuart Russell in [231] or stochastic programming languages in [110], however, the language IBAL is called a rational programming language by Pfeffer in [218], because the purpose of such languages is to support reasoning about rational agents. A precursor of IBAL was already developed by Pfeffer et al. in [161].

http://www.springer.com/978-3-642-55197-0

Semantics of the Probabilistic Typed Lambda Calculus
Markov Chain Semantics, Termination Behavior, and
Denotational Semantics
Draheim, D.
2017, VIII, 218 p. 6 illus., Hardcover
ISBN: 978-3-642-55197-0