Theory for Software Verification — draft January 20, 2009

DAVID A. NAUMANN
Stevens Institute of Technology

Semantic models are the basis for specification and verification of software. Operational, denotational, and axiomatic or algebraic methods offer complementary insights and reasoning techniques which are surveyed here. Unifying theories are needed to link models. Also considered are selected programming features for which new models are needed.

Categories and Subject Descriptors: F.3 [**Logics and Meanings of Programs**]:

General Terms: Verification, Theory, Semantics

1. INTRODUCTION

To verify a program means to produce compelling evidence, based on sound theory, that the program's behavior satisfies its specified requirements. Software verification requires mathematically rigorous reasoning about application domains, specifications of systems and their environments, software designs, code developed by programmers, code generated by tools, and the programs embodied in hardware. The theory of programming encompasses models of both the behaviors of programs and the properties described by their specifications. Algebraic and logical laws are derived from models to simplify reasoning and make it amenable to automated assistance.

The theory of programming holds a special fascination, for reasons that range from practical through scientific to philosophical. In compilers, semantic definitions are executed symbolically to analyze the behavior of code and thereby determine applicability of transformation laws used for practical optimization. Process algebras are used in the search for scientific understanding of emergent phenomena in distributed self-organizing systems. The goal of the Verified Software Initiative (VSI) is to produce programs that reason about programs, a branch of artificial intelligence that poses philosophical conundrums about self-reference [Moore 2008].

Many theories are needed and it is important to understand the reasons why. Accurate models of program execution on hardware are needed for reliable predictions about real systems. High level models of system concepts and environments are needed so people can judge whether requirements are adequately described by formal specifications. A range of programming languages, design and implementation techniques, and application domains need to be modeled. For the engineering

Naumann was supported in part by NSF grants CNS-0627338 and CNS-0708330.

of verification tools, even imprecise or incomplete models can be useful for speedy discovery of interesting conjectures.

Theory provides not only models but also links between models. A verifying compiler may check assertions in source code by efficient automated reasoning in a model at the programmer's level of abstraction —e.g., with a single unbounded address space. Yet the purpose of this check is to predict the behavior of the generated object code executed using hierarchical storage of bounded size. A link between these models can justify and delimit the sound conclusions about actual behavior that may be drawn using the more abstract model. Links connect models of system components that are implemented by different specialists using dissimilar programming languages and analysis tools.

This survey introduces the major semantic methods and accomplishments that appear to the author to be most relevant to software verification. It also considers selected programming language features and assertional methods which in the author's view are critical for a wide range of systems and which pose unsolved research problems. To save space, references for some topics are secondary sources, rather than seminal papers, and the distinction is sometimes left to the reader's discernment.

Overview. Throughout the paper we focus on two closely related reasoning tasks. The *correctness* task is to show that a system described in one notation (e.g., a programming or design language) has some property described in another (e.g., temporal logic). The *refinement* task is to show equivalence or some correctness-preserving refinement relation between two system descriptions in a common notation, for purposes such as refactoring a design model or reducing one analysis problem to another.

Sect. 2 surveys the operational approach in which program semantics is defined by modeling the internal workings of a machine. This includes idealized machines which are convenient for abstract reasoning and can be linked to many different concrete models.

The properties that we wish to verify usually do not include all the details of a system's execution. For example, we are often interested in the data that has been input and output, but less often interested in the exact timing or the order in which primitive instructions are executed. Sect. 3 surveys the denotational approach which models a program exclusively by the set of observations, direct or indirect, that are considered relevant at some chosen level of abstraction. Denotational semantics is compositional: The behaviors of a program are defined with reference to the behaviors of its subprograms —independent from the code or executions of those subprograms. Compositionality allows direct proofs of algebraic laws. It enables reasoning about a program module in the absence of other modules with which it interacts. To achieve compositionality, care is needed in the choice of what is considered to be observable; this in itself can yield insights about features of the language.

Sect. 4 surveys methods for linking between semantic models. In the older literature, the operational, denotational, and other modeling approaches sometimes appear as rivals. The current trend is towards combinations of many complementary models in order for verification to scale to large software systems.

if $prog[s(pc)]$ is ...	then s transitions to ...
$x := e$	$[s \mid x : s(e),\ pc : pc + 1]$
ifnz x goto i	$[s \mid pc : i]$ if $s(x) \neq 0$ $[s \mid pc : pc + 1]$ otherwise

Fig. 1. Transition table for simple instructions. We write $s(e)$ for the value of expression e in store s and $[s \mid x : n]$ for the store that agrees with s except for mapping x to integer n.

Operational and denotational semantics predict the specific observations that will result from execution of a program with a given set of input data and interacting with a given environment. But the purpose of software verification is to ensure that *all* observations of a program's behavior, in *all* initial conditions and in *all* environments, will have specified properties that reflect the system requirements. In Sect. 5, we introduce the *axiomatic* approach which models programs by their properties and directly supports proofs of correctness. This approach is immediately useful in the design of tools for software verification. Modeling at the level of properties leads in two startling directions. Refinement algebra embraces non-executable idealizations, in support of correctness by construction. Abstract interpretation enables fully automated verification by means of approximate models that are sufficiently accurate for the properties of interest.

The different styles a semantics are illustrated by application to three different kinds of programming language, imperative, functional, and distributed. Since the purpose is only to illustrate the semantic styles, each is treated only in its most basic form. Sect. 6 surveys some advanced features of commonly used design and implementation languages that pose difficult challenges for verification. For lack of space, the critical area of concurrency is almost entirely neglected. Sect. 7 concludes with a vision for the future.

2. TRANSITION SEMANTICS

A program instructs a machine how to behave. Turing [1937] sought to model behaviors with a simple purpose, the computation of some function, but capturing in a precise way the concrete resources and step by step action of physical devices. Such models are now known as *small-step operational semantics* or *transition semantics*. Sect. 2.1 gives typical examples, Sect. 2.2 considers observations and properties, and reasoning techniques are the topic of Sects. 2.3 and 2.4.

2.1 Transition systems

A *transition system* is a set S, elements of which are called states, together with an initial condition $I \subseteq S$, a transition relation $T \subseteq S \times S$, and a final condition $F \subseteq S$. A *run* or *computation* is a sequence of states $s_0, s_1, \ldots$ such that s_0 is in I, the successors are given by the transition relation (i.e., $(s_i, s_{i+1}) \in T$ for all i), and if the sequence is finite then the last state is in F.

For example, consider a register machine acting on some fixed set Var of integer variables. A state is a *store*, i.e., a valuation of the variables: S is $Var \to \mathbb{Z}$. The machine is running a fixed program, *prog*, which is a finite sequence of instructions numbered from 0. There is a designated *program counter* variable, *pc*; the initial condition is that *pc* is 0. The instructions include assignment $x := e$ where e is a simple arithmetic expression and $x \in Var$. The conditional instruc-

$$\langle \mathsf{skip}; c,\ s\rangle \mapsto \langle c,\ t\rangle \qquad \frac{\langle c0,\ s\rangle \mapsto \langle c1,\ t\rangle}{\langle c0; c,\ s\rangle \mapsto \langle c1; c,\ t\rangle} \qquad \frac{s(e) \neq 0}{\langle \mathsf{ifnz}\ e\ \mathsf{then}\ c0\ \mathsf{else}\ c1\ \mathsf{fi},\ s\rangle \mapsto \langle c0,\ s\rangle}$$

$$\frac{s(e) \neq 0}{\langle \mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od},\ s\rangle \mapsto \langle c; \mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od},\ s\rangle} \qquad \frac{s(e) = 0}{\langle \mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od},\ s\rangle \mapsto \langle \mathsf{skip},\ s\rangle}$$

$$\langle c \sqcap d,\ s\rangle \mapsto \langle c,\ s\rangle \qquad \langle c \sqcap d,\ s\rangle \mapsto \langle d,\ s\rangle \qquad \frac{\langle c,\ s\rangle \mapsto \langle c',\ s'\rangle}{\langle c \mathbin{|||} d,\ s\rangle \mapsto \langle c' \mathbin{|||} d,\ s'\rangle} \qquad \frac{\langle d,\ s\rangle \mapsto \langle d',\ s'\rangle}{\langle c \mathbin{|||} d,\ s\rangle \mapsto \langle c \mathbin{|||} d',\ s'\rangle}$$

Fig. 2. Selected rules in structural operational semantics. The horizontal line means "implies".

tion, ifnz x goto k, sets pc to k if x is non-zero; otherwise it sets pc to $pc + 1$ as does assignment. The final states are those where pc is out of the range of indices of *prog*. The transition relation is given in Fig. 1; it is defined only for non-final states. Many other features can be treated similarly, e.g., we could include *call* and *return* instructions and add to the state a stack of return addresses.

For static analysis, efficiency may be gained by using a transition system that models a single program, abstract protocol, etc. An alternative is to use a single transition system that models a stored program machine, e.g., by adapting the one above to use states of the form $\langle prog,\ s\rangle$.

2.1.1 *Structural operational semantics (SOS) and imperative programs.* Source languages offer structured programming constructs including the *simple imperative commands* given by the grammar $c\ ::= \mathsf{skip} \mid x := e \mid c; c \mid \mathsf{ifnz}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c\ \mathsf{fi} \mid \mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od}$. A "structural" style of transition semantics was introduced by Plotkin [2004] in lecture notes circa 1981. It eliminates the need for an explicit program counter. The cost is that the transition relation is defined inductively rather than being given by rules (as in Fig. 1) that can be directly executed in an implementation or simulation (e.g., model checking).

In our example, states take the form $\langle c,\ s\rangle$ where s is a store assigning integers to the variables of c. Command c represents the control state, initially the program of interest. The final states are those of the form $\langle \mathsf{skip},\ s\rangle$. The transition relation, $\mapsto$, is given by rules in Fig. 2. In the transitions for the conditional construct, the new control state is a sub-command of the initial command, but that is not the case for iteration. One of the rules for sequential composition has a proper antecedent, which is why the rule comprise an inductive definition of $\mapsto$.

The power and elegance of SOS is illustrated by its simple treatment of two advanced features of programming, non-determinism and concurrency. For example, look at the semantics of nondeterministic choice ($\sqcap$) and interleaving ($|||$) in Fig. 2.

2.1.2 *Parameterization and functional programs.* The *pure lambda terms* [Church 1936] are given by the grammar $M\ ::= x \mid MM \mid \lambda x.M$. Informally, $\lambda x.N$ is an anonymous function with parameter x and MN is the application of M to argument N. Application associates to the left: MNO is $(MN)O$. A transition semantics is given using states that consist of a single term. The key transition rule, $(\lambda x.M)N \mapsto (M/x{\rightarrow}N)$, is called *beta reduction.* (We use Reynolds' notation $M/x{\rightarrow}N$ for capture-avoiding substitution.) The most general transition semantics allows reduction in all subterms: if $M \mapsto M'$ then $MN \mapsto M'N$, $NM \mapsto NM'$, and $\lambda x.M \mapsto \lambda x.M'$. The nondeterminacy of this transition relation can be viewed as deliberate underspecification, just as many languages underspecify the order of evaluation for arithmetic operations. Even with the nondeterministic transition relation, all terminating computations lead to the same result, but some terms terminate or not depending on the order in which transitions are taken.

Lambda calculus guides the modeling of many forms of parameterization in programming and specification languages. As a very simple example, consider parameterized commands $p ::= (\lambda x.c)$ where parameter x is not allowed to be assigned in the procedure body c. Extend commands by $c ::= p(e)$ where e is an integer expression and extend the semantics of Sect. 2.1.1 by a single transition $\langle(\lambda x.c)(e),\ s\rangle \mapsto \langle c/x{\rightarrow}e,\ s\rangle$. To model pure functional languages, lambda terms can be augmented with primitives such as arithmetic and evaluation order made more deterministic. Moreover the transition relation is more determinate. To model *eager evaluation* (call by value), a subset of terms called values is designated: $V ::= \lambda x.M$. The transition rules are $((\lambda x.M)V) \mapsto (M/x{\rightarrow}V)$ and if $M \mapsto M'$ then $MN \mapsto M'N$ and $VM \mapsto VM'$. More generally, one can give a grammar of *evaluation contexts*, in which subterms can be reduced [Wright and Felleisen 1994], e.g., to model control constructs like throwing and catching exceptions. More concrete models use states with more than a single term, e.g., with an explicit stack [Landin 1964; Harper and Stone 1997]. A technique with many uses is *continuation passing style* which treat's a term's context as an extra parameter that can be manipulated (see Reynolds [1993] for a history).

2.1.3 *Concurrency.* The concurrent execution of two programs acting on shared state can be modeled by the union of their transition relations. More concretely, the nondeterministic scheduling of threads acting on shared store is easily modeled using command transitions; see the rules for $|||$ in Fig. 2. That semantics treats command steps as indivisible. The granularity of atomicity is critical, both for accurate modeling of hardware and for effective reasoning.

To concentrate on synchronization and communication, it is fruitful to use *process algebras* [Milner 1980; Brookes et al. 1984; Bergstra et al. 2001] with transition semantics in which states consist of nothing but a single process term. For example, consider the terms $P ::= \mathsf{in}\ ch.x; P \mid \mathsf{out}\ ch.v; P \mid (P \parallel P) \mid P \sqcap P$. Here $P \parallel Q$ executes the two processes concurrently, synchronizing by message-passing along named channels. The term $\mathsf{in}\ ch.x; P$ sets variable x to a value read on channel ch and $\mathsf{out}\ ch.v; P$ sends value v. This is made precise by the reduction

$$\mathsf{in}\ ch.x; P \parallel \mathsf{out}\ ch.v; Q \ \mapsto\ P/x{\rightarrow}v \parallel Q \tag{1}$$

Concurrency leads to nondeterminacy as in $\mathsf{out}\, ch.1 \parallel \mathsf{out}\, ch.2 \parallel \mathsf{in}\, ch.x$.

Unsynchronized concurrent steps are given by rules like those for $|||$ in Fig. 2. One of those symmetric rules can be omitted by using the *structural congruence* $P \parallel Q \equiv Q \parallel P$. Define $P \mapsto P'$ to hold just if there are Q and Q' such that $Q \mapsto Q'$ is an instance of an explicitly given rule and $P \equiv Q$ and $P' \equiv Q'$ [Milner 1992]. This amounts to defining transitions on equivalence classes of terms.

Equivalence classes help with scope and binding. Concrete implementations have complications that can be avoided in theory by identifying terms that are *alpha equivalent*, i.e., the same except for renaming of bound variables (cf. Sect. 4.4).

2.2 Observable behavior and program properties

A specification stipulates what aspects of computations are to be observed as well as which behaviors are required. Notions of observation are not limited to what is physically feasible, e.g., the abstract notion of termination is not really observable but is often required because exact time seldom matters. For transition semantics of commands, the usual forms of observation are: a finite prefix of the computation (called a *trace*); the initial and final states of a terminated trace; whether the computation is finite (termination); the sequence of values of designated variables (reactivity); the initial and final values of designated variables in a terminated trace.

Different computations may admit the same observations; abstracting to those observations is one purpose of the semantic methods discussed in Sects. 3–5.

A *labeled transition system* (LTS) augments a transition with an event that can be observed when the transition takes place. For example, out $ch.v; P \xrightarrow{ch!v} P$ and in $ch.x; P \xrightarrow{ch?v} (P/x{\to}v)$. Reduction (1) is replaced by rule $\dfrac{P \xrightarrow{ch!v} P' \qquad Q \xrightarrow{ch?v} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'}$. The distinguished label τ abstracts from an internal communication. A trace is now a sequence of events, or a sequence of non-τ events.

For unlabeled transitions, message transmission is modeled using a shared variable to record channel history. Real time can be modeled using a variable t increased appropriately by every transition. Lynch and Vaandrager [1996] use labeled transition systems in which some transitions are time steps. Alternatively, a computation can be taken as a function from $\mathbb{R}$ to states —but this goes beyond transition semantics. So too probabilistic computations [Kozen 1981; McIver and Morgan 2005] and models of true concurrency [Petri 1962; Mazurkiewicz 1986].

Let us turn from observations to properties of observations. These are often based on state conditions, e.g., whether the condition holds in the initial state, or whether it holds in every state of a trace (is *invariant*). As a property of traces, the *partial correctness* condition given by state predicates p, q is that if p holds initially and the trace is terminated then q holds in the final state. Two-state conditions can be applied to the first and last state, e.g., $x \geq 0 \wedge y' = x^2$ where y' refers to the value of y in the last state. Alternatively, this can be encoded by a pair of state conditions, $x \geq 0$ and $y = x^2$, together with a simple two-state condition (typically verified by simple syntactic checks) that requires x to be unchanged. Another use for two-state conditions is as *transition invariants* [Floyd 1967; Podelski and Rybalchenko 2004], e.g., whether a certain function of the state (called variant or measure) is decreased (in some well founded order) in every transition step. Pnueli [1977] pioneered the use of linear temporal logic, which combines state conditions with modal operators [Blackburn et al. 2001], to express conditions on runs such as whether every p-state is eventually followed by a q-state, while abstracting from specific steps in time.

Program properties are about the totality of a program's behaviors. Partial correctness, invariance, and other conditions on runs are usually lifted to programs *demonically*: every computation must satisfy the condition. This is by far the most common sort of requirement specification so it is worth rephrasing: The property is a set X of observations and the program is correct if all its behaviors are contained in X. This subsumes program refinement: let X be the behaviors of the program to be refined. Two classes of properties in this sense are important because they admit quite different proof techniques. A *safety property* says that nothing bad ever happens, where the "bad" thing is a state predicate. Partial correctness and invariance are safety properties. A *liveness property* says something "good" eventually happens, e.g., termination, or a response to every request. *Total correctness* conjoins partial correctness and termination. Every property (i.e., set of runs) is the intersection of a safety property and a liveness property [Lamport 1977; Alpern and Schneider 1985; Manolios and Trefler 2003]. Temporal logic provides a finer classification [Manna and Pnueli 1990; Sistla 1994]. Case distinctions can be avoided by using only infinite runs and modeling termination by quiescence [Manna and Pnueli 1990; Cousot and Cousot 2000]. Temporal logic can express *fairness* conditions, i.e., liveness assumptions about underlying schedulers or physical concurrency; alternatively, program properties quantify only over fair runs [Manna and Pnueli 1995].

Some interesting requirements cannot be specified as a set of observations interpreted as above. Complexity theory often treats nondeterminacy *angelically*: the question is whether from each initial state there exists a run satisfying some condition [Cook 1971]. More typical of a system requirement is the property that from every reachable state satisfying p there is a transition to one satisfying q and another to a state satisfying $\neg q$. This is expressible in branching-time logics [Emerson and Clarke 1982; Kozen 1983; Hennessy and Milner 1985] which include modalities that refer to all or some possible future paths. The property that nondeterminacy is bounded (i.e., no state has infinitely many immediate successors) is needed for certain proof techniques [Hennessy and Milner 1985; Abadi and Lamport 1988] but precludes some modeling techniques. Information confidentiality can be specified by requiring that for each run, possibly involving both high and low security actions, the subsequence consisting of only the low actions is also a run [Goguen and Meseguer 1982]. More subtle security properties can be expressed in terms of knowledge [Halpern and O'Neill 2008]. Motivated by information flow and other requirements like bounds on average response time, Clarkson and Schneider [2008] explore arbitrary conditions on run sets, dubbed *hyperproperties.*

Because it is conceptually simple and technically convenient to equate program properties with observation sets, theories provide special forms of observation. Some branching time behavior is captured by *failures* [Brookes et al. 1984] which consist of a trace together with some actions the process can refuse to make following that trace. Branching behavior of a process can also be observed by running it in parallel with a simple "test" process [DeNicola and Hennessy 1984; Milner and Sangiorgi 1992] A wealth of observations may be obtained by adding what are commonly called *ghost variables* or auxiliaries; these are explicitly assigned in program code or implicitly in the semantics. Ghost variables are used for selective exposure of details from lower abstraction layers (e.g., elapsed time), for abstraction (e.g., state predicates that refer to event history), and for expression of design concepts (e.g., access permissions and ownership). Care must be taken with the scope of ghost variables, to avoid unsoundness [Kleymann 1999].

Many other program properties can be defined on transition semantics, e.g., the number of activation records on a call stack. Abstracting from irrelevant artifacts of modeling is one goal of semantic theory.

2.3 Proof techniques and the inductive assertion method

To prove program properties using transition semantics, the basic technique is induction on states $s_0, s_1, \ldots$ of an arbitrary run. Consider the register machine program $x := 0; x := x + 1; \mathsf{ifnz}\ x\ \mathsf{goto}\ 1; x := -x$. The predicate $0 \leq pc \leq 2$ is a program invariant. Proof: The initial condition says that $pc = 0$ in the initial state, s_0. For the induction step, assume that $0 \leq pc \leq 2$ holds in s_{i-1}. Then one of the instructions is executed to obtain s_i; each step of computation sets pc to a value in the range 0..2.

For some trace properties, strong induction is needed. In every run of the code above, the variable y has the same value in every state. To prove for a given run that $\forall i \mid s_i(y) = s_0(y)$, show by induction on i that $\forall k \mid k \leq i \Rightarrow s_k(y) = s_0(y)$. The induction step is still about a single machine step.

For transition system (S, I, T, F), a state predicate $Q \subseteq S$ is called an *inductive invariant* if $I \subseteq Q$ and $\forall s, t \mid s \in Q \wedge (s, t) \in T \Rightarrow t \in Q$. (Using relation algebra: $Q; T \subseteq Q$.) Not all invariants are inductive: For the program above, the condition $pc \neq 0 \Rightarrow x \geq 0$ is invariant, but the unreachable state $[pc\!:\!3, x\!:\!1, y\!:\!0]$ satisfies the condition and steps to one that does not. If the set R of states reachable from I by iteration of T is known, then Q is invariant iff $R \subseteq Q$. Checking whether Q is an inductive invariant is usually easier than finding R. Ghost state is sometimes needed to formulate invariants that are sufficiently strong to be inductive [Owicki and Gries 1976].

2.3.1 *Type invariants.* The semantic definitions in Sect. 2.1.1 rely on the distinction between expressions and commands, interpreting the two kinds of phrase quite differently (as state functions and transition relations, respectively). Some but not all programming languages provide finer classification of phrases by means of typing, which can be exploited in semantic definitions (see Sects. 3.3 and 5). The type declaration $x : \mathsf{int}$ specifies an invariant, $x \in \mathbb{Z}$. Typing invariants are typically verified by type checking algorithms based on structurally recursive rules for *typing judgements* that record declarations and ascribe types to all phrases. Typing judgements need to carry enough information to serve as inductive invariants. This leads to higher order types, e.g., of code pointers in dispatching tables for interrupt handlers or virtual method calls in object-oriented programs.

Typing rules can be justified using a transition semantics for untyped phrases. To this end, one possibility is to introduce an error state, say $\lightning$, to which there is a transition if the code to be evaluated has an error like applying a function to the wrong number of arguments. A simpler alternative is for there to be no applicable transition —the state is called *stuck*, and only terminal states should be stuck. (See the textbook by Pierce [2002].)

2.3.2 *Inductive assertion method.* A partial correctness property can be proved by reducing it to an invariant. The technique was first articulated by Floyd [1967] (but see Jones [2003]). The method is suited to programs with a notion of *control point* such that in each state of a trace control is “at” one point. In the register machine this is explicit as variable pc. The structural operational semantics for simple imperative commands also has this notion, less explicitly. The technique can in principle be used for multi-threaded programs on a uni-processor, provided that scheduling is modeled explicitly.

Consider machine programs of length N in which every ifnz y goto i instruction has $0 \leq i \leq N-1$. Then only instruction $prog[N-1]$ can halt and only by changing pc to N; so the control points are $0..N$ with 0 the entry point and N the exit point. The key idea is to associate predicates with some control points. For practical application, "predicate" means formula, but much of the following makes sense for predicates as sets of states. A predicate associated with some control point is called an *assertion*. An *annotation* of a program is a mapping of some of its control points, including at least the entry and exit points, to predicates. (In the literature, "assertion" sometimes means state formula, and "partial correctness assertion" means the claim that a certain program satisfies a certain partial correctness property.)

For an annotation to be *correct* means that for any computation such that the initial state satisfies the initial assertion, the following is invariant: If control is at any point with an assertion, that assertion holds. To prove a partial correctness property $(pre, post)$, it suffices to find a correct annotation with pre and $post$ as the entry and exit assertions. The power of the method comes from using an annotation that is an inductive invariant, obtained as follows.

A set of control points is a *cutpoint set* if (a) it includes the start and exit points, and (b) every cycle in the control flow graph contains an element of the set. (The control flow graph has program points as vertices; there is a directed edge from i to j just if some transition can change pc from i to j. This notion becomes much less trivial for languages with higher order features.) A *basic path* is a finite sequence of control points, compatible with the program's control flow graph, starting and ending with cutpoints but containing no other cutpoints. Redefine *annotation* to require that the control points with assertions form a cutpoint set. (In terms of source code: every loop should have a specified invariant and every recursive procedure a specification.)

The instruction sequence $x := 1; z := y; z := z - 1; x := x * (y - z);$ ifnz z goto 2 satisfies the partial correctness property $(y \geq 1, x = y!)$. (To avoid distraction, say "$x = y!$" is false when $y < 0$.) To prove it, we use the annotation $0 : (y \geq 1)$, $5 : (x = y!)$, and $4 : (x = (y - z)! \wedge 0 \leq z \leq y)$. The basic paths are $0, 1, 2, 3, 4$ and $4, 2, 3, 4$ and $4, 5$.

An annotation is *inductive* if each basic path is *correct* in the following sense: For any state s that satisfies the starting assertion of the path and has $s(pc)$ at the start of the path, and any successive sequence of states that follows the path to the end, the end assertion holds in the last state. For the example above, the state $[x : 2, y : 3, z : 1, pc : 4]$ satisfies the assertion at 4 and the successive steps follow path $4, 2, 3, 4$ so the ending state must (and does) satisfy the assertion at 3. From state $[x : 6, y : 3, z : 0, pc : 4]$ the consecutive sequence does not follow path $4, 2, 3, 4$ but rather $4, 5$ so the ending state must (and does) satisfy the assertion at 5.

If an annotation is inductive then it is a correct annotation. Proof: by induction on length of the computation. Each step is along some basic path. If the step reaches an annotated point, it is the end step of the path and so the assertion holds by correctness of the path.

Closely associated with the inductive assertion method is a method for proving termination. One chooses a set W, a well founded relation on W, and a function f, called the *variant*, from states to W. For each basic path, and any execution along the path from a state s that satisfies the start assertion and goes to state t, $f(t)$ should be less than $f(s)$. More generally, one considers well foundedness of the transition relation itself [Podelski and Rybalchenko 2004].

2.3.3 *Verification conditions.* The inductive assertion method reduces partial correctness proof to the choice of an annotation together with correctness proofs for basic paths. These proofs involve the semantics of individual instructions, the accumulation of effects along the path, and mathematical reasoning about whatever predicates occur in assertions. Floyd [1967] showed that the mathematical reasoning can be factored out in the form of a single state predicate, the *verification condition*, that can be obtained —by an effective procedure— from the start and end predicates and the program. A basic path is correct iff its verification condition is valid.

The verification condition for a basic path can be constructed by "symbolic execution" of the path not on states but on formulas, which computes the strongest postcondition of the start assertion. The verification condition is that the strongest postcondition implies the end assertion. It suffices to use an approximation that is conservative, i.e., weaker than the strongest postcondition. An alternative is that the start assertion implies the weakest precondition for the path to establish the end assertion (and an approximation may strictly imply the weakest precondition). Floyd presented suitable formula transformations for simple imperative commands. Even more, he suggested that the verification conditions should be taken as specification of the programming language (see Sect. 5). Moore [2006] shows how verification conditions are generated directly from transition semantics.

2.4 Refinement, simulation, and contextual equivalence

Proofs of behavioral equivalence are often needed as basis for correctness, e.g., to justify program transformations used in verification condition generators and compilers. The correctness statement for a compiler amounts to an equivalence or refinement between a source program and an object program possibly interpreted in different models (see Sect. 4). Correctness by construction develops correct programs by refinement from specifications [Morgan 1988]. Refinement also connects concrete semantics with abstract interpretations on which static analyses are based.

Each notion of observation gives a notion of refinement: program p' refines p just if every observation of p' is also an observation of p. Two programs are equivalent iff each refines the other. For any class of program properties there is an associated notion of refinement: p' refines p iff every property of p is a property of p'.

To prove refinement directly in terms of transition semantics, the fundamental notion is simulation [Milner 1971]. A *simulation* from (S', I', T', F') to (S, I, T, F) is a relation $R \subseteq S \times S'$, such that initial states are related and moreover sRs' and $s'T't'$ imply there is some t with sTt and tRt'. In relation algebra: $R; T' \subseteq T; R$. Then T' *simulates* T if a simulation from T' to T exists. For deterministic transition systems, a simulation amounts to an inductive invariant on a cartesian product of the two systems. In *data refinement* [de Roever and Engelhardt 1998], a simulation

R is used to connect the differing internal states of two implementations and R is the identity on visible state.

Within a single transition system, state s' simulates s if there is a simulation R from the system to itself such that (s, s') is in R (here initial conditions are not relevant). This is of interest in case the program is part of the state. For an LTS, a *strong simulation* requires that all events match. A *weak simulation* does not require matching by τ-steps but rather allows an observable event to be matched by a step preceded and followed by several τ-steps.

A transition system is *bisimilar* to another if there is a simulation R and the converse of R is a simulation the other way around. This is a very fine equivalence relation: two systems are bisimilar iff they have the same branching-time temporal properties [Hennessy and Milner 1985].

For program phrases that on their own do not admit meaningful observations, the programmer's notion of equivalence is that they produce equivalent results when used in "any context". The parameterized command $(\lambda x.c)$ is not directly observable, but it can be indirectly observed in any command in which an application $(\lambda x.c)(n)$ occurs. Write $C[-]$ for some context with a "hole" into which p may be plugged to yield a well formed command $C[p]$. The *contextual approximation* relation, $\preceq_{ctx}$, is defined relative to some chosen notion of observation. As an example, if we observe pre-post states of commands then $p \preceq_{ctx} p'$ is defined to hold iff $\langle C[p],\, s\rangle \mapsto^* \langle \mathsf{skip},\, t\rangle \;\Rightarrow\; \langle C[p'],\, s\rangle \mapsto^* \langle \mathsf{skip},\, t\rangle$ for all $C[-]$ and all s, t. *Contextual equivalence*, $\simeq_{ctx}$, is the intersection of the preorder $\preceq_{ctx}$ with its converse.

2.5 Further reading

The textbook by Bradley and Manna [2007] gives a contemporary treatment of the inductive assertion method as well as axiomatic semantics. Winskel's [1993] textbook covers transition semantics as well as evaluation, denotational, and axiomatic semantics. Reynolds' [1998] textbook covers many of the topics of this survey and is our primary guide for terminology and notation. A slender book by Milner [1999] introduces the π-calculus, an influential process algebra that models mobility, and includes background on bisimulations.

3. DENOTATIONAL SEMANTICS

Specifications pertain to the totality of a program's behaviors. Denotational semantics Scott and Strachey [1971] connects a program with its set of behaviors, as such, rather than deriving the collection from a definition of its instances as in transition semantics. Moreover, the observable behaviors of a program are defined as a function of the behaviors of its constituent parts, i.e., *compositionally.* This enables reasoning about a program in terms of the behaviors of its subprograms, e.g., library procedures, without recourse to complete implementations of those components as is needed in operational semantics. To make compositionality possible, all phrases must denote something, and observations are organized in rich mathematical structures. Indeed, much of the interest is in what sorts of observation sets are considered and what operations are used to compose them.

A precedent of denotational semantics is Kleene's [1936] theory of partial recursive functions. This theory answers the question of what functions are computable not by defining some mechanism whereby functions are computed but by considering ordinary mathematical functions (the observation sets). The *partial recursive functions* are those that can be built up from arithmetic primitives using a small repertoire of operations: function composition, weak induction on the naturals (e.g., defining f via $f(x,i) = g(f(x,i-1))$ for given g), and unbounded minimization. The latter means defining $f(x)$ as the least y, $y \geq 0$, such that $g(x,y) = 0$ for given g; in there is no such y, $f(x)$ is not defined. Kleene's theory makes no reference to traces of an automaton or transition system, nor to syntax per se. Rather, it is about programs as ordinary mathematical objects.

This section presents denotational semantics in a conventional, syntax based way. Sect. 4.2 returns to the view that syntax can be dispensed with entirely.

Before delving into denotational semantics *per se*, we review some essential background on fixed points (*fixpoints* for short).

3.1 Fixed points, traces, and evaluation semantics

Recall how the theorems of a logic are defined inductively. The set Th of theorems is the smallest set that contains the axioms and is closed under the inference rules. To be more precise, for X a set of formulas let $F(X)$ be all instances of the axioms together with any formula that follows by an inference rule from some formulas in X. So Th is closed under F, i.e., $Th = F(Th)$. Moreover, it is the smallest set of formulas with this property. Function F is monotonic: if $X \subseteq Y$ then $F(X) \subseteq F(Y)$. The Knaster-Tarksi theorem [Tarski 1955] says that for a monotonic function F on sets, the least fixpoint of F is the intersection of all X such that $F(X) \subseteq X$. The Kleene fixpoint theorem says that the least fixpoint is the limit of the iterates of F, that is, the least upper bound (*lub*) of the ascending chain $\emptyset \subseteq F(\emptyset) \subseteq F^2(\emptyset) \subseteq F^3(\emptyset) \ldots$ where $F^2(X) = F(F(X))$ etc. In general, this chain must be extended to the transfinite, but under a suitable condition the fixpoint is reached as the lub of the finite iterates. In particular, for logical rules with finitely many antecedents we have $Th = \cup_{n \in \mathbb{N}} F^n(\emptyset)$. The condition is that F is *continuous*, i.e., distributes over the lub of any countable ascending chain: $F(\cup_i X_i) = \cup_i F(X_i)$ provided that $X_i \subseteq X_{i+1}$ for all $i \in \mathbb{N}$.

Let us revisit the computations of a system with states S and transition relation T, ignoring initial and final conditions. Let S^* be the set of finite sequences of states. For nonempty sequences $\overline{s}$ and $\overline{t}$, with $\overline{s}$ finite, write $\overline{s} \cdot \overline{t}$ for their matched catenation: if the last state of $\overline{s}$ is the first state of $\overline{t}$, then keep one copy of that state; otherwise, $\overline{s} \cdot \overline{t}$ is undefined. For sets X and Y of sequences, $X \cdot Y$ is the set of matched catenations. Let B be the set of blocked states, i.e., those without a T-successor. Define $F(X) = B \cup (T \cdot X)$, where T is considered as a set of two-element sequences and X ranges over subsets of S^*. The set of finite runs of (S,T) is the least fixpoint of F (least with respect to $\subseteq$) [Cousot and Cousot 1992].

The dual of least fixpoint is greatest fixpoint, which is the union of all X such that $X \subseteq F(X)$. In the case of traces, it happens that the least and greatest fixpoint coincide. In general, they are different. Let S^ω be the infinite sequences of states. The set of infinite runs of (S,T) is the greatest fixpoint of G defined by $G(Y) = T \cdot Y$ where Y ranges over subsets of S^ω. The reasoning principle for greatest fixpoints, *coinduction*, is closely related to simulation (see Sect. 3.4).

Before generalizing these ideas for use in denotational semantics, we apply them to a form of operational semantics that abstracts from intermediate steps. Whereas

$$\frac{\langle e,\ s\rangle \Downarrow v}{\langle x := e,\ s\rangle \Downarrow [s \mid x : v]} \qquad \frac{\langle c0,\ s\rangle \Downarrow s' \qquad \langle c1,\ s'\rangle \Downarrow t}{\langle c0; c1,\ s\rangle \Downarrow t} \qquad \frac{\langle e,\ s\rangle \Downarrow 0 \qquad \langle c1,\ s\rangle \Downarrow t}{\langle \mathsf{ifnz}\ e\ \mathsf{do}\ c0\ \mathsf{else}\ c1\ \mathsf{fi},\ s\rangle \Downarrow t}$$

$$\frac{\langle e,\ s\rangle \Downarrow 0}{\langle \mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od},\ s\rangle \Downarrow s} \qquad \frac{\langle e,\ s\rangle \Downarrow n \qquad n \neq 0 \qquad \langle c,\ s\rangle \Downarrow s' \qquad \langle \mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od},\ s'\rangle \Downarrow t}{\langle \mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od},\ s\rangle \Downarrow t}$$

Fig. 3. Selected rules of evaluation semantics for commands.

transition semantics defines the workings of a machine running a program, from which notions of observation are derived, an *evaluation semantics* directly defines the relation between program and observations [Kahn 1987]. Another term is *big-step operational semantics*, as this style has mostly been used for the most basic observation: the output value or final state from a given input or initial state.

For simple imperative commands, the rules in Fig. 3 define a ternary relation $\langle c,\ s\rangle \Downarrow s'$ on command c, initial store s, and final store s'. The definition can be read as a recursive interpreter that computes s' given c and s. For primitive commands like assignment, evaluation semantics is essentially the same as the transition semantics. But evaluation semantics combines complete observations, as evident in the single rule for sequence. The rules are read inductively, like proof rules: the relation $\Downarrow$ is a least fixed point. For an interpreter, the rule antecedents are recursive invocations, and the inductive reading gives the terminating executions of the interpreter.

Although the rules look like a definition by cases on syntax forms, they are not structurally inductive on the command, owing the second of the rules for loops. As in transition semantics, it is not possible to determine the behaviors of a command exclusively from the behaviors of its subparts.

For lambda terms, an evaluation semantics defines the relation $M \Downarrow V$ between a term and a value to which it reduces. The rules for eager evaluation are $V \Downarrow V$ and $\dfrac{M' \Downarrow V' \qquad (M/x{\to}V') \Downarrow V}{(\lambda x.M)M' \Downarrow V}$. The behavior $(\lambda x.M)M' \Downarrow V$ is not determined by the behaviors of M and M' as it will be in denotational semantics, but rather it requires performing textual substitution $M/x{\to}V'$ and evaluating the result.

Evaluation semantics can be used with other notions of behavior, e.g., commands can be related to their finite runs [Leroy and Grall 2008]. Abrupt termination, e.g., due to a type error, can also be modeled in evaluation semantics if error values are added to the possible observations. In transition semantics, error values and rules to manipulate them can be avoided because a finite trace ending in a stuck state is distinct from an infinite or normally terminated trace.

For observations of infinite computations, Cousot and Cousot [1992] propose evaluation semantics using greatest fixpoints. Leroy and Grall [2008] investigate coinductive evaluation semantics where a relation $\langle c,\ s\rangle \Uparrow$ represents divergence.

3.2 Denotational semantics and approximation

As a first taste of denotational semantics, we recall how the meaning of a first order formula is defined in terms of the meanings of its subformulas. The semantics is given by defining a relation $s \models F$ between formulas and the stores in which they are considered true. This can as well be written as a function $\mathcal{F}[\![-]\!]$ that maps a formula to its set of satisfying assignments. The semantics of first order logic is defined *compositionally*: by structural recursion on formulas, using in each case only the denotation of the subformulas, e.g., $\mathcal{F}[\![p \wedge q]\!]$ is defined to be $\mathcal{F}[\![p]\!] \cap \mathcal{F}[\![q]\!]$. For arithmetic terms we define $\mathcal{E}[\![e]\!](s)$, earlier written as $s(e)$, by structural recursion with clauses like $\mathcal{E}[\![x]\!](s) = s(x)$ and $\mathcal{E}[\![e + d]\!](s) = \mathcal{E}[\![e]\!](s) + \mathcal{E}[\![d]\!](s)$. Note that $+$ on the left side of the equation is program syntax and on the right it has its ordinary mathematical meaning. (Reminding the reader that they are different is one purpose of the fancy brackets.) One may view semantic functions like $\mathcal{E}[\![-]\!]$ as a *compositional translation* or homomorphism from one language to another whose meaning is known, e.g., ordinary mathematics or the logic of a theorem prover (Sect. 4.4). Owing to compositionality, equality of denotations is a congruence with respect to the constructs of the language. By contrast, bisimilarity and other operationally defined equivalences are not always congruences.

Let us consider a semantics that models initial/final observations of the terminating computations of commands. The semantic function $\mathcal{C}[\![-]\!]$ maps commands to relations on *Store*. The semantics of sequence illustrates how familiar mathematical constructions may be used in denotational definitions: define $\mathcal{C}[\![c0; c1]\!] = \mathcal{C}[\![c0]\!]; \mathcal{C}[\![c1]\!]$ where, on the right, ";" is composition of relations. For conditional, define $\mathcal{C}[\![\mathsf{ifnz}\ e\ \mathsf{then}\ c\ \mathsf{fi}]\!] = F(\mathcal{E}[\![e]\!], \mathcal{C}[\![c]\!])$ where F is defined as follows: $(s, t) \in F(g, R)$ iff either $g(s) \neq 0$ and $(s, t) \in R$ or $g(s) = 0$ and $s = t$.

Henceforth, we shall usually elide $\mathcal{E}$, $\mathcal{C}$, etc., as they can be inferred from their arguments. But they are different functions: $[\![c]\!]$ is an element of $\mathbb{P}(\mathit{Store} \times \mathit{Store})$ and $[\![e]\!]$ is an element of $\mathit{Store} \to \mathbb{Z}$.

The semantics of loops is given as a fixpoint. Operationally, $\mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od}$ has the same meaning as its unfolding, $\mathsf{if}\ e\ \mathsf{then}\ c; \mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od}\ \mathsf{fi}$. This suggests that the denotation will satisfy $[\![\mathsf{while}\ e\ \mathsf{do}\ c]\!] = F([\![e]\!], ([\![c]\!]; [\![\mathsf{while}\ e\ \mathsf{do}\ c]\!]))$ using F from semantics of conditional. Letting $G(R) = F([\![e]\!], ([\![c]\!]; R))$, we define $[\![\mathsf{while}\ e\ \mathsf{do}\ c]\!]$ to be the least fixpoint of G. It exists because G is monotonic with respect to the order $\subseteq$ on store relations. The examples in Sect. 3.1 might suggest the choice of least fixpoint, but a stronger justification is the link with transition semantics: $\langle c,\ s\rangle \mapsto^* t$ iff $(s, t) \in [\![c]\!]$ (see Sect. 4.1).

3.3 Domains: the ranges of denotational semantic functions

For a denotational model that captures divergence of commands, one might expect to use a greatest fixpoint; we consider such a model in Sect. 5.2. (Beware that in the literature one finds order relations reversed sometimes, whence greatest swapped with least.) For infinite behaviors a suitable form of observation is also needed, e.g., infinite traces or divergence (recall $\Uparrow$). The next example is a model for deterministic commands, capturing both finite and infinite behavior. Let $\mathit{Store}_\perp$ abbreviates $\mathit{Store} \cup \{\perp\}$, where $\perp$ is some object not in *Store* that stands for divergence. The semantics, $\mathcal{D}[\![-]\!]$, maps commands into the set $\mathit{Store} \to \mathit{Store}_\perp$. It is not ordered by $\subseteq$ but it does have the structure needed for fixpoints, the approximation order $\sqsubseteq$ is defined by $f \sqsubseteq g$ iff $f(s) \neq \perp \Rightarrow f(s) = g(s)$ for all $s \in \mathit{Store}$. (In general the *extensional* order on functions f, g into a set X partially ordered by $\leq$ is $f \sqsubseteq g \iff \forall v \in X \mid f(x) \leq g(x)$.) This function space also has a least element, the function $\lambda s.\perp$ (using lambda notation as meta-language). Every ascending chain $f_0, f_1, \ldots$ has a lub, namely the function g such that $g(s) = f_k(s)$ if there is some k such that $f_k(s) \neq \bot$; and $g(s) = \bot$ otherwise. Unlike the previous examples, this domain does not have lubs arbitrary collections: if, for some s, $f(s)$ differs from $g(s)$ and neither is $\bot$ then f and g have no upper bound. A partially ordered set with lubs of ascending chains is called a *complete partial order* (cpo) or *domain.*

The next ingredient for a general theory of domains is continuity. To define $\mathcal{D}[\![\mathsf{while}\ e\ \mathsf{do}\ c]\!]$ we define function G from $\mathit{Store} \to \mathit{Store}_\bot$ to $\mathit{Store} \to \mathit{Store}_\bot$, such that $G(f)(s) = s$ if $[\![e]\!](s) = 0$ and $G(f)(s) = f(\mathcal{D}[\![c]\!](s))$ otherwise. The iterates $\lambda s.\bot$, $G(\lambda s.\bot)$, $G^2(\lambda s.\bot)$, $\ldots$ form an ascending chain. For the lub of this chain to be a fixpoint of G we rely on the fact that G is continuous.

Denotations are given for every kind of phrase. As a simple illustration, we sketch a naive semantics for the simple procedures of the form $\lambda x.c$ with integer value parameters x (Sect. 2.1.2). We need a set of possible meanings to interpret the form $\lambda x.c$ and an interpretation of $(\lambda x.c)(e)$ as a function of the meanings of its parts. We define $\mathcal{P}[\![\lambda x.c]\!]$ as an element of $\mathbb{Z} \to (\mathit{Store} \to \mathit{Store}_\bot)$. (For parameters passed by name, a different domain is needed.) As usual, the semantic clause looks like a translation to the meta-language: $\mathcal{P}[\![\lambda x.c]\!] = \lambda i.\lambda s.[\![c]\!]([s \mid x\!:\!i])$. The semantics of application is $[\![(\lambda x.c)(e)]\!](s) = (\mathcal{P}[\![\lambda x.c]\!]([\![e]\!](s)))(s)$.

The next example is a language with more interesting types. The *simple types* are given by $\tau ::= \mathsf{int} \mid \tau{\to}\tau$ (using int as a representative *base type*). Simply typed lambda terms have the form $\lambda x\!:\!\tau.M$. Types are ascribed to terms by an inductively defined relation, $E \vdash M : \tau$ (called a *typing judgement*), where E assigns types to variables including the free variables of M. One of the typing rules is

$$\text{if} \quad E, x\!:\!\tau \vdash M : \tau' \quad \text{then} \quad E \vdash (\lambda x\!:\!\tau.M) : \tau{\to}\tau' \tag{2}$$

For example, closed term $(\lambda x\!:\!\mathsf{int}.\lambda f\!:\!\mathsf{int}{\to}\mathsf{int}.f(fx))$ has type $(\mathsf{int}{\to}(\mathsf{int}{\to}\mathsf{int})){\to}\mathsf{int}$. The denotational semantics begins with semantics for types: $\mathcal{T}[\![\mathsf{int}]\!] = \mathbb{Z}$ and inductively $\mathcal{T}[\![\tau{\to}\tau']\!] = \mathcal{T}[\![\tau]\!] \to \mathcal{T}[\![\tau']\!]$. The meaning of terms is defined by induction on syntax, so that for closed term $M : \tau$ the denotation $\mathcal{M}[\![M]\!]$ is an element of $\mathcal{T}[\![\tau]\!]$. More precisely, $\mathcal{M}[\![-]\!]$ is defined on typing judgements

For other type constructors there are corresponding constructs for domains. From D and D' we can form a disjoint union $D{+}D'$; this can be used to model exceptional versus normal outcomes. The product $D \times D'$, ordered pointwise, can be used to model functions with multiple arguments —but for parameters passed by value, pairs like $\langle v, \bot\rangle$ containing a single $\bot$ are of no use and removed in the *smash product* $D \otimes D'$. To add recursive definitions to simply typed lambda calculus we interpret the arrow types by $[\![\tau \to \tau']\!] = [\![\tau]\!] \to_c [\![\tau']\!]_\bot$, where $\to_c$ means the set of continuous functions (ordered extensionally). This function space has a least element $\lambda x.\bot$ on which to base approximation chains. A recursive definition denotes the least fixpoint of a function from $[\![\tau \to \tau']\!]$ to itself.

Nondeterministic commands can be taken to denote functions $D \to P(D)$ where the *powerdomain* $P(D)$ is some collection of subsets of D (typically finite, nonempty subsets). For subsets α, β of D there are several plausible ways to define $\alpha \leq \beta$, e.g., $\forall b \in \beta \mid \exists a \in \alpha \mid a \leq b$. Smyth [1983] links this with the demonic interpretation of nondeterminacy and total correctness, and the reverse condition with partial correctness. The two conditions together are appropriate when sets are values in their own right. Technical complications with powerdomains lead many to use relations or predicate transformers to model nondeterminacy (Sect. 5.2).

Not all domains can be defined by structural induction on types. Some popular languages are untyped and allow procedures to be applied to themselves. In the *extrinsic* view of types [Reynolds 1998], phrases have meanings independent of typing, contrary to the treatment above of typed terms. The canonical example is untyped lambda terms. Imagine a semantics so that $[\![M]\!]$ is an element of some semantic domain D to be defined. Now, $[\![\lambda x.M]\!]$ ought to be a function that can be applied to elements of D, so $[\![\lambda x.M]\!]$ is in $D \rightarrow_c D$, but $\lambda x.M$ is a term so it is in D. Scott [1970] found D with an isomorphism $D \cong (D \rightarrow_c D)$. The restriction to continuous functions is critical since a bijection $D \cong (D \rightarrow D)$ exists only if D is a singleton. Similar equations arise in other higher order languages (Sect. 6.3) and in languages that allow recursive definition of types (e.g., classes in C++).

As a hint about the solution of domain equations, consider the simpler, recursive equation $Str \cong (\mathbb{Z} + (\mathbb{Z} \times Str)_\bot)$ for non-empty "streams" of integers. Let Str_0 be the degenerate cpo $\{\bot\}$. Let $Str_{i+1} = (\mathbb{Z} + (\mathbb{Z} \times Str_i)_\bot)$ so in effect Str_k consists of lists of length at most k, some approximate in that they end with $\bot$. At the limit we get infinite sequences and their finite approximants. The chain of unfoldings is connected by order-embeddings Str_i into Str_{i+1} for each i. Such embeddings play a role like the ordering relation within a single cpo (see Sect. 4.3).

Domain theory encompasses the many variations on cpo's that support different constructions needed to model programming and specification language constructs. See Gunter and Scott [1990] and Abramsky and Jung [1994].

3.4 Fixed point induction and coinduction

A great benefit of denotational semantics is that many equations and refinement laws can be proved simply by use the ordering or equality of domain elements. More is needed to prove, e.g, partial correctness of a loop. Let G be the function defined in Sect. 3.2, so the semantics is the lub of the chain of iterates $G^i(\lambda s.\bot)$. A direct proof shows that each $G^i(\lambda s.\bot)$ has the property, by induction on i. Then show that the property is preserved at the limit, i.e., if every relation in an ascending chain has the property then so does the lub of the chain.

Explicit induction on i can be avoided. If $f : D \rightarrow_c D$ then by definition the least fixpoint, μf, satisfies an induction principle: For any $d \in D$, $f(d) \leq d$ implies $\mu f \leq d$. Scott's *fixpoint induction rule* is more powerful. It applies to any domain D and any $P \subseteq D$ that is *chain complete* (i.e., for any ascending chain whose elements are all in P, the lub is also in P). The rule says that $\mu f \in P$ follows from $\bot \in P$ and f-closure of P (i.e., $x \in P \Rightarrow f(x) \in P$).

Not all interesting predicates are chain complete. For example, let D be the domain of finite and infinite sequences, ordered by prefix. The finite sequences are not a chain-complete subset: the lub of an ascending chain of finite traces may be infinite.

For greatest fixpoints the dual principle is *coinduction*: if $d \leq f(d)$ then d is at most the greatest fixpoint of f. This was first described in application to simulations [Park 1981]. For transition system $(S, \mapsto)$, define an operator F on relations $R \subseteq S \times S$ as follows: $(s, t) \in F(R)$ iff $\forall s' \mid s \mapsto s' \Rightarrow \exists t' \mid t \mapsto t' \wedge (s', t') \in R$. The greatest fixpoint of F is the similarity relation and similarity of s, t can be

$$\begin{tikzcd}
RefCalc \arrow[d, "awp"] & \underline{AnnotSrc} \arrow[l, "gen"'] \arrow[r, "strip"] & SrcCode \arrow[r, "optim"] \arrow[ddll, "wp"'] \arrow[dd, "deno"'] \arrow[ddr, "eval"] & SrcCode \arrow[r, "comp"] \arrow[dd, "eval"] \arrow[ddr, "trans"] & \underline{ObjCode} \arrow[ddr, "trans'"] & \\
\underline{Frmla} \arrow[d, "\mathcal{F}"] & & & & & \\
Pred & & Beh_d \arrow[ll, "lift"'] \arrow[r, leftrightarrow] & Beh_e \arrow[r, leftrightarrow] & Beh \arrow[r, leftrightarrow, "R"] & Beh'
\end{tikzcd}$$

Fig. 4. Conceptual sketch of links comprising a verifying compiler. Verification conditions (VCs) are generated by translation to an intermediate language, $RefCalc$, to which formula-transformer semantics, awp, is applied. Links justify that if the VCs are provable (by a sound theorem prover, not shown) then the object code is correct.

proved by exhibiting some R such that $(s, t) \in R$ and $R \subseteq F(R)$. Coinduction is attractive because it is not restricted to chain-complete predicates. Gordon [1994] gives a succinct tutorial. Pitts [1996] applies coinduction to the difficult problem of reasoning about recursively defined domains.

4. MODULARITY AND LINKING

To make a case that specific software running on specific hardware behaves as required, one needs program semantics that unquestionably models the hardware and the code it executes —but also specification semantics that unquestionably models people's understanding of the specifications. The two must be linked but may be based on quite different semantic theories. Consider the Verisoft project, which aims to verify application programs in a dialect of C, with respect to a gate level hardware model. This involves links between software layers (compiler, OS layers, instruction set architecture) and in some cases multiple models per layer (e.g., big- and small-step semantics of the source language) [Alkassar et al. 2008]. In another scenario, application software is comprised of components designed and implemented in different languages, each of which may have its own model for stating and proving correctness. For modular verification of the integrated system, it must be proved that the components agree on the interpretation of their common interfaces.

The term *link* is used loosely in this survey. One simple example is a compilation function which maps source programs to object programs. Another kind of link is what we call a *coupling*: a relation between two state spaces or data representations. For example, a coupling could define how bit strings in input and output date correspond to abstract notions like "user identifiers" and "integers" in a requirements specification. A more interesting kind of link is the statement and proof of some property involving simple links. For example, suppose *optim* is a source to source translation that serves as one stage of an optimizing compiler. An interesting link is the statement and proof that the evaluation semantics, *eval*, is preserved by the translation. This is depicted by a triangle in Fig. 4. To its right is a triangle that depicts a link that says *eval* is consistent with transition semantics *trans*, modulo a coupling between the behaviors given by *eval* (Beh_e) and those given by *trans*. The figure shows a link between a denotational semantics (*deno*) and semantic weakest preconditions (*wp*) which is routine and given by a standard lifting to predicate transformers. Other links may involve difficult proofs and specialized techniques. Besides exact matches, some links may be approximate. For example, let the leftmost vertical arrow in the figure (*awp*) be a formula-transformer semantics composed with the semantics of formulas; for use in verification it may suffice for it to conservatively approximate the semantic weakest preconditions (*wp*). Sect. 4.1 describes some typical links, including the standard link between operational and denotational semantics.

For verified software, many different models and links are needed, so there is both engineering and scientific interest in modularity of models and their links. Ideally, adding a new construct to programming or specification language is achieved by modular extension of existing compilers and other tools. Such extensions are another kind of link. Sect. 4.2 considers ways in which operational and denotational methods can facilitate modularity in semantic modeling.

There is no widely accepted unifying framework for models and their links, nor indeed agreement on the need for such a framework. However, category theory (Sect. 4.3) provides unifying definitions which are widely used in semantic modeling, from the design of static analyses (Sect. 5.3) to construction of models for complex programming languages and logics. Semantic models and links are not only defined "on paper", with ordinary mathematics as the meta-theory, but also formalized within theorem provers in ways discussed in Sect. 4.4.

4.1 Linking operational and denotational

Figure 4 sketches a possible organization for proofs by a (simplified) verifying compiler. The input is a program with annotations that specify some properties to be checked. On the left is the flow for verification of whatever properties are expressed by the annotation. We return to this in Sect. 5. Rightward along the top, the annotations are discarded and the compiler does source level optimization followed by code generation. The downward arrows are semantic mappings, e.g., *trans* gives the behaviors of source code as defined by the transition semantics of Sect. 2.1.1 and *eval* gives them as defined in Sect. 3.1. Whereas the program's properties are verified using an axiomatic semantics (the formula-transformer *awp*), we wish to conclude that the properties are true of the actual machine behavior as modeled by its low level semantics $trans'$. The proof is decomposed via several links that we describe starting from the lowest level, rightmost in the figure.

We write σ for processor/memory states, $\rightarrowtail$ for its transition relation, and $trans'$ for the behaviors given by $\rightarrowtail$. Coupling R links abstract states with concrete ones: $\langle c,\, s\rangle R\sigma$ if the compilation of some command c_{main} is in certain locations in σ and the program counter register points to the beginning of the compilation of c, and moreover the valuation s corresponds to the contents of designated registers or memory locations in σ. For those variables used in the annotation and therefore considered observable, the correspondence with memory locations (mapped to I/O devices) is part of the system specification. Like any specification, we take it as given when considering proofs of correctness. Relation R also connects internal registers, using fixed bitwidth integers, with local variables, in whatever manner is needed for the link to be provable. The linking property says the following: If $\langle c,\, s\rangle R\sigma$ then $\langle c,\, s\rangle \mapsto \langle c',\, s'\rangle$ implies $\sigma \rightarrowtail^* \sigma'$ for some σ' such that $\langle c',\, s'\rangle R\sigma'$, and conversely $\sigma \rightarrowtail \sigma'$ implies $\langle c,\, s\rangle \mapsto^* \langle c',\, s'\rangle$ for some c', s' such that $\langle c',\, s'\rangle R\sigma'$. Use of reflexive, transitive closure of the transition relations caters for steps that do not match exactly (e.g., several instructions for a single assignment in source).

To justify abstract reasoning about correctness of a particular program, we only need the links for that one program. Typically, however, such links are established for a class of programs, e.g., a well behaved subset ("dialect") of the programming language that admits a simpler model of storage. Here we refrain from saying just what are the sets $ObjCode$, $AnnoProg$, etc.

The rightmost triangle in Fig. 4 links the abstract transition semantics with the evaluation semantics, which is more convenient for reasoning about optimizing transformations. Here there is no difference in the representation of state, so the desired relation is that both semantics give the same set of initial/final states (so Beh_e is Beh). The link is proved by mutual implication. We prove $\langle c,\ s\rangle \Downarrow s'$ implies $\langle c,\ s\rangle \mapsto^* \langle \mathsf{skip},\ s'\rangle$ by rule induction on the evaluation semantics, $\Downarrow$, using the lemma $\langle c,\ s\rangle \mapsto^n \langle \mathsf{skip},\ s'\rangle \ \wedge\ \langle c',\ s'\rangle \mapsto^* \langle \mathsf{skip},\ t\rangle \ \Rightarrow\ \langle c; c',\ s\rangle \mapsto^* \langle \mathsf{skip},\ t\rangle$ proved by induction on n. The proof that $\langle c,\ s\rangle \mapsto^* \langle \mathsf{skip},\ s'\rangle$ implies $\langle c,\ s\rangle \Downarrow s'$ goes by induction on c, using for the while case a nested induction on the length of $\langle c,\ s\rangle \mapsto^* \langle c',\ s'\rangle$ which in turn uses another lemma proved by induction.

The denotational semantics serves, in this particular scenario, as bridge to the axiomatic semantics used for verifying correctness of the annotated source program. For simple imperative commands, the link between evaluation and denotational semantics is simple: $\langle c,\ s\rangle \Downarrow t$ iff $(s, t) \in [\![c]\!]$. The implication left-to-right can be proved by rule induction on $\Downarrow$. The converse can be proved by structural induction on c, using an inner least-fixpoint induction in the case of while. (See Nipkow [1998] for these particular proofs.) The same notion of observation is used for both semantics and in Fig. 4 the link between Beh_d and Beh_e is the identity. For a more complicated language this would be a nontrivial relation between different sets.

In the case of a more complex programming language, there could be many more links of various kinds. It is possible, e.g., to use denotational semantics to validate source to source transformations. Or to directly link the axiomatic semantics, wp, with evaluation semantics. Ideally, a number of models and links are available, offering scientific insight as well as engineering options.

Recall the extension of simple imperative commands with parameterized commands and integer value arguments (Sect. 2.1.1). The transition semantics offers no direct observations of $(\lambda x.c)$ that could be linked to $\mathcal{P}[\![\lambda x.c]\!]$. This would be true as well in an evaluation semantics or transition semantics with explicit frame stack. The desired link is via contextual equivalence or approximation. *Adequacy* says the denotational semantics can be used to prove refinements (hence equivalences): $\mathcal{P}[\![p]\!] \sqsubseteq \mathcal{P}[\![p']\!] \Rightarrow p \preceq_{ctx} p'$. *Full abstraction* [Plotkin 1977] says that denotational equality is complete for reasoning about contextual equivalence: $p \preceq_{ctx} p' \iff \mathcal{P}[\![p]\!] \sqsubseteq \mathcal{P}[\![p']\!]$. A fully abstract semantics makes no distinctions that cannot be observed. For many complex languages full abstraction is difficult to achieve (see Sect. 6). In some case it can be achieved by *fiat*, using a quotient construction from an operational or other semantics: a term denotes the set of all observationally equivalent terms [Milner 1977]. But this offers little help for reasoning or understanding.

Just as domains can be defined inductively on type structure, so too can coupling relations be. The technique called *logical relations* [Plotkin 1973] can be used to prove contextual equivalences [Crary and Harper 2007] and also to link operational with denotational. For simplicity we explain its use to link two denotational semantics. The idea is that $\mathcal{T}[\![b]\!]$ might differ from $\mathcal{T}'[\![b]\!]$ if b is a base type (say int) but both $\mathcal{T}$ and $\mathcal{T}'$ are defined inductively like $[\![\tau \to \tau']\!] = [\![\tau]\!] \to [\![\tau']\!]$. Given a relation R_b for each base type b, the relation $R_{\tau\to\tau'} \subseteq [\![\tau \to \tau']\!] \times [\![\tau \to \tau']\!]$ is defined so that (f, f') is in $R_{\tau\to\tau'}$ just if $(f(v), f'(v')) \in R_{\tau'}$ for all $(v, v') \in R_\tau$.

4.2 Linking theories

Feng et al. [2008] demonstrate economy of reasoning in logics tailored to small sets of language features. Reasoning about the implementation of thread primitives like context switch involves higher types (code pointers in queued task blocks) but is sequential (because interrupts are disabled). The implementation of synchronization primitives relies on the thread interface, but nothing higher order; it does involve concurrency. Consistency between the two logics (and others) is a link expressed and proved in terms of a foundational logic which serves as a unifying theory, itself proved sound with respect to transition semantics of machine code. How can links be made directly and features modeled modularly?

The most powerful means to link different models is to focus on their signatures and laws. Algebraic laws can capture properties of individual constructs that remain true in combination with other constructs —e.g., associativity of sequential composition, or monoids extended to commutative groups. This is motivation for algebraic and logical semantic methods (Sect. 5) —but to justify the laws, we need other models.

In transition semantics, modular composition of features is relatively easy, e.g., by adding new instructions and transitions, perhaps with components added to the state. Fig. 2 shows how transition semantics of sequential commands lifts easily to transition semantics of multi-threaded programs (notation $|||$). Combining mechanisms, though, does not predict the properties of the combination. However, for the structural style (Sect. 2.1.1) there is a theory of rule formats that gives general results, e.g., when is bisimulation a congruence [Aceto et al. 2001] and thus suitable to justify algebraic laws. The abstract state machine (ASM) approach endows transition semantics with a high level of abstraction by taking states to be first order models. Stärk et al. [2001] use ASMs to give modular semantics of many features of Java source and JVM bytecode through a series of refinements.

For denotational semantics, addition of program constructs and observations often requires change to the underlying domains, with pervasive consequences. Indeed, the domains of a good denotational semantics manifest core features such as the difference between by-name, using $D \cong D \to_c D$, and lazy evaluation, using $D \cong (D \to_c D)_\perp$ in which $\perp$ is distinguished from the function $\lambda x.\perp$. Moggi [1991] shows that the notion of *monad* from category theory (Sect. 4.3) unifies various computational effects. A term or command denotes a function $State \to F(State)$ where F could be $\perp$-lifting, a powerdomain, etc.. To combine effects, say exceptions with nondeterminacy, F is composed from two simpler monads.

Hoare and He [1998] adapt Hehner's [1984] *predicative programming* with the intent to provide a unifying theory of programming (UTP) encompassing many features and links. This is a style of denotational semantics in which observations of all kinds are equated with valuations of designated *observation variables*, e.g., a sequence variable that records an observed trace. A specification is a predicate on observation variables. A program or design is a predicate as well, but one which satisfies some some "healthiness conditions", often expressed by closure conditions or algebraic laws. In this way, semantics of different constructs and notions of observation can be investigated without the clutter of syntax: a program just *is* a predicate, or if you like a set of observations. Economy of notation and definition is achieved through uniform use of predicates on observation variables of various types, and use of Galois connections for linking.

4.3 Category theory and Galois connections

A widely applicable unifying theory is category theory. Sometimes called the algebra of types, it offers not so much strong theorems as powerful definitions that organize and abstract from complex structures. A *category* consists of a set of objects and a set of arrows, where each arrow f has a source and target object, written $f : S \to T$. Moreover there is a partial composition operator, e.g., from $g : T \to U$ obtain $(f;g) : S \to U$. Composition is associative and there is an identity arrow for each object. There is a category Set whose objects are "all" sets, and arrows $S \to T$ are total functions from S to T, whence the notation. Another example is Cpo, whose objects are complete partial orders and arrows continuous functions. The objects of any preordered set form a category, with an arrow $s \to t$ (unique and therefore anonymous) just if $s \leq t$. A *functor* from one category to another maps objects and arrows to the same, preserving sources, targets, and composition. A functor between preordered sets is thus a monotonic function. (Pierce [1991] is a slim textbook on category theory.)

Functors serve to abstract from details of type constructions, for example disjoint sum. One representation of the set $S + T$ uses pairs $(0, s)$ and $(1, t)$ for $s \in S$ and $t \in T$. What matters is that there are injection functions $inl : S \hookrightarrow S + T$ and $inr : T \hookrightarrow S + T$. Moreover, functions from $S + T$ to some set U can be defined by cases: for any $f : S \to U$ and $g : T \to U$ there is a unique $[f, g] : S + T \to U$ such that $inl; [f, g] = f$ and $inr; [f, g] = g$. This gives a way to define $+$ on pairs of functions and the laws characterize $S + T$ up to isomorphism. This algebraic characterization works for Set, for Cpo, and for various other categories of domains.

One key notion of category theory is *adjunction*, involving a pair of functors back and forth between two categories. In the case of preordered sets, it is called a *Galois connection* and consists functions $f : (P, \leqslant) \to (Q, \lesssim)$ and $g : (Q, \lesssim) \to (P, \leqslant)$ with the property

$$f(p) \lesssim q \iff p \leqslant g(q) \text{ for all } p, q \tag{3}$$

Here f and g are approximate inverses, each mapping elements of one set to closest approximations in the other set. The lub property can be put in this form: $\sqcup(p, p') \lesssim q \iff (p, p') \leqslant \Delta(q)$ where P is $Q \times Q$ and $\Delta(q) = \langle q, q \rangle$. So too the axiom for disjoint sums, except that in a category of domains there may be many arrows (functions) from one object to another. Instead of equivalence (3) there is a bijection between arrows $h : S + T \to U$ and pairs $(f, g) : (S, T) \to \Delta(U)$.

Domain equations can be described in terms of functors. A solution D of an equation $X \cong F(X)$ in a category is the colimit of a chain of embeddings obtained from the inclusion $e : \mathsf{I} \hookrightarrow F(\mathsf{I})$ as $F(e) : F(\mathsf{I}) \hookrightarrow F^2(\mathsf{I})$ etc. [Smyth and Plotkin

1982]. Solutions are characterized by axioms similar to —indeed, generalizing— the above for disjoint sums and also generalizing least upper bounds in that there is an embedding $F^i \hookrightarrow D$ for each i.

Galois connections can be used to link different semantic domains, e.g., a model for partial correctness versus a model that considers divergence observable. For a given transition system there is a hierarchy of approximate semantic models (abstract interpretations Sect. 5.3) forming a chain of Galois connections that encompasses transition, denotational, and axiomatic semantics [Cousot 2002].

The *Curry-Howard isomorphism* is an adjunction that links types with proofs and underlies some higher order theorem provers [Bertot and Castéran 2004]. In propositional logic, if q' can be proved assuming p and q, written $p, q \vdash q'$, then also $p \vdash q \Rightarrow q'$, and vice versa. One can adapt the typing rule (2) to say that proof M of q' assuming p, q corresponds to a function taking proofs x of q to proofs of q'.

Cartesian products are dual to disjoint sums. For a pair f, g there is $\langle f, g\rangle : U \to S \times T$ such that $\langle f, g\rangle ; outl = f$ and $\langle f, g\rangle ; outr = g$. In the presence of effects — divergence, nondeterminacy, etc— these laws weaken to inequations but still hold as equalities for the effect-free arrows. Lax notions of adjunction makes possible the axiomatization of effects together with data structures [Martin et al. 1991; Power and Robinson 1997; Naumann 1998; Hyland et al. 2007].

4.4 On depth of embedding in machine checked theory

A *deep embedding* of a programming language in an automated theorem prover formalizes the syntax as data in the logic. A *shallow embedding* defines some or all of the object language in "semantic" terms only, i.e., as entities in the logic —cf. Kleene's theory, and UTP. A deep embedding of the simple imperative language represents the grammar of expressions, say $e ::= n \mid x \mid e + e$ as an inductive datatype and so too the grammar of commands. Then the semantic definitions map abstract syntax trees to semantic objects. In a shallow embedding, only the semantic objects are defined; an expression just *is* a state function. The translation of concrete syntax to such expressions is still needed in tools, but it is defined outside the logic and not subject to formal reasoning; in particular there is no syntax on which to do structural induction. A mixed embedding could treat commands as a data type, to support proofs by structural induction, while using a shallow embedding of expressions (e.g, the datatype constructor for assignment would apply to a variable name and a state function). Structural induction can be useful to prove laws that hold for programs but not for all elements of a model; on the other hand, its absence is impetus to find a better model. Nipkow [1998] discusses embeddings and uses a shallow one to streamline a proof rule for procedures [2002]. Shallow embedding raises the question of what full abstraction means, and it can happen that powerful features of the ambient logic render expected laws invalid.

Deep embedding for languages with binding constructs requires formalization of substitution and its properties. A theorem prover already provides such formalization for its logic. *Higher order abstract syntax* is a kind of shallow embedding that uses lambda abstraction in the metalanguage to interpret binding constructs [Harper et al. 1993]. Several other approaches are also under active investigation [Pitts 2003; 2006; Gordon and Melham 1996].

Another form of mixed embedding reconciles nondeterminacy with logics based on total functions. Nondeterministic transition systems are defined using a deterministic next-state function that takes as a parameter an oracle that resolves nondeterministic choices (e.g., by a scheduler or physical device). There is some loss of compositionality because the oracle is threaded through the semantic definitions. Moreover extensional equality of functions on oracles may fail to validate simple laws like $p \sqcap q = q \sqcap p$.

5. AXIOMATIC SEMANTICS

To establish that a program satisfies a specification, a reasoner uses not only specific details of the particular program and specification but also generalities about the semantics of programs and specifications. For example, if a command has the partial correctness properties (p, q) and (p, q') it also has $(p, q \wedge q')$. For a given programming and specification language, it may be possible to find a collection of sufficiently general rules to support a fixed factoring of the reasoning task: prove the rules once and for all and then use only the rules in reasoning about specific programs. Such a factoring amounts to a program logic. Whereas operational and denotational semantics relate a program to its behaviors, a formal program logic relates a program to formulas describing its properties. Floyd [1967] proposed to take verification conditions as the specification of the programming language's semantics, whence the standard term *axiomatic semantics*.

In some ways the term "axiomatic" is misleading. The idea of specifying a programming language by means of nothing more than a correctness logic is not widely accepted. Operational semantics is needed as a guide for implementation and for standardization of important details that will not be observable in models used for functional specification. (But it is possible to use operational semantics to describe unimplementable languages.) Rarely is there practical motivation to consider *all* models of an axiomatic semantics. Standard models are of interest because they reflect artifacts in actual use, and reasoners profit by moving freely between linked models. The author was tempted to coin the term "property-oriented" to describe a range of methods —formal logics, refinement algebras, and predicate transformers— all of which directly associate programs with properties and are applied directly in tools for proving correctness.

5.1 Hoare logic

The machine programs of Sect. 2.3.2 are assumed to be single-exit, but an individual ifnz instruction is usually not single-exit. The syntax of structured commands makes every sub-program single-exit, so any sub-program can be given a partial correctness specification. As a result, Hoare [1969] could adapt the inductive assertion method to a formal system based on *correctness judgements* (also called partial correctness assertions or Hoare triples) of the form $\{p\}\ c\ \{q\}$ where p and q are formulas. Using the denotational semantics of commands (Sect. 3.2), *validity* of a correctness judgement is defined by $\models \{p\}\ c\ \{q\}$ iff $pc([\![p]\!], [\![c]\!], [\![q]\!])$ where pc is defined by

$$pc(P, R, Q) \iff \forall s, t \mid s \in P \ \wedge\ (s, t) \in R \ \Rightarrow\ t \in Q \qquad (4)$$

In logic terminology, this is *truth* of the judgement, in some standard model to interpret the primitive predicates and functions in p, q, and c. One may also consider a notion of validity with respect to a class of interpetations, e.g., those

$$\frac{\{p\}\ c0\ \{R\} \qquad \{R\}\ c1\ \{q\}}{\{p\}\ c0; c1\ \{q\}}\ (\text{SEQ}) \qquad \frac{\{p \wedge e \neq 0\}\ c\ \{p\}}{\{p\}\ \mathsf{while}\ e\ \mathsf{do}\ c\ \{p \wedge e = 0\}}\ (\text{LOOP})$$

$$\frac{p \Rightarrow p' \qquad \{p'\}\ c\ \{q'\} \qquad q' \Rightarrow q}{\{p\}\ c\ \{q\}}\ (\text{CONSEQ}) \qquad \frac{\{p\}\ c\ \{q\} \qquad \{p'\}\ c\ \{q'\}}{\{p \wedge p'\}\ c\ \{q \wedge q'\}}\ (\text{CONJ})$$

Fig. 5. Selected rules of Hoare logic.

satisfying an axiomatization of arithmetic encompassing hardware implementations as well as the standard model of the integers.

Hoare logic differs from the inductive assertion method in that it does not explicitly use basic paths or annotation. It uses inference rules (Fig. 5) for deriving judgements about compound commands from judgements about their parts —and the specifications of those parts play the role of assertions at intermediate control points. Antecedents of some proof rules include first order validities as well as correctness judgements, so the proof system includes rules for first order reasoning over whatever data types are included in the language.

Besides the compositional rules associated with syntax, *structural rules* are needed for reasoning based on the meaning of specifications and correctness judgements. An example is the rule of consequence, which embodies refinement of specifications: p', q' is more refined because any command that satisfies it also satisfies p, q. Adaptation rules also embody refinement [Hoare 1971; Kleymann 1999].

The essential link is that if $\{p\}\ c\ \{q\}$ is derivable then $\models \{p\}\ c\ \{q\}$ —that is, the logic is *sound* with respect to denotational semantics (and thus, by other links, with respect to transition semantics). This is proved by induction on proof trees. For each proof rule, truth of the consequent follows from truth of the antecedents. The case of while goes by fixpoint induction, using that for any P, Q, the predicate $pc(P, -, Q)$ on state relations is chain complete.

For a formal system to be most useful —certainly for it to serve to specify the semantics of programs— we would like it to be *complete*: If a correctness assertion is valid there proof of it. This is impossible, owing to incompleteness of first order logic for the standard model of the integers. Cook's [1978] *relative completeness* says every valid correctness assertion can be proved, given an oracle for all first order validities. With a shallow embedding (Sect. 4.4) of assertions, the ambient logic serves as the oracle.

5.2 Weakest preconditions

In case c is a sequence of assignments, validity of $\{p\}\ c\ \{q\}$ is the same as correctness of a basic path where p is the start assertion and q the end assertion (Sect. 2.3.2). This brings us back to the notion of verification condition. Given a set X of states, the weakest semantic condition under which all terminating computations of c establish X, written $wp(c)(X)$, is the set defined by

$$s \in wp(c)(X) \iff \forall t \mid (s,t) \in [\![c]\!] \Rightarrow t \in X$$

(This is often written *wlp* and called "weakest liberal precondition", when considered together with the version for total correctness. Program logic for total correctness, and the links with semantics and weakest preconditions, are similar to what we discuss here.) By definitions, $\models \{p\}\ c\ \{q\}$ iff $[\![p]\!] \subseteq wp(c)([\![q]\!])$.

Imagine that for any c and q there is a formula $\mathsf{wp}(c)(q)$ that *expresses wp*, in the sense that $[\![\mathsf{wp}(c)(q)]\!] = wp(c)([\![q]\!])$. Then relative completeness could be shown as follows. Assume $\models \{p\}\ c\ \{q\}$. Thus $p \Rightarrow \mathsf{wp}(c)(q)$ is valid. Show a provability lemma saying that $\{\mathsf{wp}(c)(q)\}\ c\ \{q\}$ is derivable for all c, q. Use the rule of consequence to conclude that $\{p\}\ c\ \{q\}$ is provable.

How can the formula transformer wp be defined? For assignment it is straightforward (absent aliasing) to justify the clause $\mathsf{wp}(x := e)(q) = q/x{\rightarrow}e$. For sequence, the definition comes immediately from the semantic property $wp(c_0; c_1)(X) = wp(c_0)(wp(c_1)(X))$. (And the sequence case in the provability lemma goes by using $wp(c_1)$ to get the intermediate assertion.) For loops, $wp(\mathsf{while}\ e\ \mathsf{do}\ c)(X)$ can be expressed as a fixpoint of a monotonic function on state sets, based on $[\![e]\!]$ and $wp(c)$. These semantic properties serve as an alternate definition of wp by structural induction on commands —which amounts to a denotational semantics at the level of properties, depicted by the arrow labeled wp in Fig. 4. The link with denotational semantics (of Sect. 3.2) dictates that $wp(\mathsf{while}\ e\ \mathsf{do}\ c)(X)$ is the greatest fixpoint. (The least fixpoint is used for total correctness.)

To define $\mathsf{wp}(\mathsf{while}\ e\ \mathsf{do}\ c)(q)$, the fixpoint definition of $wp(\mathsf{while}\ e\ \mathsf{do}\ c)([\![q]\!])$ must be expressed as a formula. If the first order language is sufficiently rich this is possible and the assertion language is called *expressive*. For simple imperative commands, expressiveness is achieved using integers [Cook 1978; Winskel 1993] or finite sequences [Pierik and de Boer 2005] to encode traces. Infinitary [Back 1988] or higher order logic supports direct expression of the fixpoint. In Dynamic logic, $\mathsf{wp}(c)$ itself is added to the assertion language (see Sect. 5.4). A rich assertion language is at hand in the case of a shallow embedding of the assertion language in a higher order prover [Kleymann 1999; Nipkow 2002]. Even so, deductive reasoning about loops is impractical without given or inferred loop invariants.

For programs annotated with loop invariants (and procedure specifications), an annotated-wp function, awp, can be defined by recursion on program structure, such that $[\![\mathsf{awp}(c)(q)]\!] \subseteq wp(c)([\![q]\!])$ and hence $\models \{\mathsf{awp}(c)(q)\}\ c\ \{q\}$ [Floyd 1967]. It serves as a practical verification condition generator. In some tools, verification conditions are generated by first translating annotated source code to a collection of statements in an intermediate representation, shown leftmost in Fig. 4. This refinement calculus (Sect. 5.5) includes idealized commands that encode annotations, with semantics $\mathsf{wp}(\mathsf{assert}\ p)(q) = (p \wedge q)$ and $\mathsf{wp}(\mathsf{assume}\ p)(q) = (p \Rightarrow q)$. With this we can complete the story about basic paths and branch conditions: the example discussed Sect. 2.3.2 can now be written $\mathsf{assume}\ z \neq 0; z := z{-}1; x := x{*}(y{-}z)$. The formula transformer awp generates verification conditions that may be discharged by a theorem prover. Links in Fig 4 ensure that provability of the verification conditions implies correct behavior.

5.3 Abstract interpretation

Although $wp(c)$ applies to all *concrete predicates*, i.e., sets of states, it is those denotable by formulas that occur in specifications. It is possible to find restricted classes of formulae whose denotations are closed under wp. Still more, if the resulting collection of concrete predicates is finite then direct “execution” of the fixpoint definition of wp will converge. This effectively infers loop invariants, without recourse to annotation!

An *abstract interpretation* of a language is often given by a domain of abstract predicates and a predicate-transformer semantics using that domain [Cousot and Cousot 1977]. The theory uses Galois connections as links between abstract interpretations, to justify use of the abstract semantics to prove properties of the concrete semantics. For example, let $\gamma : (Frm, \Rrightarrow) \to (\mathbb{P}(State), \subseteq)$ be the "concretization function", previously written $[\![-]\!]$, that sends each formula in Frm to its satisfying states. If Frm is to provide an abstract interpretation, there should be a weak inverse $\alpha : (\mathbb{P}(State), \subseteq) \to (Frm, \Rrightarrow)$ that satisfies the condition (3) of Sect. 4.3, i.e., $\alpha(X) \Rrightarrow p \iff X \subseteq \gamma(p)$. Here $\Rrightarrow$ means valid-implication (which preorders the formulas in Frm). This situation together with results on "fixpoint transfer" ensures that certain properties of concrete semantics can be proved using abstract semantics. For theoretical purposes, abstract domains can be given as subsets of $\mathbb{P}(State)$ in which case γ is the identity and α a closure operator.

Viewing formulas as representatives of concrete predicates we are led to contemplate other representations. For example, a token $x : (i, j)$ could serve to express the set of states where $i \leq x < j$ and $x : (i, +\infty)$ to express $i \leq x$. This is one way to design efficient fixpoint-based analysis algorithms. Many useful abstractions have infinitely many predicates but admit "widening operators" that can be applied to leap up an approximation chain and reach fixpoints in finite time [Cousot and Cousot 1977]. (See this issue's model checking survey for much more.)

An abstract interpretation can be given by any domain D together with function $D \to D$, the least fixpoint of which is the semantics. Cousot [2002] uses this general theory to account for the links between an arbitrary transition system and its denotational and predicate transformer semantics, formalizing the consistent and complementary theories advocated by Hoare and Lauer [1974].

5.4 Other program logics

Hoare logic is convenient for theoretical study but in verification tools it is more convenient to work with programs in which some assertions (such as loop invariants) are included in the program syntax. A *proof outline logic* [Owicki and Gries 1976; Pierik and de Boer 2005] provides rules for manipulating correctly annotated programs that serve to directly specify generation of verification conditions.

A compositional logic for machine code must deal with the fact that a sequence of instructions may have multiple entries and exits. Tan and Appel [2006] develop a partial correctness logic where the correctness judgement has a precondition for each entry and a postcondition for each exit. The interpretation is in continuation style: if the code's exits are linked to code that does not err in states satisfying the associated postconditions, then the code does not err in states satisfying the preconditions. This is akin to an encoding of correctness used in some verifiers, which check whether $wp(\mathsf{assume}\, pre;\, c;\, \mathsf{assert}\, post)(true)$ equals $true$.

Hoare logic is an example of *endogenous* program logic, where the program is explicit in the judgements. Another example is Dynamic Logic [Pratt 1976; Harel et al. 2000], in which the formula $[c]p$ denotes the weakest (liberal) precondition for c to establish p. The modal $[c]$ has a dual $\langle c\rangle$ that says c has some computation that establishes p. In boolean combinations, dynamic logic formulas can express incorrectness and other properties beyond partial correctness. For command c on variables h, l, the formula $\forall l \exists r \forall h. [c](l = r)$ says that the "low security" variable l is not influenced by the initial value of h [Darvas et al. 2005]. Another alternative is to allow quantification and boolean connectives to be applied to correctness judgements, e.g., to express correctness of a command under hypotheses about correctness of procedures it calls [Reynolds 1982].

Temporal logics are *exogenous*, meaning that formulas are interpreted with respect to structures that include a fixed program [Pnueli 1977]. Linear time temporal logic formulas are interpreted on computations and feature modal operators to express "next state", "always", etc. Branching time formulas are interpreted on the tree of computations and combine linear operators with path quantification. Modalities can be indexed over observable events [Hennessy and Milner 1985; Stirling 1985]. The modalities can all be defined in terms of the next-state modality together with least and greatest fixpoint operators, e.g., "on all path, eventually p" is the least fixpoint of $\lambda q. p \vee AX\, q$ where AX is the "all next states" modality. See Kozen [1983] or Harel et al. [2000] for more on modal fixpoint calculus.

Secure information flow requirements cannot be specified simply as a set of runs (see Sect. 2.2), but it can be expressed using relations. Benton [2004] and Yang [2007] formalize program logics that capture such properties and also reasoning about program transformations and data refinement.

Many programming and specification languages rely on typing to express and enforce invariant conditions on single identifiers, e.g., $x \in \mathbb{Z}$ or "f is a function mapping each positive natural n to an array of length n". The examples might be written $x : \mathsf{int}$ and $f : (n : \mathsf{nat}) \to \mathit{Array}(n)$. Sound typing rules are often defined by structural recursion on program phrases, although *value-dependent* typing like that of f can be undecidable. Type systems are the subject of the textook by Pierce [2002], which includes pointers to the deep connections between dependent type systems and higher order logics. Verification conditions are combined with dependent typing by Nanevski et al. [2006], [2008].

Linear logic [Girard 1987] disentangles the structural rules that allow duplication or discarding of hypotheses (or variables, according to the Curry-Howard isomorphism, Sect. 4.3). This leads to type systems that encode single-threaded use of program state including the heap (Sect. 6.3).

5.5 Refinement algebra

Weakest precondition calculus and assume commands opens the door to other constructs that are not feasible to implement (cf. [Smyth 1983; Abramsky 1991]), e.g., havoc x sets x to an arbitrary value and is unboundedly nondeterministic. These commands serve to express specifications and to express correctness as refinement. Thus calculational proofs of correctness may be carried out in terms of the *refinement* ordering, $\sqsubseteq$, modeled by the pointwise ordering on predicate transformers which captures the intrinsic refinement ordering (Sect. 2.4) for partial or total correctness. Early development of the idea is due to Back [1978; Morris [1987; Morgan [1988]. See the textbooks by Back and von Wright [1998] and Morgan [1994]. A key advantage is that intermediate steps are designs which need not be restricted to efficiently implementable program constructs (cf. the use of imaginary numbers).

Predicate transformers ordered by $\sqsubseteq$ form a complete lattice and the least upper bound operator (not restricted to ascending chains) is needed to interpret some uses of ghost variables —which embody angelic nondeterminacy. Although predicate transformers are commonly used for semantics of refinement algebra, a generalization to transformers on other types is possible, using a sort of powerdomain construction that models the combination of angelic and demonic nondeterminacy [Morris and Tyrrell 2008].

Kleene algebra is a very expressive calculus of programming that subsumes Hoare logic [Kozen 1999; 2008] using conditional equations. Bloom and Ésik [1991] give a purely equational formalism that encompasses Hoare logic and fixpoints.

The treatment of refinement as reduction of nondeterminacy can be seen as a deliberate pun, confusing nondeterministic program behavior with underspecified behavior. Refinement of nondeterminacy can lead to violation of information flow properties. Abadi [1998] describes the issue in terms of fully abstract translation. Morgan [2006] uses ghost state to interpret assertions with a knowledge modality for which the intrinsic refinement order (Sect. 2.4) preserves ignorance. Joshi and Leino [2000] specify information flow by a semantic equation involving havoc.

6. SOME LANGUAGE FEATURES AND CHALLENGES

Many language features and feature combinations in common use pose challenges for modeling and reasoning. Some remain unsolved while others have been solved by elegant and practical theories. This section considers some of the areas most broadly relevant to VSI.

6.1 Functional programming

In case a sufficiently restrictive type system is used, it is possible to find tractable denotational models and to reason about functional programs using using extensional equality. Systems to be verified are seldom purely functional, but verification tools are often largely functional programs. The extreme case is functions defined within the logic of a theorem prover [Moore 2008], or extracted from its proofs via the Curry-Howard isomorphism [Bertot and Castéran 2004]. The categorical algebra of functions embeds in an algebra of relations that serve as specifications from which efficient programs are derived [Bird and de Moor 1996; Cousot 1999].

The cpo-based models (Sect. 3.2) for simply typed lambda calculus with integers and recursion [Scott 1993] turn out not to be fully abstract because at higher types there are "inherently parallel" functions in the model that are not denoted by any term [Plotkin 1977]. The search for a fully abstract model that captures the inherent sequentiality of functional programs led to *game semantics* where a term denotes a strategy for interacting with the term's environment [Abramsky et al. 1994; Hyland and Ong 2000]. It also led to refinements of domain theory that in turn led to linear logic [Girard 1989; Abadio and Curien 1998]. The search also led to development of logical relations (Sect. 4.1), including a generalization that uses a family of relations indexed by states of an abstract transition system. At arrow types one puts (f, f') in $R^s_{\tau\to\tau'}$, for state s, just if $(f(v), f'(v')) \in R^{s'}_{\tau'}$ for all future states s' and all $(v, v') \in R^{s'}_{\tau}$ [O'Hearn and Riecke 1995]. Quantifying over futures proves to be useful, e.g., to reason about states of dynamically allocated memory.

Sophisticated type systems including recursion at the type level have been studied mostly for functional programs but offer a theoretical basis for modular components in imperative and object-oriented programming. Data abstraction and hiding is expressed using $\forall$ and $\exists$ types. Consider a function *rev* reverses any type of list. Its type can be written $\forall\alpha.List(\alpha) \to List(\alpha)$ in the polymorphic lambda calculus. It is *parametric*, i.e., its behavior is uniform in the type variable α, i.e., it is a single algorithm. Consider a semantics in which *rev* denotes a family of functions $rev_\tau : List(\tau) \to List(\tau)$, indexed by types τ. (We elide semantic brackets, recall Sect. 4.4.) Uniformity of this family implies that *rev* commutes with any function on list elements. That is, for any $f : \tau \to \tau'$ we have $rev_\tau ; List(f) = List(f); rev_{\tau'}$. Here $List(f)$ maps f over lists; indeed, *List* distributes over function composition and is a functor (Sect. 4.3). A collection of arrows like *rev* that commutes with a functor is a *natural transformation.* To reason about uniformity, in particular for representation independence and contextual equivalences, naturality is generalized to a condition akin to logical relations [Reynolds 1984; 1998; Girard 1989] (but see also Sumii and Pierce [2005], Koutavas and Wand [2006]).

6.2 Shared mutable objects

For verification of imperative source code, syntactic restrictions are often used to entirely avoid the need to model aliasing of variable identifiers by means of separate mappings $Vars \to Locs$ and $Locs \to Vals$. (The well known distinction between L-value and R-value is an early contribution of theory.) Still, many languages include implicit or explicit references to heap-allocated objects. Several current verification systems for source languages like Java use arrays (functions) to model the heap, sometimes hidden by syntactic sugar in the assertion language. Variations date back to the 1970's and are reviewed in a paper where Bornat [2000] explores Burstall's ideas on expressing spatial separation. Straightforward use of array models can break full abstraction, in languages where references are opaque and programs cannot distinguish between isomorphic heap graphs. Storeless models provide unique representation of the heap in terms of access paths, at the cost of being more complicated than array representations [Deutsch 1992].

Reynolds introduced a *separating conjunction* by which separation is expressed by a logical connective, $*$, and with O'Hearn and Yang developed a logic in which locality of heap effects is manifest in specifications. A triple $\{p\}c\{q\}$ expresses that the *footprint*' of c, i.e., the part of the heap it acts on, is within the part that affects the truth of p. Owing to this *tight* interpretation of specifications, the logic supports a *frame rule* that, like Hoare's rule of Invariance, says that a predicate on separate state is preserved: if $\{p\}c\{q\}$ then $\{p * r\}c\{q * r\}$ —without need for side conditions about aliasing in the heap!

For data abstraction in imperative languages, where invariants on encapsulated data structures are not exposed in module interface specifications [Hoare 1972], separation is needed to express and enforce encapsulation for heap objects. One of many fruitful lines of current research in Separation Logic is data abstraction [O'Hearn et al. 2004; Mijajlović and Yang 2005; Birkedal and Yang 2007]. For reasoning using standard first order assertions, notions of ownership are made explicit [Leino and Müller 2004] and have been used to achieve relational parametricity in restricted forms [Banerjee and Naumann 2005].

6.3 Higher order imperative programming

Higher order procedures appear in a number of guises. Function pointers are commonly used in low level system software. Verification tools are implemented in impure functional languages like ML. Object-oriented programs use design patterns for higher order structure. Attempts to extend imperative programming to higher order encounter limits to the use of first order assertions [Clarke 1979] and it is difficult to find fully abstract models of the encapsulation provided by local variables and local procedures as parameters [Reynolds 1981; Meyer and Sieber 1988; O'Hearn and Reynolds 2000; Reynolds 1998; O'Hearn and Tennent 1997]. Some progress has been made using denotational models [Reus and Streicher 2005; Birkedal et al. 2009] but the domains needed to model commands that can be stored in variables involve mixed variance equations like $Cmd = ((Var \to (\mathbb{Z}{+}Cmd)) \to (Var \to (\mathbb{Z}{+}Cmd))$. These can be solved, but are unwieldy to use to justify proof rules in Hoare logics. Other recent work relies on more operational models [Honda et al. 2005]. Considerable progress has been made based on the model of Appel and McAllester [2001] in which types are interpreted denotationally based on a transition semantics for commands [Ahmed et al. 2009]. Instead of a chain of approximated domains in the category of cpos, the definitions use the number of future execution steps as an index of approximation (which, however, seems inherently tied to partial correctness). None of these models are able yet to prove all the expected contextual equivalences for programs using parameterized types and higher order mutable state.

6.4 Inheritance and object orientation

Inheritance gains its power from dynamically dispatched method calls, for which compositional models are higher order and thus involve the challenges mentioned above [Reus 2003] as well as issues for recursive types etc. [Pierce 2002]. A key modular reasoning technique, behavioral subtyping, is discussed in this issue's survey on specification languages. Design patterns are idioms for encapsulation and reentrant method invocations on mutable objects. Reasoning about inheritance and design patterns has been investigated using higher order separation logic [Bierman and Parkinson 2005] as well as first order assertions over ghost state [Leino and Müller 2004] and techniques from refinement calculus [Shaner et al. 2007].

7. CONCLUSION

For the basic constructs of first order imperative programming, for pure concurrent processes, and for pure functional programming including sophisticated types and module constructs, there are consistent and complementary semantics using operational, denotational and axiomatic methods to support implementation, specification, proof of correctness and refinement. However, the programming languages is current use combine subtle forms of concurrency and distribution, dynamically allocated mutable state, higher order structure, etc. Software systems are built using scripts that orchestrate the integration of components written in a variety of languages. Code is dynamically generated at multiple stages including run time. Requirements include security properties outside the scope of conventional specifications.

The literature is rich with information about interaction among some of these language features and numerous semantic models accomodate various sets of features. But many of these models are far from being implementable in verification tools and many necessary links are missing. Theorists may revel in the astonishing capabilities of current tools, achieved thanks to critical theories rendered (directly or not) in core algorithms. But the verification tools of the future will be built on models and unifying theories that remain to be discovered.

REFERENCES

Abadi, M. 1998. Protection in programming-language translations. In *25th International Colloquium on Automata, Languages and Programming.* LNCS, vol. 1443. 868–883.

Abadi, M. and Lamport, L. 1988. The existence of refinement mappings. In *Proceedings of LICS.*

Abadio, R. M. and Curien, P.-L. 1998. *Domains and Lambda-Calculi.* Cambridge U. Press.

Abramsky, S. 1991. Domain theory in logical form. *Annals of Pure and Applied Logic 51*, 1–77.

Abramsky, S., Jagadeesan, R., and Malacaria, P. 1994. Full abstraction for PCF. *Inf. Comput. 163*, 409–470.

Abramsky, S. and Jung, A. 1994. Domain theory. In *Handbook of Logic in Computer Science*, S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, Eds. Vol. 3. Clarendon Press, 1–168.

Aceto, L., Fokkink, W., and Verhoef, C. 2001. Structural operational semantics. In *Handbook of Process Algebra*, J. Bergstra, A. Ponse, and S. Smolka, Eds. Elsevier, 197–292.

Ahmed, A., Dreyer, D., and Rossberg, A. 2009. State-dependent representation independence. In *ACM Symp. on Prin. of Prog. Lang.*

Alkassar, E., Hillebrand, M. A., Leinenbach, D., Schirmer, N. W., and Starostin, A. 2008. The Verisoft approach to systems verification. In *VSTTE.* LNCS, vol. 5295. 209–224.

Alpern, B. and Schneider, F. B. 1985. Defining liveness. *Inf. Process. Lett. 21,* 4, 181–185.

Appel, A. W. and McAllester, D. 2001. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst. 23,* 5, 657–683.

Back, R. 1988. A calculus of refinements for program derivations. *Acta Inf. 25*, 593–624.

Back, R.-J. 1978. On the correctness of refinement steps in program development. Tech. Rep. A-1978-4, Department of Computer Science, University of Helsinki.

Back, R.-J. and von Wright, J. 1998. *Refinement Calculus: A Systematic Introduction.* Graduate Texts in Computer Science. Springer-Verlag.

Banerjee, A. and Naumann, D. A. 2005. Ownership confinement ensures representation independence for object-oriented programs. *J. ACM 52,* 6, 894–960.

Benton, N. 2004. Simple relational correctness proofs for static analyses and program transformations. In *ACM Symp. on Prin. of Prog. Lang.* 14–25.

Bergstra, J. A., Ponse, A., and Smolka, S. A. 2001. *Handbook of Process Algebra.* Elsevier.

Bertot, Y. and Castéran, P. 2004. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions.* Springer.

Bierman, G. and Parkinson, M. 2005. Separation logic and abstraction. In *ACM Symp. on Prin. of Prog. Lang.* 247–258.

Bird, R. and de Moor, O. 1996. *Algebra of Programming.* Prentice-Hall.

Birkedal, L., Stovring, K., and Thamsborg, J. 2009. Relational parametricity for references and recursive types. In *TLDI.*

Birkedal, L. and Yang, H. 2007. Relational parametricity and separation logic. In *Foundations of Software Science and Computational Structures.* LNCS, vol. 4423. 93–107.

Blackburn, P., de Rijke, M., and Venema, Y. 2001. *Modal Logic.* Cambridge U. Press.

Bloom, S. L. and Ésik, Z. 1991. Floyd-Hoare logic in iteration theories. *J. ACM 38,* 4, 887–934.

Bornat, R. 2000. Proving pointer programs in Hoare logic. In *Math. Prog. Construction.* LNCS, vol. 1837. 102–126.

Bradley, A. R. and Manna, Z. 2007. *The Calculus of Computation: Decision Procedures with Applications to Verification.* Springer-Verlag.

BROOKES, S. D., HOARE, C. A. R., AND ROSCOE, A. W. 1984. A theory of communicating sequential processes. *J. ACM 31,* 3, 560–599.

CHURCH, A. 1936. An unsolvable problem of elementary number theory. *American Journal of Mathematics 58*, 345–363.

CLARKE, E. 1979. Programming language constructs for which it is impossible to obtain good Hoare-like axioms. *J. ACM 26*, 129–147.

CLARKSON, M. R. AND SCHNEIDER, F. B. 2008. In *IEEE Computer Security Foundations Symposium.* 51–65. Long version submitted for publication, see http://hdl.handle.net/1813/11660.

COOK, S. 1971. The complexity of theorem proving procedures. In *Third Annual ACM Symposium on Theory of Computing.* 151–158.

COOK, S. A. 1978. Soundness and completeness of an axiom system for program verification. *SIAM J. Comput. 7,* 1, 70–90.

COUSOT, P. 1999. The calculational design of a generic abstract interpreter. In *Calculational System Design*, M. Broy and R. Steinbrüggen, Eds. NATO ASI Series F. IOS Press, Amsterdam.

COUSOT, P. 2002. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theo. Comp. Sci. 277,* 1–2, 47–103.

COUSOT, P. AND COUSOT, R. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symp. on Prin. of Prog. Lang.* 238–252.

COUSOT, P. AND COUSOT, R. 1992. Inductive definitions, semantics and abstract interpretation. In *ACM Symp. on Prin. of Prog. Lang.* 83–94.

COUSOT, P. AND COUSOT, R. 2000. Temporal abstract interpretation. In *ACM Symp. on Prin. of Prog. Lang.* 12–25.

CRARY, K. AND HARPER, R. 2007. Syntactic logical relations for polymorphic and recursive types. In *Computation, Meaning and Logic: Articles dedicated to Gordon Plotkin.* Electronic Notes in Theoretical Computer Science, vol. 172. 259–299.

DARVAS, A., HÄHNLE, R., AND SANDS, D. 2005. A theorem proving approach to analysis of secure information flow. In *Conf. on Security in Pervasive Computing.* LNCS, vol. 3450. 193–209.

DE ROEVER, W.-P. AND ENGELHARDT, K. 1998. *Data Refinement: Model-Oriented Proof Methods and their Comparison.* Cambridge U. Press.

DENICOLA, R. AND HENNESSY, M. 1984. Testing equivalences for processes. *Theo. Comp. Sci. 34*, 83–133.

DEUTSCH, A. 1992. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *IEEE Conference on Computer Languages.* 2–13.

EMERSON, E. A. AND CLARKE, E. M. 1982. Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comput. Program. 2,* 3, 241–266.

FENG, X., SHAO, Z., GUO, Y., AND DONG, Y. 2008. Combining domain-specific and foundational logics to verify complete software systems. In *VSTTE.* LNCS, vol. 5295. 54–69.

FLOYD, R. W. 1967. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, J. T. Schwartz, Ed. Symposia on Applied Math., vol. 19. Amer. Math. Soc., 19–32.

GIRARD, J.-Y. 1987. Linear logic. *Theor. Comput. Sci. 50*, 1–102.

GIRARD, J.-Y. 1989. *Proofs and Types.* Cambridge U. Press. With Paul Taylor and Yves Lafont.

GOGUEN, J. AND MESEGUER, J. 1982. Security policies and security models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy.* 11–20.

GORDON, A. D. 1994. A tutorial on co-induction and functional programming. In *Glasgow functional programming workshop.* 78–95.

GORDON, A. D. AND MELHAM, T. F. 1996. Five axioms of alpha-conversion. J. von Wright, J. Grundy, and J. Harrison, Eds. LNCS, vol. 1125. 173–190.

GUNTER, C. A. AND SCOTT, D. S. 1990. Semantic domains. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics.* 633–674.

HALPERN, J. Y. AND O'NEILL, K. R. 2008. Secrecy in multiagent systems. *ACM Trans. Inf. Syst. Secur. 12,* 1.

HAREL, D., KOZEN, D., AND TIURYN, J. 2000. *Dynamic Logic.* MIT Press.

Harper, R., Honsell, F., and Plotkin, G. 1993. A framework for defining logics. *J. ACM 40,* 1, 143–184.

Harper, R. and Stone, C. 1997. An interpretation of Standard ML in type theory. Tech. Rep. CMU–CS–97–147, Carnagie Mellon University, Pittsburgh, PA. June.

Hehner, E. C. R. 1984. Predicative programming part I. *Commun. ACM 27*, 134–143.

Hennessy, M. and Milner, R. 1985. Algebraic laws for nondeterminism and concurrency. *J. ACM 32,* 1, 137–161.

Hoare, C. and He, J. 1998. *Unifying Theories of Programming.* Prentice-Hall.

Hoare, C. A. R. 1969. An axiomatic basis for computer programming. *Commun. ACM 12*, 576–80, 583.

Hoare, C. A. R. 1971. Procedures and parameters: An axiomatic approach. In *Symposium on Semantics of Algorithmic Languages*, E. Engeler, Ed. Springer-Verlag.

Hoare, C. A. R. 1972. Proofs of correctness of data representations. *Acta Inf. 1*, 271–281.

Hoare, C. A. R. and Lauer, P. E. 1974. Consistent and complementary formal theories of the semantics of programming languages. *Acta Inf. 3*, 135–153.

Honda, K., Yoshida, N., and Berger, M. 2005. An observationally complete program logic for imperative higher-order frame rules. In *IEEE Symp. on Logic in Comp. Sci.* 260–279.

Hyland, J. M. E. and Ong, C.-H. L. 2000. On full abstraction for PCF. *Inf. and Comput. 163*, 285–408.

Hyland, M., Levy, P. B., Plotkin, G., and Power, J. 2007. *Theo. Comp. Sci. 375,* 1–3, 20–40. Festschrift for John C. Reynolds's 70th birthday.

Jones, C. B. 2003. The early search for tractable ways of reasoning about programs. *IEEE Annals of the History of Computing 25,* 2, 26–49.

Joshi, R. and Leino, K. R. M. 2000. A semantic approach to secure information flow. *Science of Computer Programming 37,* 1–3, 113–138.

Kahn, G. 1987. Natural semantics. In *Symp. Theo. Aspects Comput. Sci.*, F.-J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, Eds. 22–39.

Kleene, S. C. 1936. Lambda-definability and recursiveness. *Duke Mathematical Journal 2*, 340–353.

Kleymann, T. 1999. Hoare logic and auxiliary variables. *Formal Aspects of Computing 11*, 541–566.

Koutavas, V. and Wand, M. 2006. Small bisimulations for reasoning about higher-order imperative programs. In *ACM Symp. on Prin. of Prog. Lang.* 141–152.

Kozen, D. 1981. Semantics of probabilistic programs. *J. Comput. Syst. Sci. 22*, 328–350.

Kozen, D. 1983. Results on the propositional mu-calculus. *Theo. Comp. Sci. 27,* 3, 333–354.

Kozen, D. 1999. On Hoare logic and Kleene algebra with tests. *Trans. Comput. Logic 1*, 167–172.

Kozen, D. 2008. Nonlocal flow of control and Kleene algebra with tests. In *IEEE Symp. on Logic in Comp. Sci.* 105–117.

Lamport, L. 1977. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Engr. SE-3,* 2, 125–143.

Landin, P. J. 1964. The mechanical evaluation of expressions. *The Computer Journal 6,* 4, 308–320.

Leino, K. R. M. and Müller, P. 2004. Object invariants in dynamic contexts. In *European Conf. on Object-Oriented Programming.* 491–516.

Leroy, X. and Grall, H. 2008. Coinductive big-step operational semantics. *Inf. Comput.*. To appear.

Lynch, N. and Vaandrager, F. 1996. Forward and backward simulations part II: Timing-based systems. *Inf. Comput. 128,* 1, 1–25.

Manna, Z. and Pnueli, A. 1990. A hierarchy of temporal properties (invited paper, 1989). In *9th Annual ACM Symposium on Principles of Distributed Computing.* 377–410.

Manna, Z. and Pnueli, A. 1995. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag.

Manolios, P. and Trefler, R. 2003. A lattice-theoretic characterization of safety and liveness. In *22d Annual Symposium on Principles of Distributed Computing*. 325–333.

Martin, C., Hoare, C. A. R., and He, J. 1991. Pre-adjunctions in order enriched categories. *Mathematical Structures in Computer Science 1*, 141–158.

Mazurkiewicz, A. W. 1986. Trace theory. In *Advances in Petri Nets*, W. Brauer, W. Reisig, and G. Rozenberg, Eds. LNCS, vol. 255. 279–324.

McIver, A. and Morgan, C. 2005. *Abstraction, Refinement and Proof for Probabilistic Systems.* Springer-Verlag.

Meyer, A. R. and Sieber, K. 1988. Towards fully abstract semantics for local variables: Preliminary report. In *ACM Symp. on Prin. of Prog. Lang.* 191–203.

Mijajlović, I. and Yang, H. 2005. Data refinement with low-level pointer operations. In *Asian Symposium on Programming Languages and Systems*. 19–36.

Milner, R. 1971. An algebraic definition of simulation between programs. In *Proceedings of Second Intl. Joint Conf. on Artificial Intelligence*. 481–489.

Milner, R. 1977. Fully abstract models of typed *lambda*-calculi. *Theor. Comput. Sci. 4,* 1, 1–22.

Milner, R. 1980. *A Calculus of Communicating Systems.* LNCS, vol. 92. Springer-Verlag.

Milner, R. 1992. Functions as processes. *Math. Struct. Comput. Sci. 2,* 2, 119–141.

Milner, R. 1999. *Communicating and Mobile Systems: the π-Calculus.* Cambridge U. Press.

Milner, R. and Sangiorgi, D. 1992. Barbed bisimulation. In *Automata, Languages and Programming*. LNCS, vol. 623.

Moggi, E. 1991. Notions of computation and monads. *Inf. Comput. 93*, 55–92.

Moore, J. S. 2006. Inductive assertions and operational semantics. *International Journal on Software Tools for Technology Transfer 8,* 4–5, 359–371.

Moore, J. S. 2008. A mechanized program verifier. In *VSTTE*. LNCS, vol. 4171. Springer-Verlag, 268–276.

Morgan, C. 1988. The specification statement. *ACM Trans. Program. Lang. Syst. 10,* 3, 403–419.

Morgan, C. 1994. *Programming from Specifications, second edition.* Prentice Hall.

Morgan, C. 2006. The Shadow knows: Refinement of ignorance in sequential programs. In *Math. Prog. Construction*, T. Uustalu, Ed. LNCS, vol. 4014. 359–378.

Morris, J. M. 1987. A theoretical basis for stepwise refinement and the programming calculus. *Sci. Comput. Program. 9*, 287–306.

Morris, J. M. and Tyrrell, M. 2008. Dually nondeterministic functions. *ACM Trans. Program. Lang. Syst. 30,* 6, 1–34.

Nanevski, A., Morrisett, G., and Birkedal, L. 2006. Polymorphism and separation in Hoare type theory. In *ICFP*. 62–73.

Nanevski, A., Morrisett, G., Shinnar, A., Govereau, P., and Birkedal, L. 2008. Ynot: Dependent types for imperative programs. In *ICFP*. 229–240.

Naumann, D. A. 1998. A categorical model for higher order imperative programming. *Mathematical Structures in Computer Science 8,* 4, 351–399.

Nipkow, T. 1998. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Asp. Comput. 10,* 2, 171–186.

Nipkow, T. 2002. Hoare logics for recursive procedures and unbounded nondeterminism. In *Computer Science Logic*. LNCS, vol. 2471. 155–182.

O'Hearn, P., Yang, H., and Reynolds, J. 2004. Separation and information hiding. In *ACM Symp. on Prin. of Prog. Lang.* 268–280.

O'Hearn, P. W. and Reynolds, J. C. 2000. From algol to polymorphic linear lambda-calculus. *J. ACM 47,* 1, 167–223.

O'Hearn, P. W. and Riecke, J. G. 1995. Kripke logical relations and pcf. *Inf. Comput. 120*, 107–116.

O'Hearn, P. W. and Tennent, R. D., Eds. 1997. *Algol-Like Languages*. Vol. 1. Birkhauser, Boston.

Owicki, S. and Gries, D. 1976. An axiomatic proof technique for parallel programs I. *Acta Inf. 6*.

Park, D. 1981. Concurrency and automata on infinite sequences. In *Theoretical Computer Science*, P. Deussen, Ed. LNCS, vol. 104. 167–183.

Petri, C. A. 1962. Fundamentals of a theory of asynchronous information flow. In *IFIP Congress*. 386–390.

Pierce, B. C. 1991. *Basic Category Theory for Computer Scientists*. MIT Press.

Pierce, B. C. 2002. *Types and Programming Languages*. MIT Press.

Pierik, C. and de Boer, F. S. 2005. A proof outline logic for object-oriented programming. *Theo. Comp. Sci. 343,* 3, 413–442.

Pitts, A. M. 1996. Relational properties of domains. *Inf. Comput. 127*, 66–90.

Pitts, A. M. 2003. Nominal logic, a first order theory of names and binding. *Inf. Comput. 186*, 165–193.

Pitts, A. M. 2006. Alpha-structural recursion and induction. *Journal of the ACM 53*, 459–506.

Plotkin, G. 1973. Lambda definability and logical relations. Tech. Rep. SAI-RM-4, University of Edinburgh, School of Artificial Intelligence.

Plotkin, G. D. 1977. LCF considered as a programming language. *Theo. Comp. Sci. 5,* 3, 225–255.

Plotkin, G. D. 2004. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming 60–61*, 17–139.

Pnueli, A. 1977. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*. 46–57.

Podelski, A. and Rybalchenko, A. 2004. Transition invariants. In *IEEE Symp. on Logic in Comp. Sci.* 32–41.

Power, J. and Robinson, E. 1997. Premonoidal categories and notions of computation. *Mathematical Structures in Computer Science 7*, 453–468.

Pratt, V. R. 1976. Semantical consideration on Floyd-Hoare logic. In *17th Annual Symposium on Foundations of Computer Science*. 109–121.

Reus, B. 2003. Modular semantics and logics of classes. In *Computer Science Logic*, M. Baaz and J. A. Makowsky, Eds. LNCS, vol. 2803. 456–469.

Reus, B. and Streicher, T. 2005. About Hoare logics for higher-order store. In *Automata, Languages and Programming*. LNCS, vol. 3580. 1337–1348.

Reynolds, J. C. 1981. The essence of Algol. In *Algorithmic Languages*, J. W. de Bakker and J. C. van Vliet, Eds. North-Holland.

Reynolds, J. C. 1982. Idealized ALGOL and its specification logic. In *Tools and Notations for Program Construction: An advanced course*, D. Néel, Ed. Cambridge U. Press, 121–161. Reprinted in [O'Hearn and Tennent 1997].

Reynolds, J. C. 1984. Types, abstraction, and parametric polymorphism. In *Information Processing '83*, R. Mason, Ed. North-Holland, 513–523.

Reynolds, J. C. 1993. The discoveries of continuations. *Lisp and Symbolic Computation 6,* 3–4, 233–247.

Reynolds, J. C. 1998. *Theories of Programming Languages*. Cambridge U. Press.

Scott, D. and Strachey, C. 1971. Towards a mathematical semantics for computer languages. In *Proceedings of the Symposium on Computers and Automata*. Polytechnic Institute of Brooklyn.

Scott, D. S. 1970. Outline of a mathematical theory of computation. Technical Monograph PRG–2, Oxford University Computing Laboratory. November.

Scott, D. S. 1993. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theo. Comp. Sci. 121,* 1-2, 411–440. Manuscript circulated since 1969.

Shaner, S. M., Leavens, G. T., and Naumann, D. A. 2007. Modular verification of higher-order methods with mandatory calls specified by model programs. In *ACM Conf. on Object-Oriented Prog. Lang., Sys., and Applic.* 351–368.

Sistla, A. P. 1994. Safety, liveness and fairness in temporal logic. 495–511.

Smyth, M. and Plotkin, G. 1982. The category-theoretic solution of recursive domain equations. *SIAM J. Comput. 11,* 4, 761–783.

Smyth, M. B. 1983. Power domains and predicate transformers: A topological view. In *ICALP*. LNCS 154.

Stärk, R. F., Schmid, J., and Börger, E. 2001. *Java and the Java Virtual Machine: Definition, Verification, Validation.* Springer-Verlag.

Stirling, C. 1985. A proof-theoretic characterization of observational equivalence. *Theo. Comp. Sci. 39*, 27–45.

Sumii, E. and Pierce, B. C. 2005. A bisimulation for type abstraction and recursion. In *ACM Symp. on Prin. of Prog. Lang.* 63–74.

Tan, G. and Appel, A. W. 2006. A compositional logic for control flow. In *Conf. on Verification, Model Checking, and Abstract Interpretation.* LNCS, vol. 3855. 80–94.

Tarski, A. 1955. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics 5,* 2, 285–309.

Turing, A. M. 1937. On computable numbers, with an application to the entscheidungsproblem. *Proc. London Math. Soc. 2,* 42, 230–265. A correction appears in number 43, pages 544–6.

Winskel, G. 1993. *Formal Semantics of Programming Languages.* MIT Press.

Wright, A. K. and Felleisen, M. 1994. A syntactic approach to type soundness. *Inf. Comput. 115,* 1, 38–94.

Yang, H. 2007. Relational separation logic. *Theo. Comp. Sci. 375*, 308–334.

Received ...