Constructing the Views Framework

Stephan van Staden

University College London, United Kingdom
s.vanstaden@cs.ucl.ac.uk

Abstract. The views framework of Dinsdale-Young and others unifies several compositional techniques for reasoning about concurrent programs. This paper uses simple mathematics to construct the views framework incrementally from first principles. The result is a *model* for the views framework, which can also be understood as an *independent theory* of concurrent programs. Along the lines of "sequential programs are binary relations", the theory adopts the maxim "concurrent programs are formal languages". Consequently, programs obey familiar algebraic laws that can simplify reasoning; there is no need to postulate operational rules; the views program logic can be constructed in a stepwise fashion from more basic logics; program logic and operational thinking become largely decoupled; proving partial correctness becomes straightforward and it holds irrespectively of the specific choice of programming language constructs, operational rules, and the atomic actions that are implemented in a computer. All theorems have been formally checked with Isabelle/HOL. A proof script is available online.

Key words: semantics, formal languages, concurrency, programming calculi

1 Introduction

In a 2013 POPL pearl [1], Dinsdale-Young and others showed that their views framework can faithfully describe a wide variety of techniques for reasoning about concurrent programs. For example, concurrent separation logic, concurrent abstract predicates, type systems for recursive references and for unique pointers, a version of the rely-guarantee method, and even an adaptation of the Owicki-Gries method can all be seen as instances of the views framework. The fact that these seemingly different techniques fit into a common framework is truly remarkable. It suggests that it is possible to understand the essence of concurrent programming by studying a unifying formalism instead of a collection of complex and seemingly unrelated ones.

This paper is not concerned with instances of the views framework, but with its presentation and metatheory. The treatment in [1] follows what has become common practice in the programming language community. It regards programs as syntactic objects without intrinsic meaning. The 'meaning' of a program is given in terms of judgements that describe its behaviour, for instance its detailed operational behaviour and/or its summarised behaviour in the case of a program logic. Each judgement is defined by a set of syntax-directed rules that forms a calculus. Operational rules are usually left as postulates. Program logic judgements are shown to be partially correct with respect to an operational calculus, which means that they correctly summarise a program's behaviour according to the operational calculus. Establishing partial correctness requires

induction on derivations with the postulated program logic, and the reasoning hinges crucially on the syntactic structure of programs and the particular choice of operational rules. Consequently, the simplicity of the ideas that underpin a program logic can become masked by the details of syntax and the interactions of potentially large sets of detailed rules. In addition to lengthy proofs, the theorems themselves can be brittle and not as general as one might expect, because they are tied to the specifics of the chosen syntax and rules.

This paper explores the views framework by adopting a complementary viewpoint: programs are mathematical objects, and techniques for reasoning about their execution and correctness follow basic principles that are independent of syntax and particular rules. This yields an alternative account of the framework, where it is constructed stepwise from the ground up. In formal terms, this paper provides a mathematical model for the entire views framework. The model can also be understood as an independent theory of concurrent programs and reasoning. Its development is incremental and should be easy to follow. Prior knowledge of the views framework is thus not necessary to appreciate the material.

If programs are mathematical and not syntactic objects, then what kind of mathematical object is appropriate for modeling them? Firstly, one needs a domain that is rich enough to describe all the basic programming constants and operators, and to validate the expected interactions between them. Secondly, it must be as simple as possible. Lastly, it is preferable to reuse mature concepts from standard mathematics that are widely known and easy to apply. Along the lines of "sequential programs are binary relations", this work adopts the maxim "concurrent programs are formal languages". Formal languages have rich algebraic structure that the reasoning can exploit. They elegantly model concurrent composition as the familiar language interleaving (or shuffle) operator. Moreover, formal languages are simple and almost universally known among computer scientists. It is remarkable that ordinary formal languages with finite words can form a basis for concurrent imperative programming in such a straightforward manner. These familiar mathematical objects are not limited to simple computational mechanisms such as finite state automata, but can also be used to understand Algol-like languages with complex behaviour that includes non-terminating executions.

When programs are mathematical objects, one can of course use ordinary mathematics to define interesting judgements about them. All definitions in this paper are succinct and can be explained without reference to rules or other calculi. The rules that conventionally define the judgements are instead proved as theorems. The proofs are all fairly simple, and it is likewise straightforward to prove relationships between judgements such as partial correctness. These theorems are robust with respect to the programming language syntax and the choice of rules.

Since there is no grammar that defines programs as syntactic objects, no definition or proof can use induction on the syntax of a program. It is consequently easy to accommodate new programming constructs that can be modeled as formal languages. In contrast to their treatment in the mainstream approach, such extensions can never invalidate existing results.

The definitions of the judgements give quite a bit of freedom with respect to the eventual choice of rules. As long as a rule is a theorem, one can adopt it without in-

validating any property. In particular, there is no need to change the proof of partial correctness. This, together with the fact that the judgements can be explained independently, serve to decouple deductive (i.e. program logic) and operational thinking.

In summary, this paper complements [1] with a semantic theory that:

1. Comes with a rich algebra for program equivalence and refinement.
2. Deals independently with the program logic and the operational aspects.
3. Justifies the reasoning principles that these calculi embody by proving that their rules are theorems.
4. Establishes partial correctness without reference to rules, language constructs, or the atomic actions that a computer might perform.
5. Is extensible with respect to programming and specification constructs, reasoning rules, and calculi.
6. Presents an integrated account of the denotational, algebraic, operational, and program logic aspects of concurrent programs.
7. Uses simple mathematics and straightforward proofs.

The whole development has been formalised and checked in Isabelle/HOL. A proof script is available online [2].

Outline Section 2 summarises the definitions, operators and algebraic laws of languages. Section 3 outlines abstract calculi that follow from the algebraic laws. Section 4 covers concepts that are common to the deductive calculi in Section 5 and the operational calculi in Section 6. The partial correctness of the deductive calculi with respect to the operational calculi follows in Section 7, and Section 8 concludes.

2 Languages and laws

Formal languages offer a simple and expressive formalism for modeling concurrent programs. This section summarises the definitions, operators and algebraic laws of formal languages that are used in the rest of the paper.

An *alphabet* is a set. A *word* over an alphabet A is a finite sequence of elements from A. A set of words over an alphabet form a *language*. The languages over an alphabet A, being subsets of the set of all words over A, form a complete Boolean algebra. The intended alphabet will always be clear from the context, and is omitted when the results hold for an arbitrary alphabet. There are several lattice-theoretic constants and operators, for example:

- $\bot$ is the empty language $\{\}$. $\top$ is the language consisting of all words.
- $\bigcup X$ is the least upper bound of the set of languages X, and $(\cup)$ its binary variant.
- $\bigcap X$ is the greatest lower bound of languages in X, and $(\cap)$ its binary variant.

Languages have several other operators that are also familiar. In particular:

- *skip* is the language consisting only of the empty word: $\mathit{skip} \stackrel{\text{def}}{=} \{[]\}$
- $(;)$ is the language concatenation operator: $P\,;Q \stackrel{\text{def}}{=} \{p +\!\!+ q \mid p \in P \ \wedge \ q \in Q\}$, where $p +\!\!+ q$ denotes the concatenation of words p and q.

- The Kleene star operator concatenates its argument zero or more times:
  $P^* \stackrel{\text{def}}{=} \bigcup\{P^n \mid n \in \mathbb{N}\}$, where $P^0 \stackrel{\text{def}}{=} \mathit{skip}$ $P^{k+1} \stackrel{\text{def}}{=} P\,;P^k$
- ($\|$) is the language interleaving (or shuffle) operator:
  $P \parallel Q \stackrel{\text{def}}{=} \bigcup\{p \otimes q \mid p \in P \ \wedge \ q \in Q\}$
  Here $p \otimes q$ denotes the set of all interleavings of the words p and q:
  $[] \otimes q \stackrel{\text{def}}{=} \{q\}$ $p \otimes [] \stackrel{\text{def}}{=} \{p\}$
  $(e : p) \otimes (e' : q) \stackrel{\text{def}}{=} \{e : r \mid r \in p \otimes (e' : q)\} \cup \{e' : r \mid r \in (e : p) \otimes q\}$

Languages have a rich algebra that can often simplify formal reasoning. Some basic properties of the operators appear in Table 1.

	$\cup$	$\cap$	;	$\|$
Commutative	yes	yes	no	yes
Associative	yes	yes	yes	yes
Idempotent	yes	yes	no	no
Unit	$\bot$	$\top$	*skip*	*skip*
Zero	$\top$	$\bot$	$\bot$	$\bot$

Table 1. Basic properties of the operators.

Since $(\mathbb{P}(\top), \cup, ;, ^*, \bot, \mathit{skip})$ is a Kleene algebra [3], all the usual laws and identities hold as theorems. For example, the Kleene star is monotone and satisfies the following laws:

- $\mathit{skip} \cup (P\,;P^*) \subseteq P^*$
- $P \cup (Q\,;R) \subseteq R \ \Rightarrow \ Q^*\,;P \subseteq R$
- $\mathit{skip} \cup (P^*\,;P) \subseteq P^*$
- $P \cup (R\,;Q) \subseteq R \ \Rightarrow \ P\,;Q^* \subseteq R$

All the binary operators distribute through ($\cup$) and are consequently monotone. A stronger statement is true for $\circ \in \{\cap, ;, \|\}$:

- $P \circ (\bigcup X) = \bigcup\{P \circ Q \mid Q \in X\}$
- $(\bigcup X) \circ P = \bigcup\{Q \circ P \mid Q \in X\}$

The same holds for $\circ = \cup$ when X is not empty. In fact it is known that $(\mathbb{P}(\top), \cup, \bot, \|, ;, \mathit{skip})$ is a concurrent Kleene algebra [4], and hence (;) and ($\|$) interact as follows (the properties also appear in [5] as Proposition 5.3 and Corollary 5.4):

- $(P \parallel Q)\,;(R \parallel S) \subseteq (P\,;R) \parallel (Q\,;S)$
- $P\,;(Q \parallel R) \subseteq (P\,;Q) \parallel R$
- $(P \parallel Q)\,;R \subseteq P \parallel (Q\,;R)$
- $P\,;Q \subseteq P \parallel Q$

In summary, the languages with interleaving satisfy all the laws of programming mentioned in [6,7,8].

3 Abstract calculi

The laws yield abstract versions of several familiar calculi of programming, as discussed in [4,6,7,8]. Later sections of the paper will instantiate selected rules of the abstract calculi to obtain concrete calculi. By making the abstract calculi explicit, one can often see immediately why a concrete rule is a theorem. Although it is not the primary goal of this paper, the connection between the abstract and concrete calculi can help to understand the relationship between the results in [4,6,7,8] and their counterparts in more conventional work.

3.1 Abstract Hoare logic

The abstract Hoare triple is defined as follows:

$$P \langle Q \rangle R \stackrel{\text{def}}{=} P ; Q \subseteq R$$

Rules of abstract Hoare logic follow as theorems from the algebra:

(HAskip) $P \langle \mathit{skip} \rangle P$

(HAseq) $P \langle Q \rangle R \ \wedge \ R \langle Q' \rangle S \ \Rightarrow \ P \langle Q ; Q' \rangle S$

(HAchoice) $(\forall Q \in X : P \langle Q \rangle R) \ \Rightarrow \ P \langle \bigcup X \rangle R$

(HAiter) $P \langle Q \rangle P \ \Rightarrow \ P \langle Q^* \rangle P$

(HAcons) $P' \subseteq P \ \wedge \ P \langle Q \rangle R \ \wedge \ R \subseteq R' \ \Rightarrow \ P' \langle Q \rangle R'$

(HAdisj) $(\forall P \in X : P \langle Q \rangle R) \ \Rightarrow \ \bigcup X \langle Q \rangle R$

3.2 Abstract Plotkin calculus

The basic judgement of the abstract small-step operational calculus of Plotkin is defined in terms of a set of languages called $Actions$. The only requirement on $Actions$ is that it includes *skip*.

$$\langle P, s \rangle \longrightarrow \langle P', s' \rangle \stackrel{\text{def}}{=} \exists Q \in Actions : P \supseteq Q ; P' \ \wedge \ s ; Q \supseteq s'$$

Several rules hold as theorems:

(PAaction) $P \in Actions \ \wedge \ s' \subseteq s ; P \ \Rightarrow \ \langle P, s \rangle \longrightarrow \langle \mathit{skip}, s' \rangle$

(PAseq1) $\langle \mathit{skip} ; P, s \rangle \longrightarrow \langle P, s \rangle$

(PAseq2) $\langle P, s \rangle \longrightarrow \langle R, s' \rangle \ \Rightarrow \ \langle P ; P', s \rangle \longrightarrow \langle R ; P', s' \rangle$

(PAchoice) $P \in X \ \Rightarrow \ \langle \bigcup X, s \rangle \longrightarrow \langle P, s \rangle$

(PAiter1) $\langle P^*, s \rangle \longrightarrow \langle \mathit{skip}, s \rangle$

(PAiter2) $\langle P^*, s \rangle \longrightarrow \langle P ; P^*, s \rangle$

(PAconc1) $\langle \mathit{skip} \parallel P, s \rangle \longrightarrow \langle P, s \rangle$

(PAconc2) $\langle P \parallel \mathit{skip}, s \rangle \longrightarrow \langle P, s \rangle$

(PAconc3) $\langle P, s \rangle \longrightarrow \langle R, s' \rangle \ \Rightarrow \ \langle P \parallel P', s \rangle \longrightarrow \langle R \parallel P', s' \rangle$

(PAconc4) $\langle P, s \rangle \longrightarrow \langle R, s' \rangle \ \Rightarrow \ \langle P' \parallel P, s \rangle \longrightarrow \langle P' \parallel R, s' \rangle$

3.3 Milner calculus

While the Plotkin calculus hides actions, the Milner calculus makes them explicit:

$$P \xrightarrow{Q} R \stackrel{\text{def}}{=} Q \in \mathit{Actions} \ \wedge \ P \supseteq Q\,;R$$

The Milner rules describe small execution steps:

(Maction) $\quad P \in \mathit{Actions} \ \Rightarrow \ P \xrightarrow{P} \mathit{skip}$

(Mseq1) $\quad \mathit{skip}\,;P \xrightarrow{\mathit{skip}} P$

(Mseq2) $\quad P \xrightarrow{Q} R \ \Rightarrow \ P\,;P' \xrightarrow{Q} R\,;P'$

(Mchoice) $\quad P \in X \ \Rightarrow \ \bigcup X \xrightarrow{\mathit{skip}} P$

(Miter1) $\quad P^* \xrightarrow{\mathit{skip}} \mathit{skip}$

(Miter2) $\quad P^* \xrightarrow{\mathit{skip}} P\,;P^*$

(Mconc1) $\quad \mathit{skip} \parallel P \xrightarrow{\mathit{skip}} P$

(Mconc2) $\quad P \parallel \mathit{skip} \xrightarrow{\mathit{skip}} P$

(Mconc3) $\quad P \xrightarrow{Q} R \ \Rightarrow \ P \parallel P' \xrightarrow{Q} R \parallel P'$

(Mconc4) $\quad P \xrightarrow{Q} R \ \Rightarrow \ P' \parallel P \xrightarrow{Q} P' \parallel R$

3.4 Abstract Kahn calculus

The abstract Kahn calculus is a big-step operational calculus that executes programs to completion. Consequently, its judgement does not mention actions:

$$\langle P,\, s\rangle \longrightarrow s' \stackrel{\text{def}}{=} s\,;P \supseteq s'$$

The abstract Kahn rules that are theorems include:

(KAskip) $\quad \langle \mathit{skip},\, s\rangle \longrightarrow s$

(KAseq) $\quad \langle P,\, s\rangle \longrightarrow s' \ \wedge \ \langle P',\, s'\rangle \longrightarrow s'' \ \Rightarrow \ \langle P\,;P',\, s\rangle \longrightarrow s''$

(KAchoice) $\quad P \in X \ \wedge \ \langle P,\, s\rangle \longrightarrow s' \ \Rightarrow \ \langle \bigcup X,\, s\rangle \longrightarrow s'$

(KAiter1) $\quad \langle P^*,\, s\rangle \longrightarrow s$

(KAiter2) $\quad \langle P,\, s\rangle \longrightarrow s' \ \wedge \ \langle P^*,\, s'\rangle \longrightarrow s'' \ \Rightarrow \ \langle P^*,\, s\rangle \longrightarrow s''$

(KAconc1) $\quad \langle P,\, s\rangle \longrightarrow s' \ \wedge \ \langle P',\, s'\rangle \longrightarrow s'' \ \Rightarrow \ \langle P \parallel P',\, s\rangle \longrightarrow s''$

(KAconc2) $\quad \langle P',\, s\rangle \longrightarrow s' \ \wedge \ \langle P,\, s'\rangle \longrightarrow s'' \ \Rightarrow \ \langle P \parallel P',\, s\rangle \longrightarrow s''$

4 States, traces, descriptions and atoms

Since imperative programs manipulate state, the theory is parametric in a set of computational states Σ. For example, an instantiation of the theory might choose Σ to be the set of all functions mapping variables into values. Let σ, possibly with decorations, range over Σ.

The bulk of the paper uses $\Sigma \times \Sigma$ as the language alphabet. That is, the individual elements of a word are pairs of states. Such a word is called a *trace*. A set of traces, i.e. a language over $\Sigma \times \Sigma$, is called a *description*.

A description whose traces all have length one is called an *atom*. The atoms are isomorphic to the binary relations on states, which makes them suitable for modeling the state-transformation behaviour of the (possibly nondeterministic) primitive operations of a programming language. Let $Atoms$ be the set of all atoms, ranged over by a. The next definitions abbreviate the behaviour of an atom:

$$a(\sigma) \stackrel{\text{def}}{=} \{\sigma' \mid [(\sigma, \sigma')] \in a\}$$
$$a(S) \stackrel{\text{def}}{=} \bigcup\{a(\sigma) \mid \sigma \in S\}$$

Given a set of atoms, one can construct more complicated programs by using operators such as (;) and ($\|$). This gives rise to complex traces in descriptions. However, since the theory does not demand that a description was obtained in this way, it will be easy to introduce new constructions without violating existing results.

The use of sets of sequences of state pairs in compositional models of concurrency dates back to Park [9] and is also prominent in the work of Brookes on transition traces [10]. These models include infinite sequences of state pairs to mirror the infinite executions that arise from a small-step operational calculus. In contrast to this, the model of the current paper works on a deeper level. Its traces do not originate from an operational calculus. Instead, its finite traces form the foundation of small-step calculi (see Section 6) that in turn can yield infinite executions. This novel arrangement has interesting semantic consequences, for instance:

- It is unnecessary to postulate an operational calculus.
- The domain is much simpler without infinite traces.
- The deductive and operational theories are largely decoupled; everything is based on elementary traces.
- Partial correctness theorems say that the expected relationships with small-step calculi nevertheless hold (see Section 7).
- Regarding loops and recursion: while Park and Brookes must use greatest fixpoints with "strange" properties [9] to gain infinite traces, we can use least fixpoints.

It is also intellectually satisfying that ordinary formal languages with finite words, which are widely known by computer scientists, can underpin widely known programming languages and sophisticated behaviour such as infinite executions. The later sections will flesh out the details of this relationship.

A trace t is internally consistent, written $ic(t)$, when the states between adjacent pairs are equal:

$$
\begin{aligned}
&ic([]) \stackrel{\text{def}}{=} True \\
&ic([(\sigma, \sigma')]) \stackrel{\text{def}}{=} True \\
&ic((\sigma, \sigma') : (\sigma'', \sigma''') : t) \stackrel{\text{def}}{=} \sigma' = \sigma'' \ \wedge \ ic((\sigma'', \sigma''') : t)
\end{aligned}
$$

In the absence of interference from the environment, a program's execution must transform the state in an internally consistent way. In other words, the state it transforms in the next step must be the state it produced in the previous step. A trace that is not internally consistent represents behaviour that will never be observed in isolation, but that may be activated by appropriate interference from a concurrent program. This 'dormant' behaviour makes concurrent programming especially challenging, and it is important to record such information.

Descriptions are not only useful for modeling programs. The following functions yield the set of all internally consistent traces that end in (one of) the given state(s):

$$
\begin{aligned}
&\mathit{ic\text{-}traces\text{-}ending\text{-}in}(\sigma) \stackrel{\text{def}}{=} \{t \mid ic(t) \ \wedge \ \exists t', \sigma' : t = t' +\!\!+ [(\sigma', \sigma)]\} \\
&\mathit{ic\text{-}traces\text{-}ending\text{-}in}(S) \stackrel{\text{def}}{=} \bigcup \{\mathit{ic\text{-}traces\text{-}ending\text{-}in}(\sigma) \mid \sigma \in S\}
\end{aligned}
$$

For every state σ, the set $\mathit{ic\text{-}traces\text{-}ending\text{-}in}(\sigma)$ is non-empty, since for example $[(\sigma, \sigma)]$ will always be a member. Note that $\mathit{ic\text{-}traces\text{-}ending\text{-}in}(S)$ describes all consistent traces that establish the characteristic predicate of S.

Let $\mathit{Inconsistent}$ be the set of all traces that are not internally consistent. An inconsistency in a trace cannot be undone later on: $\mathit{Inconsistent} ; P \subseteq \mathit{Inconsistent}$.

The material so far is sufficient for studying several well-known calculi of programming. The choice to discuss deductive calculi before operational calculi is arbitrary, as the two approaches can be understood independently of each other.

5 Deductive calculi

The purpose of a deductive calculus, or program logic, is to facilitate reasoning about the correctness of programs. This section studies Hoare logic and three views calculi, where each calculus builds on its precursor (Hoare logic builds on abstract Hoare logic). By using stronger judgements about programs and more properties of assertions, it becomes possible to validate more sophisticated rules in later calculi. The first views calculus corresponds directly to Hoare logic, but it uses views and not sets of states for assertions. It supports neither the frame rule nor the concurrency rule of the views framework. The second views calculus validates the frame rule but not the concurrency rule. In the third views calculus, both the frame and the concurrency rules are theorems:

(frame) $\quad \{v\}\, C\, \{v'\} \ \Rightarrow \ \{v * v''\}\, C\, \{v' * v''\}$

(concurrency) $\quad \{v_1\}\, C_1\, \{v_1'\} \ \wedge \ \{v_2\}\, C_2\, \{v_2'\} \ \Rightarrow \ \{v_1 * v_2\}\, C_1 \parallel C_2\, \{v_1' * v_2'\}$

Let $Views$ be the set of all views, ranged over by v. Each view v describes, or is associated with, a set of computational states $\lfloor v \rfloor$. Notice that views need not be isomorphic to sets of states. This decoupling has an important benefit for reasoning: it allows views to have additional structure that does not necessarily exist for sets of states. For example, there might be an operator $(*)$ on views with no corresponding operator on sets of states such that $\lfloor v * v' \rfloor = \lfloor v \rfloor * \lfloor v' \rfloor$. The operator $(*)$ and its properties, and other aspects of views, will be introduced when needed in the presentation.

5.1 Hoare logic

The definition of the Hoare triple uses the abstract Hoare triple and the mapping from sets of states to descriptions:

$$\begin{aligned} S\,\{P\}\,S' \;\stackrel{\text{def}}{=}\; & (\mathit{ic\text{-}traces\text{-}ending\text{-}in}(S) \cup \mathit{Inconsistent})\,\langle P\rangle \\ & (\mathit{ic\text{-}traces\text{-}ending\text{-}in}(S') \cup \mathit{Inconsistent}) \end{aligned}$$

The judgement $S\,\{P\}\,S'$ asserts that, whenever a trace that established a state in S is extended with a trace of P in a consistent way, the resulting trace will establish a state in S'. This is exactly the meaning of the familiar judgement of Hoare logic. Formally:

$$\begin{aligned} S\,\{P\}\,S' \;\Leftrightarrow\; & [\forall t \in \mathit{ic\text{-}traces\text{-}ending\text{-}in}(S) : \forall t' \in P : \\ & \mathit{ic}(t{+\!\!+}t') \;\Rightarrow\; t{+\!\!+}t' \in \mathit{ic\text{-}traces\text{-}ending\text{-}in}(S')] \end{aligned}$$

When P is an atom, we have $S\,\{a\}\,S' \;\Leftrightarrow\; a(S) \subseteq S'$ and hence:

(Hatom) $$a(S) \subseteq S' \;\Rightarrow\; S\,\{a\}\,S'$$

Several rules of Hoare logic follow immediately as theorems from the definition of the triple and their counterparts in the abstract Hoare logic. In particular:

(Hskip) $S\,\{\mathit{skip}\}\,S$

(Hseq) $S\,\{P\}\,S' \;\wedge\; S'\,\{P'\}\,S'' \;\Rightarrow\; S\,\{P\,;P'\}\,S''$

(Hchoice) $(\forall P \in X : S\,\{P\}\,S') \;\Rightarrow\; S\,\{\bigcup X\}\,S'$

(Hiter) $S\,\{P\}\,S \;\Rightarrow\; S\,\{P^*\}\,S$

The rule of consequence follows from (HAcons) and the equivalence:

$$\mathit{ic\text{-}traces\text{-}ending\text{-}in}(S) \subseteq \mathit{ic\text{-}traces\text{-}ending\text{-}in}(S') \;\Leftrightarrow\; S \subseteq S'$$

(Hcons) $$S \subseteq S' \;\wedge\; S'\,\{P\}\,S'' \;\wedge\; S'' \subseteq S''' \;\Rightarrow\; S\,\{P\}\,S'''$$

Finally, the familiar rule of disjunction holds by (HAcons), (HAdisj) and the fact that $\mathit{Inconsistent}\,;P \subseteq \mathit{Inconsistent}$.

(Hdisj) $$(\forall S \in Z : S\,\{P\}\,S') \;\Rightarrow\; \bigcup Z\,\{P\}\,S'$$

5.2 Basic views calculus

The basic views triple is defined in terms of the Hoare judgement:

$$v \ll P \gg v' \stackrel{\text{def}}{=} \lfloor v \rfloor \{P\} \lfloor v' \rfloor$$

It says that whenever a trace that established v is extended with a trace of P in a consistent way, the resulting trace will establish v'.

Since different instantiations of the theory may choose to use different views and atoms, the views calculi are parametric in a set $Axioms \subseteq Views \times Atoms \times Views$. The axioms describe how atoms transform views. The basic calculus requires that each axiom must be sound in the obvious sense:

- $(v, a, v') \in Axioms \;\Rightarrow\; a(\lfloor v \rfloor) \subseteq \lfloor v' \rfloor$

This requirement and (Hatom) imply the basic rule for atoms:

(Batom) $\qquad (v, a, v') \in Axioms \;\Rightarrow\; v \ll a \gg v'$

Several rules of the basic views calculus follow immediately as theorems from their counterparts in Hoare logic and the definition of the basic triple. For instance:

(Bskip) $\quad v \ll \mathit{skip} \gg v$

(Bseq) $\quad v \ll P \gg v' \;\wedge\; v' \ll P' \gg v'' \;\Rightarrow\; v \ll P\,;P' \gg v''$

(Bchoice) $\quad (\forall P \in X : v \ll P \gg v') \;\Rightarrow\; v \ll \bigcup X \gg v'$

(Biter) $\quad v \ll P \gg v \;\Rightarrow\; v \ll P^* \gg v$

(Bcons′) $\quad \lfloor v \rfloor \subseteq \lfloor v' \rfloor \;\wedge\; v' \ll P \gg v'' \;\wedge\; \lfloor v'' \rfloor \subseteq \lfloor v''' \rfloor \;\Rightarrow\; v \ll P \gg v'''$

The judgement of the basic calculus is rather weak and easy to establish. This has two important consequences. Firstly, it does not constrain the program description (the middle operand of the triple) much, so the calculus has broad applicability. Secondly, it does not support reasoning that depends on stronger assumptions about programs. For example, reasoning with the frame rule requires that programs must preserve all frames. Although the basic calculus does not guarantee this, it is the primary goal of the framing calculus.

5.3 Framing calculus

The frame rule uses a binary operator $(*)$ that combines two views to yield a third. A basic requirement is that $(Views, *)$ must form an Abelian semigroup (Parameter D in [1]):

- $(*)$ is associative and commutative.

The judgement of the framing calculus is defined as a basic views triple that preserves all frames:

$$v\,[P]\,v' \stackrel{\text{def}}{=} v \ll P \gg v' \;\wedge\; \forall v'' \in Views : (v * v'') \ll P \gg (v' * v'')$$

The framing judgement is clearly stronger than the basic one:

Theorem 1. $v\,[P]\,v' \;\Rightarrow\; v \ll P \gg v'$

and the frame rule is a theorem because $(*)$ is associative:

(Fframe) $$v\,[P]\,v' \;\Rightarrow\; v * v''\,[P]\,v' * v''$$

To obtain a rule that is similar to (Batom), the chosen axioms must additionally preserve all frames:

- $(v, a, v') \in \mathit{Axioms} \;\Rightarrow\; \forall v'' : a(\lfloor v * v'' \rfloor) \subseteq \lfloor v' * v'' \rfloor$

A convenient way to combine the requirements is to define the helper judgement (Definition 11 in [1]):

$$a \Vdash \{v\}\{v'\} \;\stackrel{\text{def}}{=}\; a(\lfloor v \rfloor) \subseteq \lfloor v' \rfloor \;\wedge\; \forall v'' : a(\lfloor v * v'' \rfloor) \subseteq \lfloor v' * v'' \rfloor$$

Then all axioms must be sound in the following sense (Property G in [1]):

$$(v, a, v') \in \mathit{Axioms} \;\Rightarrow\; a \Vdash \{v\}\{v'\}$$

Since the equivalence $a \Vdash \{v\}\{v'\} \;\Leftrightarrow\; v\,[a]\,v'$ holds, which could alternatively serve as definition of the helper judgement, we have:

(Fatom) $$(v, a, v') \in \mathit{Axioms} \;\Rightarrow\; v\,[a]\,v'$$

Several rules follow directly from the corresponding ones in the basic calculus and the definition of the framing triple:

(Fskip) $v\,[\mathit{skip}]\,v$

(Fseq) $v\,[P]\,v' \;\wedge\; v'\,[P']\,v'' \;\Rightarrow\; v\,[P\,;P']\,v''$

(Fchoice) $(\forall P \in X : v\,[P]\,v') \;\Rightarrow\; v\,[\bigcup X]\,v'$

(Fiter) $v\,[P]\,v \;\Rightarrow\; v\,[P^*]\,v$

However, a direct counterpart of (Bcons′) does not hold in the framing calculus. Although the assumptions yield a valid Hoare triple, there is no guarantee that this triple will preserve all frames. An elegant solution is to use a stronger notion of consequence that will properly constrain the views of the new pre- and postcondition. This is the purpose of the view entailment relation:

$$v \preceq v' \;\stackrel{\text{def}}{=}\; \lfloor v \rfloor \subseteq \lfloor v' \rfloor \;\wedge\; (\forall v'' : \lfloor v * v'' \rfloor \subseteq \lfloor v' * v'' \rfloor)$$

View entailment has several equivalent characterisations (the last characterisaton, i.e. $v \preceq v' \;\Leftrightarrow\; \mathit{noop} \Vdash \{v\}\{v'\}$, corresponds to Definition 12 in [1]):

Lemma 1. $v \preceq v' \;\Leftrightarrow\; v\,[\mathit{skip}]\,v' \;\Leftrightarrow\; \mathit{noop} \Vdash \{v\}\{v'\}$

Here *noop* is the atom that is isomorphic to the identity relation on states. This gives a simple way to specify entailments: if $(v, \mathit{noop}, v') \in \mathit{Axioms}$, then $v \preceq v'$.

View entailment is reflexive and transitive, and hence a preorder. The definition also implies that $\lfloor - \rfloor$ is monotone with respect to entailment: $v \preceq v' \;\Rightarrow\; \lfloor v \rfloor \subseteq \lfloor v' \rfloor$. The associativity of $(*)$ makes it monotone as well: $v \preceq v' \;\Rightarrow\; v * v'' \preceq v' * v''$. It is therefore trivial to prove the next theorems:

(Bcons) $\quad v \preceq v' \;\wedge\; v' \ll P \gg v'' \;\wedge\; v'' \preceq v''' \;\Rightarrow\; v \ll P \gg v'''$

(Fcons) $\quad v \preceq v' \;\wedge\; v' \,[P]\, v'' \;\wedge\; v'' \preceq v''' \;\Rightarrow\; v \,[P]\, v'''$

The framing calculus supports only top-level framing – it does not constrain what happens at the intermediate steps of a computation. Compositional reasoning about concurrency usually demands internal framing, which ensures that concurrent programs do not interfere during execution to invalidate each other's views. The next calculus gains the concurrency rule by doing exactly this.

5.4 Full views calculus

The full views calculus does not reason directly about descriptions. Instead, users of the calculus (e.g. programming languages or humans) must represent a description as a *command*, which is a formal language over atoms. Factoring a description into atoms provides a simple way for the calculus to consider its internal structure: if every atom supports framing, then every intermediate step will preserve the view of a concurrent environment. As a result, the calculus can offer a compositional rule for concurrency.

The mapping from commands to descriptions is straightforward. Every atom sequence has an associated set of traces:

$$\begin{aligned} &\mathit{traces\text{-}of\text{-}atom\text{-}seq}([]) \;\stackrel{\text{def}}{=}\; \mathit{skip} \\ &\mathit{traces\text{-}of\text{-}atom\text{-}seq}(a : as) \;\stackrel{\text{def}}{=}\; a \,;\, \mathit{traces\text{-}of\text{-}atom\text{-}seq}(as) \end{aligned}$$

Likewise, every command also has a corresponding set of traces, which is the description it denotes:

$$\mathit{traces\text{-}of\text{-}comm}(C) \;\stackrel{\text{def}}{=}\; \bigcup \{ \mathit{traces\text{-}of\text{-}atom\text{-}seq}(as) \mid as \in C \}$$

Let a, when used as a command, denote the singleton language $\{[a]\}$.

The judgement of the full views calculus uses an auxiliary judgement for atom sequences:

$$\begin{aligned} &v \,\#[]\#\, v' \;\stackrel{\text{def}}{=}\; v \,[\mathit{skip}]\, v' \\ &v \,\#(a : as)\#\, v' \;\stackrel{\text{def}}{=}\; \exists v'' \in \mathit{Views} : v \,[a]\, v'' \;\wedge\; v'' \,\#as\#\, v' \end{aligned}$$

The definition of the main judgement quantifies over all the atom sequences of a command:

$$\{v\}\, C \,\{v'\} \;\stackrel{\text{def}}{=}\; \forall as \in C : v \,\#as\#\, v'$$

The new judgements are stronger than the judgement of the framing calculus in the obvious sense – by induction on as, it follows from (Fseq) that:

Lemma 2. $v \,\#as\#\, v' \;\Rightarrow\; v\,[\mathit{traces\text{-}of\text{-}atom\text{-}seq}(as)]\,v'$

and this lemma, together with (Fchoice), imply:

Theorem 2. $\{v\}\,C\,\{v'\} \;\Rightarrow\; v\,[\mathit{traces\text{-}of\text{-}comm}(C)]\,v'$

Most rules of the full views calculus rely on lemmas about the auxiliary judgement. These lemmas are typically proved by induction on atom sequences. Standard mathematical machinery, such as induction, will not be mentioned in the discussion below. Only the important ingredients of a proof are made explicit, such as direct or indirect dependencies on the properties of views. If a rule immediately follows a lemma, then it is a trivial corollary thereof.

Using (Fatom) and (Fskip), it is simple to establish:

(Vatom) $$(v, a, v') \in \mathit{Axioms} \;\Rightarrow\; \{v\}\,a\,\{v'\}$$

The rule (Fskip) implies:

(Vskip) $$\{v\}\,\mathit{skip}\,\{v\}$$

By (Fseq), it holds that:

Lemma 3. $v \,\#as\#\, v' \;\wedge\; v' \,\#as'\#\, v'' \;\Rightarrow\; v \,\#(as +\!\!+ as')\#\, v''$

(Vseq) $$\{v\}\,C\,\{v'\} \;\wedge\; \{v'\}\,C'\,\{v''\} \;\Rightarrow\; \{v\}\,C\,;C'\,\{v''\}$$

It is trivial to establish the rule for nondeterministic choice:

(Vchoice) $$(\forall C \in Y : \{v\}\,C\,\{v'\}) \;\Rightarrow\; \{v\}\,\bigcup Y\,\{v'\}$$

By (Vskip) and (Vseq), the following lemma holds:

Lemma 4. $\{v\}\,C\,\{v\} \;\Rightarrow\; \{v\}\,C^n\,\{v\}$

This lemma and (Vchoice) imply the rule for iteration:

(Viter) $$\{v\}\,C\,\{v\} \;\Rightarrow\; \{v\}\,C^*\,\{v\}$$

The rule (Fcons) and the reflexivity of $\preceq$ can be used to prove:

Lemma 5. $v \preceq v' \;\wedge\; v' \,\#as\#\, v'' \;\wedge\; v'' \preceq v''' \;\Rightarrow\; v \,\#as\#\, v'''$

(Vcons) $$v \preceq v' \;\wedge\; \{v'\}\,C\,\{v''\} \;\wedge\; v'' \preceq v''' \;\Rightarrow\; \{v\}\,C\,\{v'''\}$$

From (Fframe) follows:

Lemma 6. $v \,\#as\#\, v' \;\Rightarrow\; (v * v'') \,\#as\#\, (v' * v'')$

(Vframe) $$\{v\}\,C\,\{v'\} \;\Rightarrow\; \{v * v''\}\,C\,\{v' * v''\}$$

With (Fframe), (Fseq) and the commutativity of $*$, one can show:

Lemma 7. $v_1 \#as_1\#\, v_1' \;\wedge\; v_2 \,\#as_2\#\, v_2' \;\wedge\; as \in as_1 \otimes as_2 \;\Rightarrow (v_1 * v_2) \,\#as\#\, (v_1' * v_2')$

This directly yields the compositional rule for concurrency:

(Vconc) $$\{v_1\}\,C_1\,\{v_1'\} \;\wedge\; \{v_2\}\,C_2\,\{v_2'\} \;\Rightarrow\; \{v_1 * v_2\}\,C_1 \parallel C_2\,\{v_1' * v_2'\}$$

The frame rule can be seen as a special case of the concurrency rule, since (Vframe) can be derived from (Vconc), (Vskip) and $C \parallel \mathit{skip} = C$.

6 Operational calculi

Program execution can be investigated formally with operational calculi, which help to discover valid executions of programs.

Small-step calculi, such as the Plotkin [11] and Milner [12] ones, are concerned with how a computation can unfold by performing (a sequence of) actions that are easy to implement in a computer. These calculi are therefore parametric in a set

$$AtomicOperations \subseteq Atoms$$

whose elements model small atomic steps. For example, a hypothetical programming language might include Boolean tests, variable assignments and heap operations in this set. This section defines the set $Actions$, as mentioned in the abstract Plotkin and Milner calculi of Section 3, in terms of $AtomicOperations$:

$$Actions \stackrel{\text{def}}{=} \{\textit{skip}\} \cup AtomicOperations$$

Think about *skip* as the trivial action that takes zero time to execute. It cannot change the computational state. In contrast to this, an action from $AtomicOperations$ embodies real work and may transform the state. Think about its execution as taking, say, one time unit.

Big-step calculi, such as Kahn's natural semantics [13] (see [14] for its application to imperative programming), describe only the ultimate results of a computation. Because the focus is on discovering complete executions, there is no mention of small steps or partial executions and hence no possibility for nontermination.

This section develops operational calculi for descriptions and also for commands, where a command is a formal language over atoms.

6.1 Descriptions

The abstract Plotkin and Kahn calculi do not mention computational states explicitly. However, they can be instantiated to obtain the familiar versions.

Plotkin calculus The judgement of the Plotkin calculus is defined in terms of the abstract one:

$\langle P, \sigma\rangle \longrightarrow \langle P', \sigma'\rangle \stackrel{\text{def}}{=}$
$\exists t \in \textit{ic-traces-ending-in}(\sigma) : \exists t' \in \textit{ic-traces-ending-in}(\sigma') : \langle P, \{t\}\rangle \longrightarrow \langle P', \{t'\}\rangle$

It says that one way of executing P is to execute some action followed by P'. The action itself is hidden – only its effect on the state is explicit in the judgement.

There is also an equivalent characterisation, which expresses that what happened before the initial state does not matter:

Lemma 8. $\langle P, \sigma\rangle \longrightarrow \langle P', \sigma'\rangle \Leftrightarrow$
$\forall t \in \textit{ic-traces-ending-in}(\sigma) : \exists t' \in \textit{ic-traces-ending-in}(\sigma') : \langle P, \{t\}\rangle \longrightarrow \langle P', \{t'\}\rangle$

The rules of the Plotkin calculus are all easy to derive. A rule for atomic operations follows from (PAaction):

(Patom) $\quad a \in \mathit{AtomicOperations} \ \wedge \ \sigma' \in a(\sigma) \ \Rightarrow \ \langle a, \sigma\rangle \longrightarrow \langle \mathit{skip}, \sigma'\rangle$

Other rules are trivial consequences of their abstract counterparts.

(Pseq1) $\quad \langle \mathit{skip}\,; P, \sigma\rangle \longrightarrow \langle P, \sigma\rangle$

(Pseq2) $\quad \langle P, \sigma\rangle \longrightarrow \langle R, \sigma'\rangle \ \Rightarrow \ \langle P\,; P', \sigma\rangle \longrightarrow \langle R\,; P', \sigma'\rangle$

(Pchoice) $\quad P \in X \ \Rightarrow \ \langle \bigcup X, \sigma\rangle \longrightarrow \langle P, \sigma\rangle$

(Piter1) $\quad \langle P^*, \sigma\rangle \longrightarrow \langle \mathit{skip}, \sigma\rangle$

(Piter2) $\quad \langle P^*, \sigma\rangle \longrightarrow \langle P\,; P^*, \sigma\rangle$

(Pconc1) $\quad \langle \mathit{skip} \parallel P, \sigma\rangle \longrightarrow \langle P, \sigma\rangle$

(Pconc2) $\quad \langle P \parallel \mathit{skip}, \sigma\rangle \longrightarrow \langle P, \sigma\rangle$

(Pconc3) $\quad \langle P, \sigma\rangle \longrightarrow \langle R, \sigma'\rangle \ \Rightarrow \ \langle P \parallel P', \sigma\rangle \longrightarrow \langle R \parallel P', \sigma'\rangle$

(Pconc4) $\quad \langle P, \sigma\rangle \longrightarrow \langle R, \sigma'\rangle \ \Rightarrow \ \langle P' \parallel P, \sigma\rangle \longrightarrow \langle P' \parallel R, \sigma'\rangle$

Notice that, if $a \in \mathit{AtomicOperations}$, and $\forall \sigma \in \Sigma : a(\sigma) \neq \emptyset$, then the above rules can discover an infinite execution from the starting configuration $\langle a^*, \sigma\rangle$ that performs real work (by executing a) infinitely many times. This is consistent with [1], and shows that the small-step way of discovering executions can give rise to nontermination despite the fact that all the traces of a^* are finite. Nontermination results from the way in which an execution strategy views a program. It is not an intrinsic property of the program itself. This philosophical viewpoint differs significantly from work that defines the meaning of a program in terms of a particular operational calculus. The difference is also reflected by the fact that infinite traces are excluded from our domain.

The definitions of the iterated and reflexive transitive versions of the judgement are standard:

$$\langle P, \sigma\rangle \longrightarrow^0 \langle P', \sigma'\rangle \ \stackrel{\text{def}}{=} \ P = P' \ \wedge \ \sigma = \sigma'$$

$$\langle P, \sigma\rangle \longrightarrow^{n+1} \langle P', \sigma'\rangle \ \stackrel{\text{def}}{=} \ \exists P'', \sigma'' : \langle P, \sigma\rangle \longrightarrow \langle P'', \sigma''\rangle \ \wedge \ \langle P'', \sigma''\rangle \longrightarrow^n \langle P', \sigma'\rangle$$

$$\langle P, \sigma\rangle \longrightarrow^* \langle P', \sigma'\rangle \ \stackrel{\text{def}}{=} \ \exists n : \langle P, \sigma\rangle \longrightarrow^n \langle P', \sigma'\rangle$$

Kahn calculus The judgement of the Kahn calculus is defined in terms of the abstract Kahn judgement:

$$\langle P, \sigma\rangle \longrightarrow \sigma' \ \stackrel{\text{def}}{=}$$
$$\exists t \in \mathit{ic\text{-}traces\text{-}ending\text{-}in}(\sigma) : \exists t' \in \mathit{ic\text{-}traces\text{-}ending\text{-}in}(\sigma') : \langle P, \{t\}\rangle \longrightarrow \{t'\}$$

It says that P has an internally consistent trace that can transform the initial state σ into the final state σ'. What brought about the initial state is again unimportant:

Lemma 9. $\langle P, \sigma\rangle \longrightarrow \sigma' \ \Leftrightarrow$
$\forall t \in \mathit{ic\text{-}traces\text{-}ending\text{-}in}(\sigma) : \exists t' \in \mathit{ic\text{-}traces\text{-}ending\text{-}in}(\sigma') : \langle P, \{t\}\rangle \longrightarrow \{t'\}$

It is straightforward to obtain a rule for executing atoms:

(Katom) $\qquad \sigma' \in a(\sigma) \;\Rightarrow\; \langle a,\, \sigma\rangle \longrightarrow \sigma'$

Other rules follow directly from Lemma 9 and their abstract counterparts:

(Kskip) $\qquad \langle \mathit{skip},\, \sigma\rangle \longrightarrow \sigma$

(Kseq) $\qquad \langle P,\, \sigma\rangle \longrightarrow \sigma' \;\wedge\; \langle P',\, \sigma'\rangle \longrightarrow \sigma'' \;\Rightarrow\; \langle P\,;P',\, \sigma\rangle \longrightarrow \sigma''$

(Kchoice) $\qquad P \in X \;\wedge\; \langle P,\, \sigma\rangle \longrightarrow \sigma' \;\Rightarrow\; \langle \bigcup X,\, \sigma\rangle \longrightarrow \sigma'$

(Kiter1) $\qquad \langle P^*,\, \sigma\rangle \longrightarrow \sigma$

(Kiter2) $\qquad \langle P,\, \sigma\rangle \longrightarrow \sigma' \;\wedge\; \langle P^*,\, \sigma'\rangle \longrightarrow \sigma'' \;\Rightarrow\; \langle P^*,\, \sigma\rangle \longrightarrow \sigma''$

(Kconc1) $\qquad \langle P,\, \sigma\rangle \longrightarrow \sigma' \;\wedge\; \langle P',\, \sigma'\rangle \longrightarrow \sigma'' \;\Rightarrow\; \langle P \parallel P',\, \sigma\rangle \longrightarrow \sigma''$

(Kconc2) $\qquad \langle P',\, \sigma\rangle \longrightarrow \sigma' \;\wedge\; \langle P,\, \sigma'\rangle \longrightarrow \sigma'' \;\Rightarrow\; \langle P \parallel P',\, \sigma\rangle \longrightarrow \sigma''$

Relationships The Plotkin judgement can be characterised in terms of the Milner and Kahn judgements:

Lemma 10. $\langle P,\, \sigma\rangle \longrightarrow \langle P',\, \sigma'\rangle \;\Leftrightarrow\; \exists Q : P \xrightarrow{Q} P' \;\wedge\; \langle Q,\, \sigma\rangle \longrightarrow \sigma'$

This lemma makes it clear that the Plotkin judgement hides the action that effected the state change, as Q is existentially quantified.

An equivalent and perhaps more familiar formulation uses a function $[\![-]\!]$ that captures the state-transformation behaviour of a description:

$$[\![P]\!](\sigma) \;\stackrel{\text{def}}{=}\; \{\sigma' \mid \langle P,\, \sigma\rangle \longrightarrow \sigma'\}$$

Thus $\sigma' \in [\![P]\!](\sigma) \;\Leftrightarrow\; \langle P,\, \sigma\rangle \longrightarrow \sigma'$, and the relationship of Lemma 10 can be written as follows: $\langle P,\, \sigma\rangle \longrightarrow \langle P',\, \sigma'\rangle \;\Leftrightarrow\; \exists Q : P \xrightarrow{Q} P' \wedge \sigma' \in [\![Q]\!](\sigma)$. Note that actions have simple state-transformation behaviour: $[\![\mathit{skip}]\!](\sigma) = \{\sigma\}$ and $[\![a]\!](\sigma) = a(\sigma)$.

Another relationship involves the Plotkin and Kahn judgements. The judgement $\langle P,\, \sigma\rangle \longrightarrow^* \langle \mathit{skip},\, \sigma'\rangle$ says that P can transform the initial state σ into the final state σ':

Lemma 11. $\langle P,\, \sigma\rangle \longrightarrow^* \langle \mathit{skip},\, \sigma'\rangle \;\Rightarrow\; \langle P,\, \sigma\rangle \longrightarrow \sigma'$

It also says that the input/output state transformations described by the reflexive transitive closure of the Plotkin judgement approximate those of the Kahn judgement (whether or not the converse holds depends on the choice of $\mathit{AtomicOperations}$). Nevertheless, this does not mean that the Plotkin judgement is useless. It yields a calculus with interesting syntax-directed rules for concurrency, while the Kahn calculus has only trivial ones. Furthermore, the Plotkin calculus induces non-terminating behaviours and the Kahn calculus does not.

Lemma 11 may surprise readers who are used to operational judgements that are defined in terms of syntax-directed rules. For example, (Kconc1) and (Kconc2) do not express the interleaving of concurrent descriptions, so it is easy to give examples where

the Plotkin calculus can derive $\langle P \parallel P',\, \sigma\rangle \longrightarrow^* \langle \mathit{skip},\, \sigma'\rangle$ but the Kahn calculus cannot derive $\langle P \parallel P',\, \sigma\rangle \longrightarrow \sigma'$. This does not contradict Lemma 11, however, because here the operational judgements are not defined by rules. Instead, each definition directly describes the idea behind a judgement. Rules such as (Kconc1) and (Pconc3) are simply theorems that help to discover valid executions – they do not define anything in this formalisation.

6.2 Commands

Although commands are only dressed-up descriptions, it may be useful to reason directly about their execution. Building operational calculi for commands on top of those for descriptions is simple because the mapping from commands to descriptions preserves the programming operators:

Lemma 12 (Homomorphism).

1. $\mathit{traces\text{-}of\text{-}comm}(a) = a$
2. $\mathit{traces\text{-}of\text{-}comm}(\mathit{skip}) = \mathit{skip}$
3. $\mathit{traces\text{-}of\text{-}comm}(C\,;C') = \mathit{traces\text{-}of\text{-}comm}(C)\,;\mathit{traces\text{-}of\text{-}comm}(C')$
4. $\mathit{traces\text{-}of\text{-}comm}(\bigcup Y) = \bigcup\{\mathit{traces\text{-}of\text{-}comm}(C) \mid C \in Y\}$
5. $\mathit{traces\text{-}of\text{-}comm}(C^*) = \mathit{traces\text{-}of\text{-}comm}(C)^*$
6. $\mathit{traces\text{-}of\text{-}comm}(C \parallel C') = \mathit{traces\text{-}of\text{-}comm}(C) \parallel \mathit{traces\text{-}of\text{-}comm}(C')$

Plotkin calculus The Plotkin judgement for commands is defined in terms of the one for descriptions:

$$\langle C,\, \sigma\rangle \longrightarrow \langle C',\, \sigma'\rangle \stackrel{\text{def}}{=} \langle \mathit{traces\text{-}of\text{-}comm}(C),\, \sigma\rangle \longrightarrow \langle \mathit{traces\text{-}of\text{-}comm}(C'),\, \sigma'\rangle$$

The Plotkin rules for commands immediately follow as theorems from the corresponding rules for descriptions and Lemma 12.

Iterated and reflexive transitive versions of the judgement can be defined in the usual way, and the following relationship holds:

Lemma 13. $\langle C,\, \sigma\rangle \longrightarrow^* \langle C',\, \sigma'\rangle \;\Rightarrow$
$\langle \mathit{traces\text{-}of\text{-}comm}(C),\, \sigma\rangle \longrightarrow^* \langle \mathit{traces\text{-}of\text{-}comm}(C'),\, \sigma'\rangle$

Milner calculus The Milner judgement for commands uses the Milner judgement for descriptions:

$$C \stackrel{C'}{\longrightarrow} C'' \stackrel{\text{def}}{=} \mathit{traces\text{-}of\text{-}comm}(C) \stackrel{\mathit{traces\text{-}of\text{-}comm}(C')}{\longrightarrow} \mathit{traces\text{-}of\text{-}comm}(C'')$$

The previous Milner rules and Lemma 12 directly yield Milner rules for executing commands.

Kahn calculus Here is the Kahn judgement for commands:

$$\langle C,\, \sigma\rangle \longrightarrow \sigma' \stackrel{\text{def}}{=} \langle \mathit{traces\text{-}of\text{-}comm}(C),\, \sigma\rangle \longrightarrow \sigma'$$

As expected, Lemma 12 is useful for deriving big-step rules for commands from the ones for descriptions.

Relationships The Plotkin, Milner and Kahn judgements for commands enjoy a similar relationship as before:

Lemma 14. $\langle C, \sigma\rangle \longrightarrow \langle C', \sigma'\rangle \;\Leftrightarrow\; \exists C'' : C \xrightarrow{C''} C' \;\wedge\; \langle C'', \sigma\rangle \longrightarrow \sigma'$

This can also be formulated in terms of a state-transformation function for commands:

$$[\![C]\!](\sigma) \;\stackrel{\text{def}}{=}\; \{\sigma' \mid \langle C, \sigma\rangle \longrightarrow \sigma'\}$$

Note that $[\![C]\!] = [\![\mathit{traces\text{-}of\text{-}comm}(C)]\!]$, so Lemma 12 can help to characterise the state-transformation behaviour of commands as e.g. equations.

A familiar relationship holds between the Plotkin and Kahn judgements:

Lemma 15. $\langle C, \sigma\rangle \longrightarrow^* \langle \mathit{skip}, \sigma'\rangle \;\Rightarrow\; \langle C, \sigma\rangle \longrightarrow \sigma'$

It is a direct consequence of Lemmas 11 to 13.

Discussion The small-step calculi above do not strictly mirror a command's internal structure. However, it is possible to obtain alternative small-step calculi that take the factoring of atoms into account.

For example, by instantiating the set $\mathit{Actions}$ in Section 3 with $\{\mathit{skip}\} \cup \{\{[a]\} \mid a \in \mathit{AtomicOperations}\}$, the Milner triple $C \xrightarrow{a} C'$ holds exactly when the atom a is in $\mathit{AtomicOperations}$ and all atom sequences in C' with a prepended to them are elements of C.

One can then define an alternative Plotkin judgement by using this Milner triple in the relationship of Lemma 14.

It is easy to show that these judgements are stronger than the versions of before. Moreover, all the expected operational rules hold as theorems. The proof script [2] contains more details.

7 Partial correctness

We have seen the definitions of several judgements and explored theorems about them. Many theorems, such as (Vconc) and (Kseq), involve only one kind of judgement. Other theorems relate different kinds of deductive judgement (e.g. Theorems 1 and 2), or different operational judgements (e.g. Lemmas 10 and 11). This section describes correctness theorems that relate various deductive and operational judgements. A correctness theorem makes it clear that a deductive judgement correctly summarises the behaviour of programs according to an operational judgement. If this is the case, then all executions that the operational calculus can discover will be consistent with what the deductive calculus says about a program.

The proof that Hoare logic is correct with respect to the Kahn calculus is straightforward.

Theorem 3. $S\,\{P\}\,S' \;\Leftrightarrow\; (\forall \sigma \in S : \forall \sigma' : \langle P, \sigma\rangle \longrightarrow \sigma' \;\Rightarrow\; \sigma' \in S')$

Together with earlier definitions and results, this theorem simplifies the proofs of other correctness theorems. For example, the following statement is an immediate consequence:

Corollary 1. $S\,\{P\}\,S' \;\Leftrightarrow\; (\forall \sigma \in S : [\![P]\!](\sigma) \subseteq S')$

Lemma 11 renders the correctness of Hoare logic with respect to the Plotkin calculus trivial:

Corollary 2. $S\,\{P\}\,S' \;\Rightarrow\; (\forall \sigma \in S : \langle P, \sigma\rangle \longrightarrow^* \langle \mathit{skip}, \sigma'\rangle \;\Rightarrow\; \sigma' \in S')$

The basic views calculus is consequently also correct:

Corollary 3. $v \ll P \gg v' \;\Rightarrow\; (\forall \sigma \in \lfloor v \rfloor : \langle P, \sigma\rangle \longrightarrow \sigma' \;\Rightarrow\; \sigma' \in \lfloor v' \rfloor)$

Corollary 4. $v \ll P \gg v' \;\Rightarrow\; (\forall \sigma \in \lfloor v \rfloor : [\![P]\!](\sigma) \subseteq \lfloor v' \rfloor)$

Corollary 5. $v \ll P \gg v' \;\Rightarrow\; (\forall \sigma \in \lfloor v \rfloor : \langle P, \sigma\rangle \longrightarrow^* \langle \mathit{skip}, \sigma'\rangle \;\Rightarrow\; \sigma' \in \lfloor v' \rfloor)$

The correctness of the framing calculus follows by Theorem 1.

Corollary 6. $v\,[P]\,v' \;\Rightarrow\; (\forall \sigma \in \lfloor v \rfloor : \langle P, \sigma\rangle \longrightarrow \sigma' \;\Rightarrow\; \sigma' \in \lfloor v' \rfloor)$

Corollary 7. $v\,[P]\,v' \;\Rightarrow\; (\forall \sigma \in \lfloor v \rfloor : [\![P]\!](\sigma) \subseteq \lfloor v' \rfloor)$

Corollary 8. $v\,[P]\,v' \;\Rightarrow\; (\forall \sigma \in \lfloor v \rfloor : \langle P, \sigma\rangle \longrightarrow^* \langle \mathit{skip}, \sigma'\rangle \;\Rightarrow\; \sigma' \in \lfloor v' \rfloor)$

Theorem 2 and Lemma 15 then imply that the full views calculus is also correct with respect to the operational calculi:

Corollary 9. $\{v\}\,C\,\{v'\} \;\Rightarrow\; (\forall \sigma \in \lfloor v \rfloor : \langle C, \sigma\rangle \longrightarrow \sigma' \;\Rightarrow\; \sigma' \in \lfloor v' \rfloor)$

Corollary 10. $\{v\}\,C\,\{v'\} \;\Rightarrow\; (\forall \sigma \in \lfloor v \rfloor : [\![C]\!](\sigma) \subseteq \lfloor v' \rfloor)$

Corollary 11. $\{v\}\,C\,\{v'\} \;\Rightarrow\; (\forall \sigma \in \lfloor v \rfloor : \langle C, \sigma\rangle \longrightarrow^* \langle \mathit{skip}, \sigma'\rangle \;\Rightarrow\; \sigma' \in \lfloor v' \rfloor)$

These proofs of correctness are all short and straightforward. In contrast to the proof in [1], there is no need for coinduction, no mention of particular rules, and no inspection of the program syntax. Notice that the correctness of the full views calculus is essentially reduced to the correctness of Hoare logic, which is simpler and more familiar.

8 Conclusion

The views framework distills the essence of various techniques for reasoning about concurrent programs. This paper constructed a new model that explains its foundations incrementally. Modeling programs as formal languages makes it unnecessary to postulate calculi to define the meaning of programs. The relevant judgements for program correctness and/or execution can instead be defined directly, and the rules can then be established as theorems with the help of familiar algebraic laws. This setup also leads to a short and straightforward proof of partial correctness. The proof does not depend on particular sets of rules or on the syntax of programs. It is even parametric in the choice of the primitive machine-executable operations. In this way, the deductive and operational calculi are largely decoupled from each other. The presentation did not mention (or need) concepts that are important only for particular instantiations of the framework, such as resources, separation, thread permissions, etc. Nonetheless, all the many excellent examples in [1] transfer verbatim to this setting. They show how familiar program logics and type systems can be constructed on top of the theory.

Acknowledgements Special thanks to Tony Hoare for many fruitful discussions and encouragement. This work was supported by the SNSF.

References

1. Dinsdale-Young, T., Birkedal, L., Gardner, P., Parkinson, M., Yang, H.: Views: compositional reasoning for concurrent programs. In: Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL '13, New York, NY, USA, ACM (2013) 287–300
2. Isabelle/HOL proofs. Online at `http://www0.cs.ucl.ac.uk/staff/s.vanstaden/proofs/Views.tgz` (2014)
3. Kozen, D.: A completeness theorem for Kleene algebras and the algebra of regular events. Inf. Comput. **110** (May 1994) 366–390
4. Hoare, C., Möller, B., Struth, G., Wehrman, I.: Concurrent Kleene algebra. In Bravetti, M., Zavattaro, G., eds.: CONCUR 2009 - Concurrency Theory. Volume 5710 of Lecture Notes in Computer Science. Springer Berlin/Heidelberg (2009) 399–414
5. Bloom, S.L., Ésik, Z.: Free shuffle algebras in language varieties. Theoretical Computer Science **163**(1-2) (1996) 55–98
6. Hoare, T., van Staden, S.: The laws of programming unify process calculi. In Gibbons, J., Nogueira, P., eds.: Mathematics of Program Construction. Volume 7342 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2012) 7–22
7. Hoare, T., van Staden, S.: In praise of algebra. Formal Aspects of Computing **24** (2012) 423–431
8. van Staden, S., Hoare, T.: Algebra unifies operational calculi. In Wolff, B., Gaudel, M.C., Feliachi, A., eds.: Unifying Theories of Programming. Volume 7681 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2013) 88–104
9. Park, D.: On the semantics of fair parallelism. In Bjørner, D., ed.: Abstract Software Specifications. Volume 86 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (1980) 504–526
10. Brookes, S.: Full abstraction for a shared-variable parallel language. Information and Computation **127**(2) (1996) 145–163
11. Plotkin, G.D.: A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark (September 1981)
12. Milner, R.: A Calculus of Communicating Systems. Volume 92 of Lecture Notes in Computer Science. Springer (1980)
13. Kahn, G.: Natural semantics. In: Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science. STACS '87, London, UK, Springer-Verlag (1987) 22–39
14. Nielson, H.R., Nielson, F.: Semantics with Applications: A Formal Introduction. Revised edition, July 1999, online at `http://www.daimi.au.dk/~bra8130/Wiley_book/wiley.html`. Original edition published 1992 by John Wiley & Sons.
15. Wehrman, I., Hoare, C.A.R., O'Hearn, P.W.: Graphical models of separation logic. Inf. Process. Lett. **109**(17) (2009) 1001–1004
16. Hoare, C.A.R., Hussain, A., Möller, B., O'Hearn, P., Petersen, R., Struth, G.: On locality and the exchange law for concurrent processes. In Katoen, J.P., König, B., eds.: CONCUR 2011 – Concurrency Theory. Volume 6901 of Lecture Notes in Computer Science. Springer Berlin/Heidelberg (2011) 250–264