Chapter 1

Introduction

A *language* is a systematic means of communicating ideas or feelings among people by the use of conventionalized signs. In contrast, a *programming language* can be thought of as a syntactic formalism which provides a means for the communication of computations among people and (abstract) machines. Elements of a programming language are often called *programs*. They are formed according to formal rules which define the relations between the various components of the language. Examples of programming languages are conventional languages like Pascal [196] or C++ [187], and also the more theoretical languages such as the λ-calculus [52,26] or CCS [144].

A programming language can be interpreted on the basis of our intuitive concept of computation. However, an informal and vague interpretation of a programming language may cause inconsistency and ambiguity. As a consequence different implementations may be given for the same language possibly leading to different (sets of) computations for the same program. Had the language interpretation been defined in a formal way, the implementation could be proved or disproved correct. There are different reasons why a formal interpretation of a programming language is desirable: to give programmers unambiguous and perhaps informative answers about the language in order to write correct programs, to give implementers a precise definition of the language, and to develop an abstract but intuitive model of the language in order to reason about programs and to aid program development from specifications.

Mathematics often emphasizes the formal correspondence between a notation and its meaning. For example, in mathematical logic, we interpret a formal

theory on the basis of a more intuitive mathematical domain which properly fits the theory (that is, the interpretation of all theorems must be valid). Similarly, the formal *semantics of a programming language* assigns to every program of the language an element of a mathematical structure. This mathematical structure is usually called the *semantic domain*. Several mathematical structures can be used as semantic domain, and the choice as to which one is to be preferred often depends upon the programming language under consideration. Since a programming language is a formal notation, its semantics can be seen as a translation of a formal system into another one. The need for a formal semantics of a programming language can thus be rephrased as the need for a suitable mathematical structure closer to our computational intuition. From this mathematical structure we expect to gain insights into the language considered.

There are several ways to formally define the semantics of a programming language. Below we briefly describe the three main approaches to semantics, namely the operational, the denotational and the axiomatic approach. Other important approaches to the semantics of programming languages are given by the *algebraic semantics* [78,144,104,28,48,92], with mathematical foundations based on abstract algebras [79,61], and the *action semantics* [152], based on three kinds of primitive semantics entities: actions, data and yielders.

In the *operational semantics* one defines the meaning of a program in terms of the computations performed by an abstract machine that executes the program. For this reason the operational semantics is considered to be close to what actually happens in reality when executing a program on a real computer. *Transition systems* are the most commonly used abstract machines which support a straightforward definition of a computation by the stepwise execution of atomic actions. There are different ways to collect the information about the computations of a transition system which give rise to different operational semantics. Moreover, transition systems support a structural approach to operational semantics as advocated by Plotkin [161]: the transition relation can be defined by induction on the structure of the language constructs.

The *denotational* approach to the semantics of programming languages is due to Scott and Strachey [177]. Programs are mapped to elements of some mathematical domain in a compositional way according to the 'Fregean principle' [72]: the semantics of a language construct is defined in terms of its components. Due to the possibility of self-application given by some programming languages, the semantic domain must sometimes be defined in a recursive way. This is often impossible with an ordinary set-theoretical construction because of a cardinality problem. Therefore often a topological structure is associated

with the semantic domain which takes into account qualitative or quantitative information about the computations. Typical topological structures used for the denotational semantics of programming languages are complete partial orders, put forward by Scott [173,175], and complete metric spaces, introduced in semantics by Arnold and Nivat [11] and extensively studied by the Amsterdam Concurrency Group (for an overview see [22] and also [23]). The denotational semantics is close to the operational semantics but abstracts from certain details so that attention can be focussed on issues at a higher level.

The *axiomatic* approach characterizes programs in a logical framework intended for reasoning about their properties. Proof systems are usually used for axiomatic semantics: computations are expressed by relating programs to assertions about their behaviour. The most well-known axiomatic semantics is Hoare logic [101] for total correctness. Assertions are of the form $\{P\}\, S\, \{Q\}$ meaning that the program S when started at input satisfying the predicate P terminates and its output satisfies the predicate Q. There are many other kinds of axiomatic semantics using proof systems such as temporal logic [162], dynamic logic [163] and Hennessy-Milner logic [93]. Axiomatic semantics can also be given without the use of formal proof systems: the behaviour of a program can be expressed as a function which transforms predicates about the program. For example, Dijkstra's weakest precondition semantics [56] regards a program S as a function which maps every predicate Q on the output state space of S to the weakest predicate among all P's such that the Hoare assertion $\{P\}S\{Q\}$ is valid. Axiomatic semantics is closely related to the verification of the correctness of programs with respect to a given specification. An axiomatic semantics should preferably be such that the verification of the correctness of a program can be done by verifying the correctness of its components, as advocated by Turing [190] and Floyd [70] (see also the discussion in [19]).

The choice among the operational, the denotational or the axiomatic semantics for a programming language will depend on the particular goals to be achieved. To take advantage of these different semantic views of a program it is important to study their relationships.

The denotational semantics of a programming language is, by definition, compositional. Since an operational semantics is not required to be compositional we cannot have, in general, an equivalence between the two semantics. Two criteria about the relation between denotational and operational semantics are commonly accepted. The first criterion says that the denotational semantics has to assign a different meaning to those programs of the language which in some context can be distinguished by the operational semantics. This can be

achieved, for example, by proving the existence of an abstraction function that when composed with the denotational semantics gives exactly the operational semantics. In this case the denotational semantics is said to be *correct*, or adequate, with respect to the operational semantics. The second criterion looks for the most abstract denotational semantics which is correct with respect to a given operational semantics. This can formally be expressed by requiring that the denotational semantics assigns a different meaning to two programs of the language if and only if they can be distinguished in some context by the operational semantics. In this case the denotational semantics is said to be *fully abstract* with respect to the operational semantics [143].

The relationship between the denotational and the axiomatic semantics is the main topic of this monograph. Depending on which kind of information has to be taken in account, there are different transformations which ensure the correctness of one semantics in terms of the other. The common factor in all these transformations is that they form dualities rather than equivalences: the denotational meaning of a program viewed as a function from the input to the output space is mapped to a function from predicates on the output space to predicates on the input space. Conversely, the axiomatic meaning of a program regarded as a function from predicates on the output to predicates on the input is mapped to a function from the input space to the output space.

The dualities between the denotational and the axiomatic views of a program are often *topological* in the sense that they are set in a topological framework. This is motivated by the tight connection between topology and denotational semantics: topology has become an essential tool for denotational semantics and denotational semantics has influenced new activities in topology [179,1,192,182,47,146,23]. The fundamental insight due to Smyth [179] is that a topological space may be seen as a 'data type' with the open sets as 'observable predicates', and functions between topological spaces as 'computations'. These ideas form the basis for a computational interpretation of topology.

Abramsky [1,2], Zhang [199] and Vickers [192,5] carried the ideas of Smyth much further by systematically developing a propositional program logic from a denotational semantics. The main ingredient in their work is a duality (in categorical terms a contravariant equivalence) between the category of certain topological spaces and a corresponding category of frames (algebraic structures with two classes of operators representing finite conjunctions and infinite disjunctions). On one side of the duality, topological spaces can arise as semantic domains for the denotations of programs; on the other side of the duality, frames can arise as the Lindenbaum algebras of a propositional program

logic with properties as elements and proof rules provided by the various constructions. Accordingly, topological dualities are considered as the appropriate framework to connect denotational semantics and program logics [1,192,199].

From a broader perspective, topological dualities (in the form of representation theorems) can be used to characterize models of abstract algebraic structures in terms of concrete topological structures. Therefore the ultimate purpose of setting up a topological duality is to capture axiomatically the class of properties we have in mind. Let us quote Johnstone [112, page XX] to summarize the importance of topological dualities: 'Abstract algebra cannot develop to its fullest extent without the infusion of topological ideas, and conversely if we do not recognize the algebraic aspects of fundamental structures of analysis our view of them will be one-sided.'

The contributions of our work may roughly be classified into the following three kinds. The first kind of contribution consists in the characterization for a given language of an axiomatic semantics using insights from a denotational semantics. For example, we define a weakest precondition semantics for a sequential language with a backtrack operator using a simple denotational interpretation. We also characterize a compositional predicate transformer semantics for a concurrent language with a shared state space. The semantics is based on a denotational interpretation of the language given by considering programs to be functions abstracted from a transition system modulo bisimulation [24,76].

The second kind of contribution is dual to the first one: the characterization of a denotational semantics for a language using an axiomatic semantics. We characterize a denotational semantics for the refinement calculus (a language with an associated axiomatic semantics based on monotonic predicate transformers). We use the denotational semantics to derive a new operational interpretation of the refinement calculus based on hyper transition systems. The denotational semantics of the refinement calculus is proved fully abstract with respect to the operational interpretation (in fact they are equivalent).

The third kind of contribution is more abstract in nature. We have set up a framework for a systematic development of a propositional logic for the specification of programs from a denotational semantics. In particular, it gives a conceptual foundation which answers the question posed by Abramsky [2, page 74] about the possibility of expressing infinite conjunctions in the logic of domains. The logic derived from a denotational semantics by means of the duality between the category of certain topological spaces and the corresponding category of frames is not expressive enough to be used for specification

purposes: infinite conjunctions should be added [1,199]. However, such an extension would necessarily takes us outside open sets. Our contribution consists in the the development of an abstract algebraic framework which allows both infinite conjunctions and infinite disjunctions of abstract open sets. This framework is related to ordinary topological spaces by means of a representation theorem, and it is applied by deriving an infinitary logic for transition systems.

Outline of the chapters

This monograph is divided into three parts. In the first part we consider predicates as subsets of an abstract set of states. In the second part we refine the notion of predicates by considering affirmative predicates. They are open subsets of an abstract set of states equipped with a topology. Finally, in the third part we forget about states and we take predicates to be elements of an abstract algebra with algebraic operations to represent unions and intersections.

We start by introducing in Chapter 2 some basic concepts in category theory, partial orders, and metric spaces. Category theory is not needed for understanding the first two parts. Metric spaces will only play a major role in Chapter 7.

With Chapter 3 we start the first part. The chapter is about the semantics of sequential languages. In particular we consider the weakest precondition and the weakest liberal precondition semantics, and the relationships to various state transformer semantics. These relationships generalize the duality of Plotkin [159] between predicate transformers and the Smyth powerdomain. We also discuss the weakest precondition semantics of a sequential non-deterministic language with a backtrack operator.

In Chapter 4 we extend sequential languages with specification constructs. We use the language of the refinement calculus introduced by Back [13]. The refinement calculus is based on a predicate transformer semantics which supports both unbounded angelic and unbounded demonic non-determinism. We give a state transformer semantics for the refinement calculus and relate it to the predicate transformer semantics by means of a duality. We give also an operational interpretation of the refinement calculus in terms of the atomic steps of the computations of the programs. The latter operational view is connected to the state transformer semantics.

The second part begins with Chapter 5. In this chapter we refine the notion

of predicate introduced in Chapter 3. Following the view of Smyth [179], affirmative predicates are introduced as open sets of a topological space. Several basic concepts taken from topology are introduced and motivated from the point of view of affirmative predicates.

In Chapter 6 we rework in a topological framework the dualities between predicate and state transformers that were introduced in Chapter 3. These dualities show us how to generalize predicate transformers to topological predicate transformers. The latter can be used as domain for a backward semantics of non-sequential programming languages. Our starting point is Smyth's duality between the upper powerspace of a topological space and certain functions between affirmative predicates. We show that Smyth's duality holds in a general topological context. Also, we propose dualities for the lower powerspace and the (more classical) Vietoris construction on general topological spaces. In passing, several topological characterizations of metric and order based powerdomains constructions are investigated.

Chapter 7 is devoted to the semantics of a sequential non-deterministic language extended with a parallel operator. A domain of metric predicate transformers is defined as the solution of a recursive domain equation in the category of complete metric spaces. A compositional predicate transformer semantics is given to the language, and it is shown to be isometric to a state transformer semantics based on the resumption domain of De Bakker and Zucker [24]. Partial and total correctness properties are studied for the above language using a connection between the domain of metric predicate transformers and the two domains of predicate transformers given in Chapter 3. As a consequence, the semantics of a sequential language is obtained as the abstraction of the unique fixed point of a metric-based higher-order transformation, and is proved correct with respect to three order-based semantics obtained as least fixed points of three higher-order transformations, respectively. Also, we briefly discuss the study of temporal properties of a concurrent language via our metric predicate transformer semantics.

The third and last part starts with Chapter 8. We abstract from open sets and regard predicates as elements of an abstract algebra. We consider a topological space as a function from the abstract set of affirmative predicates (with algebraic operations representing arbitrary unions and finite intersections) to the abstract set of specifications (with algebraic operations representing arbitrary unions and arbitrary intersections). This structure is called an observation frame. We show that in certain cases topological spaces can be reconstructed from observation frames. We obtain a categorical duality between the category of certain topological spaces (not necessarily sober) and a corresponding

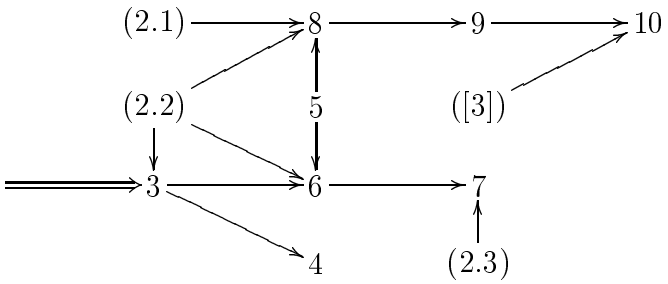
category of observation frames. We also give a propositional logic of observation frames with arbitrary conjunctions and arbitrary disjunctions. The logic is shown to be sound and complete if and only if the observation frame corresponds, canonically, to a topological space. Finally we apply the above theory in order to obtain dualities for various sub-categories of topological spaces.

Chapter 9 relates topological spaces seen as frames to topological spaces seen as observation frames. A new characterization of sober spaces in terms of completely distributive lattices is given. This characterization can be used for freely extending the geometric logic of topological spaces to an infinitary logic. We also show that observation frames are algebraic structures in a precise categorical sense.

We end our work with Chapter 10. In this chapter an extension of Abramsky's finitary domain logic for transition systems to an infinitary logic with arbitrary conjunctions and arbitrary disjunctions is presented. To obtain this extension we apply the theory developed in the previous two chapters. The extension is conservative in the sense that the domain represented in logical form by the infinitary logic coincides with the domain represented in logical form by Abramsky's finitary logic. As a consequence we obtain soundness and completeness of the infinitary logic for the class of all finitary transition systems.

Interdependence of the chapters

The three parts of this monograph can be read almost independently. The logical interdependence between the chapters is schematically represented by the following diagram.



The sections between parentheses and the article [3] are only necessary as references to proofs.

Origins of the chapters

This monograph is a revison of author's Ph.D. thesis [33]. Several results presented here have already appeared in publications. Chapter 3 is mostly based on [37] and [40]. The second half of Chapter 4 appeared as an extended abstract in [42]. Chapter 6 is an extension of [38] and [39]. The first half of Chapter 7 is based on the paper [44] while the second half is new. Chapter 8 is a revised version of the paper [36]. Finally, Chapter 10 is based on [41]. Chapters 9 contain mostly original material, while Chapter 5 follows ideas of Smyth originally presented in [179] and [182].