An Expressive Trace Logic for Recursive Programs

Dilian Gurov
KTH Royal Institute of Technology, Stockholm, Sweden

Reiner Hähnle
Technical University of Darmstadt, Germany

Abstract

We present an expressive logic over trace formulas, based on binary state predicates, chop, and least fixed points, for precise specification of programs with recursive procedures. Both programs and trace formulas are equipped with a direct-style, fully compositional, denotational semantics that on programs coincides with the standard SOS of recursive programs. We design a compositional proof calculus for proving finite-trace program properties, and prove soundness as well as (relative) completeness. We show that each program can be mapped to a semantics-preserving trace formula and, vice versa, each trace formula can be mapped to a canonical program over slightly extended programs, resulting in a Galois connection between programs and formulas. Our results shed light on the correspondence between programming constructs and logical connectives.

2012 ACM Subject Classification Theory of computation → Denotational semantics; Theory of computation → Operational semantics; Theory of computation → Program specifications; Theory of computation → Program verification; Theory of computation → Logic and verification; Theory of computation → Modal and temporal logics

Keywords and phrases Denotational semantics, compositional semantics, program specification, compositional verification, fixed point logic, trace logic

Digital Object Identifier 10.4230/LIPIcs.FSCD.2025.21

Related Version *Full Version*: `https://arxiv.org/abs/2411.13125` [14]

1 Introduction

It is uncommon that specification languages used in program verification are as expressive as the programs they are intended to specify: In model checking [7] one typically abstracts away from data and unwinds unbounded control structures. Specification of unbounded computations is achieved by recursively defined temporal operators. In deductive verification [15], first-order Hoare-style contracts [18] are the basis of widely used specification languages [3, 22]. The latter specify computation by taking symbolic memory snapshots in terms of first-order formulas, typically at the beginning and at the end of unbounded control structures, such as procedure call and return, or upon loop entry and exit. Contracts permit to approximate the effect of unbounded computation as a finite number of first-order formulas against which a program's behavior can be verified.

Imagine, in contrast, a logic for program specification that is *at least as expressive* as the programs it is supposed to describe. Formulas ϕ of such a logic characterize a generally infinite set of program computation traces σ. One can then form *judgments* of program statements S of the form $S : \phi$ with the natural semantics that any possible trace σ of S is one of the traces described by ϕ.

Two arguments against such a *trace logic* are easily raised: first, one expects a specification language to be capable of *abstraction* from implementation details; second, its computational complexity. Regarding the first, we note that any desired degree of abstraction can be achieved by *definable* constructs, i.e. *syntactic sugar*, in a sufficiently rich trace logic. Regarding the second, it is far from obvious whether the computational worst-case tends to manifest itself in *practical* verification [15]. First experiments seem to indicate this is not the case. On the other hand, a rich, trace-based specification logic bears many advantages:

1. On the practical side, trace-based specification permits to express, for example, that an event or a call happened (exactly once, at least once) or that did not happen. Likewise, one can specify without reference to the target code (via assertions) that a condition holds at some point which is not necessarily an endpoint of a procedure call. It is also possible to specify inter-procedural properties, such as call sequences, absence of callbacks, etc. Such properties are important in security analysis and they are difficult or impossible to express with Hoare-style contracts or in temporal logic.
2. The semantics of trace formulas in terms of trace sets can be closely aligned with a suitable program semantics, which greatly simplifies soundness and completeness proofs.
3. An expressive trace logic makes for simple and natural completeness proofs for judgments $S : \phi$, because it is unnecessary to encode the program semantics in order to construct sufficiently strong first-order conditions.
4. In a calculus for judgments of the form $S : \phi$, one can design *inference* rules that directly decompose judgments into simpler ones, but also *algebraic* rules that simplify and transform ϕ, and one may mix both reasoning styles.
5. The semantics of programs, of trace formulas, and the rules of the calculus can be formulated in a *fully compositional* manner: definitions and inference rules involve no intermediate state or context.
6. In consequence, we are able to establish a close correspondence between programs and formulas, which sheds light on the exact relation between each program construct and its corresponding logical connective. We formulate and prove a Galois connection that formalizes this fact.

Our main contribution is a trace logic for imperative programs with recursive procedures where we formalize and prove the claims above. We restrict attention to the *terminating executions* of programs, i.e. to their *finite trace semantics*. This is not a fundamental limitation, but the desire not to obscure the construction with complications that may be added later. Still, adaptating the theory to maximal traces (including infinite runs) is not trivial, and working out the details is left as future work (see Section 8.3).

Our paper is structured as follows: In Section 2 we define the programming language **Rec** used throughout this paper and we give it a standard SOS semantics [29]. In Section 3 we define a denotational semantics for **Rec** in fully compositional style and prove it equivalent to the SOS semantics. In Section 4 we introduce our trace logic and map programs to formulas by the concept of a *strongest trace formula*, which is shown to fully preserve program semantics. In Section 5 we define a proof calculus for judgments and show it to be sound and complete. In Section 6, using the concept of a *canonical program*, we map trace formulas back to programs in the slightly extended language $\mathbf{Rec}^*$ with non-deterministic guards. We prove that $\mathbf{Rec}^*$ and trace formulas form a Galois connection via strongest trace formulas and canonical programs. In Section 7 we discuss related work, in Section 8 we sketch some extensions, including options to render specifications more abstract and how to prove consequence among trace formulas. In Section 9 we conclude.

All proofs and some examples can be found in the accompanying report [14]. The most important of these are also available in the present Appendix.

2 The Programming Language Rec

We define a simple programming language **Rec** with recursive procedures and give it a standard SOS semantics. We follow the notation of [26], adapted to **Rec** syntax.

▶ **Definition 2.1** (**Rec** Syntax). *A* **Rec** *program is a pair* $\langle S, T\rangle$*, where* S *is a statement with the* abstract syntax *defined by the grammar:*

$$S ::= \textbf{skip} \mid x := a \mid S_1; S_2 \mid \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \mid m()$$

and where T *is a* procedure table *given by* $T ::= M^*$ *with declarations of parameter-less procedures* $M ::= m\,\{S\}$*.*

In the grammar, a *ranges over arithmetic expressions* **AExp***, and* b *over Boolean expressions* **BExp***. Both are assumed to be side-effect free and are not specified further. All variables are global and range over the set of integers* $\mathbb{Z}$*. We assume programs are* well-formed*, i.e., only declared procedures are called, and procedure names are unique.*

▶ **Example 2.2.** Consider the **Rec** program $p_0 \stackrel{\text{def}}{=} \langle S, T\rangle$ with statement $S \stackrel{\text{def}}{=} x := 3; \mathit{even}()$ and procedure table:

$$T \stackrel{\text{def}}{=} \begin{array}{l} \mathit{even}\ \{\textbf{if } x = 0 \textbf{ then } y := 1 \textbf{ else } x := x - 1; \mathit{odd}()\} \\ \ \mathit{odd}\ \{\textbf{if } x = 0 \textbf{ then } y := 0 \textbf{ else } x := x - 1; \mathit{even}()\} \end{array}$$

The intended semantics is that *even* terminates in a state where $y = 1$ if and only if it is started in a state where x is even and non-negative, and it terminates in a state where $y = 0$ if and only if it is started in a state where x is odd and non-negative.

▶ **Example 2.3.** Consider the **Rec** program $p_1 \stackrel{\text{def}}{=} \langle S, T\rangle$ with $S \stackrel{\text{def}}{=} \mathit{down}()$ and procedure table:

$$T \stackrel{\text{def}}{=} \mathit{down}\,\{\textbf{if } x > 0 \textbf{ then } x := x - 2; \mathit{down}() \textbf{ else skip}\}$$

The intended semantics is that p_1 terminates in a state where $x = 0$ if and only if it is started in state where x is even and non-negative.

▶ Remark 2.4. Loops can be defined with the help of (tail-)recursive programs. For example, a loop of the form "**while** b **do** S" can be simulated with a procedure declared in T as:

$$m\,\{\textbf{if } b \textbf{ then } S; m() \textbf{ else skip}\}$$

using a unique name m and replacing the occurrence of the loop with a call to $m()$.

A standard, *structural operational semantics* (SOS) for **Rec** is defined in Figure 1 (sometimes referred to as *small-step semantics*). We use it as a baseline when defining the denotational finite-trace semantics in Section 3.

Let *Var* be the set of program variables, and **State** the set of program states $s : \mathit{Var} \to \mathbb{Z}$. Let $\mathcal{A}[\![a]\!]\,(s) \in \mathbb{Z}$ denote the integer value of the arithmetic expression a when evaluated in state s, and $\mathcal{B}[\![b]\!]\,(s) \in \mathbb{T}$ denote the truth value of the Boolean expression b when evaluated in state s, both defined as expected.

A *configuration* is either a pair $\langle S, s\rangle$ consisting of a statement and a state, designating an initial or intermediate configuration; or a state s, designating a final configuration. To simplify notation we assume that S is evaluated relative to a **Rec** program with a procedure table T which is not explicitly specified.

The transitions of the SOS either relate two intermediate configurations, or an intermediate with a final one, and thus have the shape $\langle S, s\rangle \Rightarrow \langle S', s'\rangle$ or $\langle S, s\rangle \Rightarrow s'$, respectively.

$$\textsc{Skip}\ \frac{-}{\langle \mathbf{skip}, s\rangle \Rightarrow s} \qquad \textsc{Assign}\ \frac{-}{\langle x := a, s\rangle \Rightarrow s[x \mapsto \mathcal{A}[\![a]\!]\,(s)]}$$

$$\textsc{Seq-1}\ \frac{\langle S_1, s\rangle \Rightarrow s'}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_2, s'\rangle} \qquad \textsc{Seq-2}\ \frac{\langle S_1, s\rangle \Rightarrow \langle S_1', s'\rangle}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_1'; S_2, s'\rangle}$$

$$\textsc{If-1}\ \frac{-}{\langle \mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2, s\rangle \Rightarrow \langle S_1, s\rangle} \quad \text{if } \mathcal{B}[\![b]\!]\,(s) = \mathbf{tt}$$

$$\textsc{If-2}\ \frac{-}{\langle \mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2, s\rangle \Rightarrow \langle S_2, s\rangle} \quad \text{if } \mathcal{B}[\![b]\!]\,(s) = \mathbf{ff}$$

$$\textsc{Call}\ \frac{-}{\langle m(), s\rangle \Rightarrow \langle S, s\rangle} \quad \text{if } m \text{ is declared as } m\,\{S\} \text{ in } T$$

Figure 1 SOS rules for **Rec**.

▶ **Definition 2.5** (**Rec** SOS). *The* structural operational semantics *(SOS) of* **Rec** *is defined by the rules given in Figure 1.*

The structural operational semantics of **Rec** *induces* a finite-trace semantics in terms of the sequences of states that are traversed from an initial to a final configuration when executing a given statement in the SOS. Let $\mathbf{State}^+$ denote the set of all non-empty, finite sequences of states. Formally, we define a function $\mathcal{S}_{sos}[\![S]\!] : \mathbf{Stm} \to 2^{\mathbf{State}^+}$, i.e., a function such that $\mathcal{S}_{sos}[\![S]\!] \subseteq \mathbf{State}^+$ for any statement S.

▶ **Definition 2.6** (Induced Finite-Trace Semantics). *Let S be a statement. Then, $\mathcal{S}_{sos}[\![S]\!]$ is defined as the set of (finite) sequences $s_0 \cdot s_1 \cdot \ldots \cdot s_n$ of states for which there are statements $S_0, S_1, \ldots, S_{n-1}$ such that $S_0 = S$, $\langle S_i, s_i\rangle \Rightarrow \langle S_{i+1}, s_{i+1}\rangle$ for all $0 \leq i \leq n-2$, and $\langle S_{n-1}, s_{n-1}\rangle \Rightarrow s_n$.*

Observe that in Definition 2.6 *any* state $s_0 \in \mathbf{State}$ can serve as the initial state of a finite trace. Next we design a "direct-style", denotational finite-trace semantics that *conforms* with the SOS, in the sense that it is equal to the finite-trace semantics induced by the SOS.

3 A Denotational Finite-Trace Semantics for Rec

We define the semantic function $\mathcal{S}_{tr} : \mathbf{Stm} \to 2^{\mathbf{State}^+}$ with the intention that $\mathcal{S}_{tr}[\![S]\!] = \mathcal{S}_{sos}[\![S]\!]$. Unlike $\mathcal{S}_{sos}$, however, $\mathcal{S}_{tr}$ is defined directly, without referring to other semantic rules as SOS does (hence the term "direct-style"). We define $\mathcal{S}_{tr}[\![S]\!]$ in the style of *denotational semantics*, compositionally, by induction on the structure of S, and through defining equations. We let σ range over traces.

Let us define a unary *restriction* operator on trace sets, for any trace set A and Boolean expression $b \in \mathbf{BExp}$, as follows: $A|_b \overset{\text{def}}{=} \{s \cdot \sigma \in A \mid \mathcal{B}[\![b]\!]\,(s) = \mathbf{tt}\}$. It filters out all traces in A whose first state does not satisfy b. Another unary operator on trace sets is defined as $\sharp A \overset{\text{def}}{=} \{s \cdot s \cdot \sigma \mid s \cdot \sigma \in A\}$ which duplicates the first state in each trace in A. Finally, let us define the binary operator on trace sets: $A \frown B \overset{\text{def}}{=} \{\sigma_A \cdot s \cdot \sigma_B \mid \sigma_A \cdot s \in A \wedge s \cdot \sigma_B \in B\}$ which concatenates traces from A with traces from B that agree on the last and first state, respectively, but without duplicating that state, see also [16].

▶ **Definition 3.1** (Denotational Finite-Trace Semantics of **Rec**). *Let $M = \{m_1, \ldots, m_n\}$ be the set of procedure names in* **Rec** *program $\langle S, T\rangle$, where every m_i is declared as $m_i\,\{S_i\}$ in T. We define a helper function $\mathcal{S}^0_{tr}[\![S]\!]_\rho$ that is relativized on an interpretation $\rho : M \to 2^{\mathbf{State}^+}$*

$$\mathcal{S}^0_{tr}[\![\mathbf{skip}]\!]_\rho \overset{\text{def}}{=} \{s \cdot s \mid s \in \mathbf{State}\} \qquad \mathcal{S}^0_{tr}[\![x := a]\!]_\rho \overset{\text{def}}{=} \{s \cdot s[x \mapsto \mathcal{A}[\![a]\!](s)] \mid s \in \mathbf{State}\}$$
$$\mathcal{S}^0_{tr}[\![S_1; S_2]\!]_\rho \overset{\text{def}}{=} \mathcal{S}^0_{tr}[\![S_1]\!]_\rho {}^\frown \mathcal{S}^0_{tr}[\![S_2]\!]_\rho \qquad \mathcal{S}^0_{tr}[\![m_i()]\!]_\rho \overset{\text{def}}{=} \rho(m_i)$$
$$\mathcal{S}^0_{tr}[\![\mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2]\!]_\rho \overset{\text{def}}{=} (\sharp\mathcal{S}^0_{tr}[\![S_1]\!]_\rho)|_b \cup (\sharp\mathcal{S}^0_{tr}[\![S_2]\!]_\rho)|_{\neg b}$$

Figure 2 Finite-trace semantic equations for **Rec**.

of the procedure names, inductively, by the equations given in Figure 2. The duplication of the initial states in the equation for the **if** *statement is needed to remain faithful to the SOS of* **Rec***, which allocates a small step for evaluation of the guard b. We then introduce a semantic function $H : \left(2^{\mathbf{State}^+}\right)^n \to \left(2^{\mathbf{State}^+}\right)^n$ defined as:*

$$H(\rho) \overset{\text{def}}{=} (\sharp\mathcal{S}^0_{tr}[\![S_1]\!]_\rho, \ldots, \sharp\mathcal{S}^0_{tr}[\![S_n]\!]_\rho)$$

Function H is monotonic and continuous in the CPO with bottom $\left(\left(2^{\mathbf{State}^+}\right)^n, \sqsubseteq, \varnothing^n\right)$, where $\sqsubseteq$ denotes point-wise set inclusion. Hence, by the Knaster-Tarski Theorem, it has a least fixed point. Let ρ_0 denote this least fixed point. The denotational finite-trace semantics *of statements S of* **Rec** *is defined relative to this interpretation as: $\mathcal{S}_{tr}[\![S]\!] \overset{\text{def}}{=} \mathcal{S}^0_{tr}[\![S]\!]_{\rho_0}$.*

▶ **Example 3.2.** We execute the statement $x := 2; down()$ with the procedure table from Example 2.3. The program execution (or run) we obtain from an arbitrary state s is:

$$\begin{aligned}
&\langle x := 2; down(), s\rangle \Rightarrow \langle down(), s[x \mapsto 2]\rangle \Rightarrow \\
&\langle \mathbf{if}\ x > 0\ \mathbf{then}\ x := x - 2; down()\ \mathbf{else\ skip}, s[x \mapsto 2]\rangle \Rightarrow \\
&\langle x := x - 2; down(), s[x \mapsto 2]\rangle \Rightarrow \langle down(), s[x \mapsto 0]\rangle \Rightarrow \\
&\langle \mathbf{if}\ x > 0\ \mathbf{then}\ x := x - 2; down()\ \mathbf{else\ skip}, s[x \mapsto 0]\rangle \Rightarrow \langle \mathbf{skip}, s[x \mapsto 0]\rangle \Rightarrow s[x \mapsto 0]
\end{aligned}$$

The finite-trace semantics agrees with the SOS of **Rec**, in the sense that $\mathcal{S}_{tr}[\![S]\!]$ coincides with the finite-trace semantics $\mathcal{S}_{sos}[\![S]\!]$ induced by the SOS, as defined in Definition 2.6.

▶ **Theorem 3.3** (Correctness of Trace Semantics). *For all statements S of* **Rec***, we have:*

$$\mathcal{S}_{tr}[\![S]\!] = \mathcal{S}_{sos}[\![S]\!].$$

4 A Logic over Finite Traces

Our trace logic can be seen as an Interval Temporal Logic [16] with μ-recursion [28], or alternatively, as a temporal μ-calculus with a binary temporal operator corresponding to the chop operation over sets of traces (see, e.g., [32] for a general introduction to μ-calculus).

4.1 Syntax and Semantics of the Logic

The philosophy behind our logic is to have logical counterparts to the statements of the programming language in terms of their finite-trace semantics. For instance, we use binary relation symbols that correspond to the atomic statements, and a chop operator corresponding to sequential composition. This design choice helps to simplify proofs of the properties of the logic and the calculus.

▶ **Definition 4.1** (Logic Syntax). *The* syntax *of the logic of* trace formulas *is defined by the following grammar:*

$$\phi \ ::= \ p \mid R \mid X \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 {}^\frown \phi_2 \mid \mu X.\phi$$

$$\begin{array}{rclrcl}
\|p\|_{\mathcal{V}} & \overset{\text{def}}{=} & \{s \cdot \sigma \mid s \models p\} & \|R\|_{\mathcal{V}} & \overset{\text{def}}{=} & \{s \cdot s' \mid R(s, s')\} \\
\|X\|_{\mathcal{V}} & \overset{\text{def}}{=} & \mathcal{V}(X) & \|\phi_1 \wedge \phi_2\|_{\mathcal{V}} & \overset{\text{def}}{=} & \|\phi_1\|_{\mathcal{V}} \cap \|\phi_2\|_{\mathcal{V}} \\
\|\phi_1 \vee \phi_2\|_{\mathcal{V}} & \overset{\text{def}}{=} & \|\phi_1\|_{\mathcal{V}} \cup \|\phi_2\|_{\mathcal{V}} & \|\phi_1 \frown \phi_2\|_{\mathcal{V}} & \overset{\text{def}}{=} & \|\phi_1\|_{\mathcal{V}} \frown \|\phi_2\|_{\mathcal{V}}
\end{array}$$

$$\|\mu X.\phi\|_{\mathcal{V}} \overset{\text{def}}{=} \bigcap \left\{ \gamma \subseteq \mathbf{State}^+ \;\middle|\; \|\phi\|_{\mathcal{V}[X \mapsto \gamma]} \subseteq \gamma \right\}$$

Figure 3 Finite-trace semantic equations for formulas.

where p ranges over state formulas not further specified here, but assumed to contain at least the Boolean expressions **BExp**, *R ranges over binary relation symbols over states, and X over a set* RVar *of* recursion variables*.*

▶ **Definition 4.2** (Logic Semantics)**.** *The* finite-trace semantics *of a formula ϕ is defined as its denotation $\|\phi\|_{\mathcal{V}} \subseteq \mathbf{State}^+$, relativized on a* valuation *$\mathcal{V}$: RVar $\to 2^{\mathbf{State}^+}$ of the recursion variables, inductively by the equations given in Figure 3, where in the last clause $\mathcal{V}[X \mapsto \gamma]$ denotes the updated valuation.*

One can show that the transformers $\lambda\gamma.\ \|\phi\|_{\mathcal{V}[X \mapsto \gamma]}$ are monotonic functions in the complete lattice $(2^{\mathbf{State}^+}, \subseteq)$ and hence, by Tarski's fixed point theorem for complete lattices [33], they have least and greatest fixed points. In particular, the least fixed point is simultaneously also the least pre-fixed point, hence the defining equation for $\mu X.\phi$. And because it is a fixed point, we have the following result for unfolding fixed point formulas.

▶ **Proposition 4.3** (Fixed Point Unfolding)**.** *Let $\mu X.\phi$ be a formula and $\mathcal{V}$ a valuation. Then: $\|\mu X.\phi\|_{\mathcal{V}} = \|\phi[\mu X.\phi/X]\|_{\mathcal{V}}$.*

Our calculus is based on closed formulas of the logic. Observe that fixed point unfolding preserves closedness. For closed formulas the valuation $\mathcal{V}$ is immaterial to the semantics $\|\phi\|_{\mathcal{V}}$. In this case, we often omit the subscript and simply write $\|\phi\|$.

4.2 Binary Relations

We instantiate the set Rel of binary relation symbols with two specific relations:

$$\begin{array}{rcl}
Id(s, s') & \overset{\text{def}}{\Longleftrightarrow} & s' = s \\
Sb_x^a(s, s') & \overset{\text{def}}{\Longleftrightarrow} & s' = s[x \mapsto \mathcal{A}[\![a]\!]\,(s)]
\end{array}$$

These two relations are used to model **skip** and assignment statements, respectively. The *transitive closure* R^+ of a binary relation R over states is easily defined as a recursive formula in our logic as $R^+ \overset{\text{def}}{\Longleftrightarrow} \mu X.\ (R \vee R \frown X)$.

▶ **Example 4.4.** For any arithmetic expression a let Dec_a be a binary relation symbol interpreted as follows: $Dec_a(s, s') \overset{\text{def}}{\Longleftrightarrow} \mathcal{A}[\![a]\!]\,(s') \leq \mathcal{A}[\![a]\!]\,(s)$. That is, the value of a does not increase between two consecutive states. With this symbol, the formula Dec_a^+ expresses the property that the value of a does not increase throughout the whole execution of a program.

$$\mathsf{stf}(\overline{X}, \textbf{skip}) \overset{\mathsf{def}}{=} \mathit{Id} \qquad \mathsf{stf}(\overline{X}, x := a) \overset{\mathsf{def}}{=} \mathit{Sb}_x^a \qquad \mathsf{stf}(\overline{X}, S_1;\, S_2) \overset{\mathsf{def}}{=} \mathsf{stf}(\overline{X}, S_1) \frown \mathsf{stf}(\overline{X}, S_2)$$

$$\mathsf{stf}(\overline{X}, \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2) \overset{\mathsf{def}}{=} (b \wedge \mathit{Id} \frown \mathsf{stf}(\overline{X}, S_1)) \vee (\neg b \wedge \mathit{Id} \frown \mathsf{stf}(\overline{X}, S_2))$$

$$\mathsf{stf}(\overline{X}, m()) \overset{\mathsf{def}}{=} \begin{cases} \mathit{Id} \frown \mu X_m .\, \mathsf{stf}(\overline{X} \cup \{m\}, S_m) & m \notin \overline{X},\ m\,\{S_m\} \in T \\ \mathit{Id} \frown X_m & \text{otherwise} \end{cases}$$

Figure 4 Definition of strongest trace formula.

4.3 Strongest Trace Formulas

Since our program logic is able to characterize program traces, and not merely pre- and postconditions or intermediate assertions, it is possible to establish a close correspondence between programs and trace formulas. This correspondence is captured by the following – constructive – definition of the *strongest trace formula* $\mathsf{stf}(S)$ for a given program S, which characterizes all terminating traces of S.

For each procedure declaration $m\,\{S_m\}$ in T, we create a fixed point formula, whenever m is called the first time. Subsequent calls to m result in a recursion variable. To achieve this, we parameterize the strongest trace formula function with the already created recursion variables $\overline{X}$. This parameter is initialized to $\varnothing$ and is ignored by all case definitions except the one for a recursive call.

▶ **Definition 4.5** (Strongest Trace Formula). *Let $\langle S, T\rangle$ be a* **Rec** *program. The* strongest trace formula *for S, denoted* $\mathsf{stf}(S)$*, is defined as* $\mathsf{stf}(S) \overset{\mathsf{def}}{=} \mathsf{stf}(\varnothing, S)$*, where* $\mathsf{stf}(X, S)$ *is defined inductively in Figure 4.*

▶ **Example 4.6.** For the program $\mathit{even}()$ with the procedure table of Example 2.2, the strongest trace formula is:

$$\mathit{Id} \frown \mu X_{even}.\Big(\big(x = 0 \,\wedge\, \mathit{Id} \frown \mathit{Sb}_y^1\big) \,\vee$$
$$\big(x \neq 0 \wedge \mathit{Id} \frown \mathit{Sb}_x^{x-1} \frown \mathit{Id} \frown \mu X_{odd}.((x = 0 \wedge \mathit{Id} \frown \mathit{Sb}_y^0) \vee (x \neq 0 \wedge \mathit{Id} \frown \mathit{Sb}_x^{x-1} \frown \mathit{Id} \frown X_{even}))\big)\Big)$$

The binder for X_{odd} can be removed without changing the semantics.

▶ **Example 4.7.** For the program in Example 2.3, the strongest trace formula is:

$$\mathsf{stf}(S) = \mathit{Id} \frown \mu X_{down}.\, \big((x > 0 \wedge \mathit{Id} \frown \mathit{Sb}_x^{x-2} \frown \mathit{Id} \frown X_{down}) \vee (x \leq 0 \wedge \mathit{Id} \frown \mathit{Id})\big).$$

▶ **Theorem 4.8** (Characterisation of Strongest Trace Formula). *Let $\langle S, T\rangle$ be a program of* **Rec***. Then the following holds:* $\|\mathsf{stf}(\varnothing, S)\| \;=\; \mathcal{S}_{tr}[\![S]\!]$.

The proof of Theorem 4.8 requires an inner fixed point induction for which it is advantageous to break down stf into a *modal equation system* mes [23] that preserves the structure of procedure declarations.

▶ **Definition 4.9** (Modal Equation System). *Given a closed trace formula ϕ, the* modal equation system $\mathsf{mes}(\phi)$ *is an open trace formula together with a set of modal equations of the form $X_i = \phi_i$, inductively defined over ϕ as follows:* $\mathsf{mes}(\phi)$ *is just the homomorphism over the abstract syntax, except when $\phi = \mu X.\phi'$: In this case,* $\mathsf{mes}(\phi) \overset{\mathsf{def}}{=} X$*, and a new equation* $X = \mathsf{mes}(\phi')$ *is added.*

The semantics of modal equation systems is defined as in the literature [23], from where we also take the semantic equivalence between ϕ and $\mathsf{mes}(\phi)$.

▶ **Example 4.10.** The modal equation system corresponding to the strongest trace formula in Example 4.6 is: $\mathsf{mes}(\mathsf{stf}(even())) = Id^\frown X_{even}$, where:

$$\begin{aligned} X_{even} &= \big((x = 0 \land Id^\frown Sb_y^1) \lor (x \neq 0 \land Id^\frown Sb_x^{x-1\frown} Id^\frown X_{odd})\big) \\ X_{odd} &= \big((x = 0 \land Id^\frown Sb_y^0) \lor (x \neq 0 \land Id^\frown Sb_x^{x-1\frown} Id^\frown X_{even})\big) \end{aligned}$$

Observe the structural similarity between the formula $Id^\frown X_{even}$ in the context of the defining equations for X_{even} and X_{odd}, and the statement $even()$ in the context of the procedure table T of Example 2.2.

To use the decomposition of a trace formula into a modal equation system, we need to define for each program S an open trace formula corresponding to $\mathsf{mes}(\mathsf{stf}(S))$. This transformation, called $\mathsf{stf}'(S)$, is defined exactly as $\mathsf{stf}(X, S)$ in Definition 4.5 (ignoring the parameter X), except for the case $S = m()$, which is defined as $\mathsf{stf}'(m()) = X_m$.

▶ **Lemma 4.11.** *Let $\langle S, T\rangle$ be a* **Rec** *program and M the procedures declared in T. Let $\rho : M \to 2^{\mathbf{State}^+}$ be an arbitrary interpretation of the procedures $m \in M$, and let $\mathcal{V}_\rho : \mathsf{RVar} \to 2^{\mathbf{State}^+}$ be the (induced) valuation of the recursion variables defined by $\mathcal{V}_\rho(X_m) \overset{\mathsf{def}}{=} \rho(m)$. We have, for all statements S of* **Rec**: $\|\mathsf{stf}'(S)\|_{\mathcal{V}_\rho} = \mathcal{S}^0_{tr}[\![S]\!]_\rho$.

5 A Proof Calculus

We present a proof calculus for our logic in the form of a Gentzen-style deductive proof system, which is *compositional* both in the statement and the formula.

5.1 Definition of the Calculus

To obtain a compositional proof rule for procedure calls, its shape will essentially embody the principle of Fixed Point Induction. For this we need to represent recursion variables in the **Rec** language, whose syntax is extended with a set SVar of *statement variables*, ranged over by Y. We add these as a new category of atomic statements to **Rec**.

To define the semantics of programs in the presence of statement variables, we relativize the finite-trace semantics $\mathcal{S}_{tr}[\![S]\!]_{\mathcal{I}}$ of statements S on *interpretations* $\mathcal{I} : \mathsf{SVar} \to 2^{\mathbf{State}^+}$ of the statement variables, lifted from $\mathcal{S}_{tr}[\![S]\!]$ in the canonical manner.

▶ **Definition 5.1** (Calculus Syntax). Judgments *are of the form $S : \phi$, where S is a* **Rec** *statement, possibly containing statement variables, and ϕ is a closed trace formula. The* sequents *of the calculus are of the shape $\Gamma \vdash S : \phi$, where Γ is a possibly empty set of judgments.*

5.1.1 Rules

The calculus has exactly one rule for each kind of **Rec** statement, except for statement variables which do not occur in initial judgments to be proven, but are only created intermittently in proofs by the (Call) rule. All statement rules are *compositional* in the sense that only the statement S in focus without any context appears in the conclusion.

$$\textsc{Skip}\quad \frac{-}{\Gamma \vdash \mathbf{skip} : \mathit{Id}} \qquad\qquad \textsc{Assign}\quad \frac{-}{\Gamma \vdash x := a : Sb^a_x}$$

$$\textsc{Seq}\quad \frac{\Gamma \vdash S_1 : \phi_1 \quad \Gamma \vdash S_2 : \phi_2}{\Gamma \vdash S_1; S_2 : \phi_1 {}^\frown \phi_2} \qquad \textsc{If}\quad \frac{\Gamma \vdash \mathbf{skip}; S_1 : \neg b \vee \phi \quad \Gamma \vdash \mathbf{skip}; S_2 : b \vee \phi}{\Gamma \vdash \mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2 : \phi}$$

$$\textsc{Unfold}\quad \frac{\Gamma \vdash S : \phi[\mu X.\phi/X]}{\Gamma \vdash S : \mu X.\phi} \qquad\qquad \textsc{Cons}\quad \frac{\Gamma \vdash S : \phi'}{\Gamma \vdash S : \phi} \quad \phi' \models \phi$$

$$\textsc{Call}\quad \frac{\begin{array}{c} Y_m : \phi_m \notin \Gamma \qquad m\,\{S_m\} \in T \\ \Gamma, Y_m : \phi_m \vdash S_m[\mathbf{skip}; Y_m/m(), \mathbf{skip}; Y_{m_1}/m_1(), \ldots, \mathbf{skip}; Y_{m_n}/m_n()] : \phi_m \end{array}}{\Gamma \vdash m() : \mathit{Id} {}^\frown \phi_m}$$

Figure 5 The rules of the proof calculus.

The statement rules and two selected logical rules of the calculus are shown in Figure 5. The remaining logical rules, in particular the axioms, are the standard Gentzen-style ones and are omitted.

To prove the judgment $S : \phi$ for a program $\langle S, T\rangle$, we prove the sequent $\vdash S : \phi$. All rules, except the (Call) rule, leave the antecedent Γ invariant.

We first explain the two logical rules. The (Unfold) rule is based on Proposition 4.3 and is used to unfold fixed point formulas. The consequence rule (Cons) permits to strengthen the trace formula ϕ in the succedent, i.e., the specification of the program under verification. This is typically required to achieve a suitable syntactic form of ϕ, or to strengthen an inductive claim. The rule assumes the existence of an *oracle* for proving the logical entailment between trace formulas.

The (Skip) and (Assign) rules handle the atomic statements, using the two binary relation symbols defined in Section 4.2. The (Seq) rule is a rule for sequential composition. Observe that it is compositional in the sense that *no intermediate state* between S_1 and S_2 needs to be considered.

The (If) rule is a compositional rule for conditional statements. The trace formulas $\neg b \vee \phi$ in the left premise (and $b \vee \phi$ in the right one) might at first appear counter-intuitive. Formula $\neg b \vee \phi$ is read as follows: We need not consider program S_1 for any trace, where $\neg b$ holds in the beginning, because these traces relate to S_2; otherwise, ϕ must hold. A more intuitive notation would be $b \to \phi$, but we refrain from introducing implication in our logic.

Unsurprisingly, the (Call) rule is the most complex. We associate with each declaration of a method m in T a unique statement variable Y_m. The antecedent Γ contains judgments of the form $Y_m : \phi_m$. One can think of the Y_m as a generic continuation of any recursive call to m of which we know that it must conform to its contract ϕ_m. Once this conformance has been established, the fact is memorized in the antecedent Γ. Therefore, the (Call) rule is triggered only when a call to procedure $m()$ is encountered the first time. This is ensured by the condition $Y_m : \phi_m \notin \Gamma$. To avoid having to apply the call rule again to recursive calls of $m()$, all such calls in the body S_m are replaced with $\mathbf{skip}; Y_m$, where the **skip** models unfolding and Y_m is justified by the assumption in Γ. Likewise, any other $Y_{m_i} : \phi_{m_i} \in \Gamma$ triggers an analogous substitution. Now the procedure body $S_m[\ldots]$ in the premise contains at most procedure calls to m' that do not occur in Γ.

If a different judgment than $\mathit{Id}{}^\frown\phi_m$ is to be proven, then rule (Cons) must be applied before (Call) to achieve the required shape.

▶ **Example 5.2.** We prove the judgment $\mathit{even}() : \mathsf{stf}(\mathit{even}())$ for the program from Example 2.2 in Figure 6. We abbreviate the fixed point formula in $\mathsf{stf}(\mathit{even}())$ from Example 4.6 with ϕ_{even}, so that $\mathsf{stf}(\mathit{even}()) = \mathit{Id}{}^\frown\phi_{even}$, the body of $\mathit{even}()$ with S_{even}, and similarly for $\mathit{odd}()$.

$$\textsc{Seq}\;\dfrac{\cdots \vdash \textbf{skip} : \mathit{Id} \qquad \textsc{Cons}\;\dfrac{Y_{even} : \phi_{even}, Y_{odd} : \phi_{odd} \;\vdash\; Y_{even} : \phi_{even}}{Y_{even} : \phi_{even}, Y_{odd} : \phi_{odd} \;\vdash\; Y_{even} : \mu X_{even}.(\cdots {}^\frown \phi_{odd})}}{Y_{even} : \phi_{even}, Y_{odd} : \phi_{odd} \;\vdash\; \textbf{skip}; Y_{even} : \mathit{Id}{}^\frown \mu X_{even}.(\cdots {}^\frown \phi_{odd})}$$

$$\vdots$$

$$\textsc{Cons}\;\dfrac{\textsc{Call}\;\dfrac{Y_{even} : \phi_{even}, Y_{odd} : \phi_{odd} \;\vdash\; S_{odd}[\textbf{skip}; Y_{even}/\mathit{even}()] : \phi_{odd}}{Y_{even} : \phi_{even} \;\vdash\; \mathit{odd}() : \mathsf{stf}(\mathit{odd}())}}{Y_{even} : \phi_{even} \;\vdash\; \mathit{odd}() : \mathit{Id}{}^\frown \mu X_{odd}.((x = 0 \land \mathit{Id}{}^\frown \mathit{Sb}_y^0) \lor (x \neq 0 \land \mathit{Id}{}^\frown \mathit{Sb}_x^{x-1}{}^\frown \mathit{Id}{}^\frown \phi_{even})}$$

$$\vdots$$

$$\textsc{Unfold}\;\dfrac{Y_{even} : \phi_{even} \;\vdash\; S_{even} : \phi'_{even}[\phi_{even}/X_{even}]}{\textsc{Call}\;\dfrac{Y_{even} : \phi_{even} \;\vdash\; S_{even} : \phi_{even}}{\vdash\; \mathit{even}() : \mathsf{stf}(\mathit{even}())}}$$

Figure 6 Proof of $\mathit{even}() : \mathsf{stf}(\mathit{even}())$.

Moreover, we abbreviate:

$$\phi'_{even} = \big(x = 0 \land \mathit{Id}{}^\frown \mathit{Sb}_y^1\big) \lor \big(x \neq 0 \land \\ \mathit{Id}{}^\frown \mathit{Sb}_x^{x-1}{}^\frown \mathit{Id}{}^\frown \mu X_{odd}.((x = 0 \land \mathit{Id}{}^\frown \mathit{Sb}_y^0) \lor (x \neq 0 \land \mathit{Id}{}^\frown \mathit{Sb}_x^{x-1}{}^\frown \mathit{Id}{}^\frown X_{even}))\big)$$

The proof starts with the call rule, followed by an unfold of the fixed point formula ϕ_{even}. We now proceed with the other statement rules simultaneously on S_{even} and the formula on the right until we encounter the call of $\mathit{odd}()$ in S_{even}. Here we would like to apply the call rule to $\mathit{odd}()$, but we do not have ϕ_{odd} on the right, because the unfolding of ϕ_{even} went "too deep". To avoid a lengthy derivation at this point, we use the fact that trace formula on the right is *equivalent* to ϕ_{odd}, and use the consequence rule to obtain it.

Now we descend into $\mathit{odd}()$, similarly as before; however, because the judgment $Y_{even} : \phi_{even}$ is present on the left, the call rule replaces $\mathit{even}()$ in S_{even} with $\textbf{skip}; Y_{even} : \phi_{even}$. Finally, we encounter the statement variable Y_{even}, but again the fixed point formula on the right is "too deep". After a second transformation we close the proof with an axiom.

▶ Remark 5.3. It is easy to derive a rule for loops from the (Call) rule using the encoding $m\,\{\textbf{if}\ b\ \textbf{then}\ S; m()\ \textbf{else skip}\}$ given in Remark 2.4. In a pure loop program no unprocessed recursive call except $m()$ ever occurs in the body, so the (Call) rule is applicable with $\Gamma = \varnothing$. Rule (Call) instantiated to m and a suitable ϕ_m gives $S_m[\textbf{skip}; Y_m/m()] =$ $\textbf{if}\ b\ \textbf{then}\ S; \textbf{skip}; Y_m\ \textbf{else skip}$ in the premise on the right, so its single premise becomes: $Y_m : \phi_m \vdash \textbf{if}\ b\ \textbf{then}\ S; \textbf{skip}; Y_m\ \textbf{else skip} : \phi_m$. Subsequent application of rule (If) yields the two premises $Y_m : \phi_m \vdash \textbf{skip}; S; \textbf{skip}; Y_m : \neg b \lor \phi_m$ and $Y_m : \phi_m \vdash \textbf{skip}; \textbf{skip} : b \lor \phi_m$.. In each premise is a spurious **skip** resulting from evaluating the method call which is only due to the encoding. In addition, the antecedent is not needed to prove the second premise and can be removed. After reordering and simplification, rule (While) is obtained as:

$$\textsc{While}\;\dfrac{\Gamma \vdash \textbf{skip} : b \lor \phi \qquad \Gamma, Y : \phi \vdash \textbf{skip}; S; Y : \neg b \lor \phi}{\Gamma \vdash \textbf{while}\ b\ \textbf{do}\ S : \phi}$$

where Γ contains judgments of the form $Y : \phi$ originating from while loops encountered previously. These are only needed in a proof in the presence of nested loops.

5.1.2 Semantics

▶ **Definition 5.4** (Calculus Semantics). *A judgment $S : \phi$ is termed* valid *in $\mathcal{I}$, denoted $\models_{\mathcal{I}} S : \phi$, whenever $\mathcal{S}_{tr}[\![S]\!]_{\mathcal{I}} \subseteq \|\phi\|$. A sequent $\Gamma \vdash S : \phi$ is termed* valid*, denoted $\Gamma \models S : \phi$, if for every interpretation $\mathcal{I}$, $S : \phi$ is valid in $\mathcal{I}$, whenever all judgments in Γ are valid in $\mathcal{I}$.*

It is not possible to prove a judgment for $m()$ (or any other statement) that is stronger than its strongest trace formula. In this sense, $\mathsf{stf}(m())$ can be seen as a *contract* for m, in fact the *strongest possible* contract. This is captured in the following result:

▶ **Corollary 5.5** (Strongest Trace Formula). *Let S be a statement not involving any statement variables. Then the strongest trace formula $\mathsf{stf}(S)$ of S entails any valid formula for S. That is, if $\models S : \phi$, then $\mathsf{stf}(S) \models \phi$.*

Proof. The result follows directly from Theorem 4.8 and Definition 5.4. ◀

5.2 Soundness and Relative Completeness of the Calculus

Our proof system is *sound*, in the sense that it can only derive valid sequents.

▶ **Theorem 5.6** (Soundness). *The proof system is sound: every derivable sequent is valid.*

Our proof system is *complete*, in the sense that every valid sequent can be derived, *relative* to an oracle, used by rule (Cons), that provides logical entailment between trace formulas.

By Theorem 4.8 and Definition 5.4 we know that every judgment of the shape S : $\mathsf{stf}(S)$ is valid. We next show that all such judgments are derivable in our proof system. Together with Corollary 5.5, we obtain completeness, with the help of the rule (Cons). By definition: $\mathsf{stf}(m()) = \mathit{Id}^\frown \mu X_m.\, \mathsf{stf}(\{m\}, S_m)$, where S_m is the body of m. We abbreviate $\phi_m = \mu X_m.\, \mathsf{stf}(\{m\}, S_m)$ and in the following use $\mathsf{stf}(m()) = \mathit{Id}^\frown \phi_m$ without mentioning it explicitly.

▶ **Theorem 5.7** (Existence of Canonical Proof). *Let $\langle S, T\rangle$ be a* **Rec** *program with n many method declarations in T, and let $\Gamma = \{Y_{m_1} : \phi_{m_1}, \ldots, Y_{m_n} : \phi_{m_n}\}$. Then, the judgment $\Gamma \vdash S[\mathbf{skip}; Y_{m_1}/m_1(), \ldots, \mathbf{skip}; Y_{m_n}/m_n()] : \mathsf{stf}(S)$ is derivable in our calculus.*

▶ **Corollary 5.8** (Relative Completeness). *The proof system is relatively complete: for every* **Rec** *program S without statement variables and every closed formula ϕ, any valid judgment of the form $S : \phi$ is derivable in the proof system.*

Proof. By Theorem 4.8 we know that $\mathsf{stf}(S) \models \phi$, so we can use rule (Cons) to obtain $S : \mathsf{stf}(S)$, which is derivable by Theorem 5.7. ◀

Compared to a typical completeness proof of first-order Dynamic Logic, where the invariant is constructed as equations over the Gödelized program in the loop, the argument is much simpler, because the inductive specification logic is sufficiently expressive to characterize recursive programs (and loops as a special case). First-order quantifiers are not even necessary, so our logic is *not* first-order, even though it is obviously Turing-hard and thus undecidable.

6 From Trace Formulas to Programs

In Section 4.3 we showed that any **Rec** program S can be translated into a trace formula $\mathsf{stf}(S)$ that has the same semantics in terms of traces. Now we look at the other direction: Given a trace formula ϕ, can we construct a *canonical* program $\mathsf{can}(\phi)$ that has the same semantics in terms of traces? In general, this is not possible, as the following example shows:

▶ **Example 6.1.** Consider the trace formula: $Sb^0_y \frown \mu X. \big(Id \vee Sb^{y+1}_y \frown X\big)$. Its semantics are the traces that count y up from 0 to any finite number. It is not possible to model the non-deterministic choice in the fixed point formula directly in **Rec**, because the number of calls is unbounded.

There is a **Rec** program that produces exactly the same traces as the formula above, up to auxiliary variables, for example, $y := 0; m()$, where m is declared as: $m()\ \{\textbf{if}\ (y \leq x)\ \textbf{then}\ y := y + 1; m()\ \textbf{else skip}\}$. However, to transform an arbitrary formula with unbounded non-determinism in the number of calls to an equivalent one with non-deterministic initialization, is difficult and not natural.

6.1 Rec Programs with Non-deterministic Choice

To achieve a *natural* translation from the trace logic to canonical programs, it is easiest to introduce *non-deterministic choice* in the form of a statement **if** $*$ **then** S_1 **else** S_2. The extension of language **Rec** with the corresponding grammar rule is called **Rec***.

The SOS rules for non-deterministic choice are:

$$*\text{-}i \quad \frac{-}{\langle \textbf{if}\ *\ \textbf{then}\ S_1\ \textbf{else}\ S_2, s\rangle \Rightarrow \langle S_i, s\rangle} \quad i \in \{1, 2\}$$

The finite-trace semantics of non-deterministic choice is:

$$\mathcal{S}_{tr}[\![\textbf{if}\ *\ \textbf{then}\ S_1\ \textbf{else}\ S_2]\!] \overset{\mathsf{def}}{=} \sharp\mathcal{S}_{tr}[\![S_1]\!] \cup \sharp\mathcal{S}_{tr}[\![S_2]\!]$$

The extension of Theorem 3.3 for non-deterministic choice is completely straightforward. Likewise, the theory of strongest trace formulas is easy to extend:

$$\mathsf{stf}(\textbf{if}\ *\ \textbf{then}\ S_1\ \textbf{else}\ S_2) \overset{\mathsf{def}}{=} Id \frown \mathsf{stf}(S_1) \vee Id \frown \mathsf{stf}(S_2)$$

It is easy to adapt the proof of Theorem 4.8. The corresponding calculus rule is:

$$\textsc{If-}* \quad \frac{\Gamma \vdash \textbf{skip}; S_1 : \phi \quad \Gamma \vdash \textbf{skip}; S_2 : \phi}{\Gamma \vdash \textbf{if}\ *\ \textbf{then}\ S_1\ \textbf{else}\ S_2 : \phi}$$

It is also easy to extend the proofs of Theorem 5.6 and Theorem 5.7. A more problematic aspect of the translation to canonical programs concerns formulas of the form $\phi_1 \wedge \phi_2$, because there is no natural programming construct that computes the intersection of traces. But in Definition 4.5 general conjunction is not required, so without affecting the results in previous sections we can restrict the syntax of trace formulas in Definition 4.1 to $p \wedge \phi$.

A final issue are the programs that characterize a trace formula of the form p with semantics $\|p\| = \mathbf{State}^+|_p$. A program that produces such traces requires a **havoc** statement that resets *all* variables to an arbitrary value. This goes beyond non-deterministic choice quite a bit, but luckily, it is not required: As seen above, trace formulas of the form p occur only as subformulas of $p \wedge \phi$. Further, formulas of the form $p \vee \phi$ occur only as intermediate formulas in derivations, and nowhere else. Altogether, for the purpose of mapping formulas to programs, we can leave out the production for p. The grammar in Definition 4.1 is thus simplified to:

$$\phi \ ::= \ Id \mid Sb^a_x \mid X \mid p \wedge \phi \mid \phi \vee \psi \mid \phi \frown \psi \mid \mu X.\phi$$

In addition, we assume without loss of generality that all recursion variables in a trace formula have unique names.

6.2 Canonical Programs

▶ **Definition 6.2** (Canonical Program). *Let ϕ be trace formula. The* canonical program *for ϕ, denoted* $\mathsf{can}(\phi) = \langle S_\phi, T_\phi \rangle$, *is inductively defined as follows:*

$$\begin{aligned}
\mathsf{can}(Id) &\overset{\text{def}}{=} \langle \mathbf{skip}, \epsilon \rangle & \mathsf{can}(p \wedge \phi) &\overset{\text{def}}{=} \langle \mathbf{if}\ p\ \mathbf{then}\ S_\phi\ \mathbf{else}\ \mathbf{diverge}, T_\phi \rangle \\
\mathsf{can}(Sb^a_x) &\overset{\text{def}}{=} \langle x := a, \epsilon \rangle & \mathsf{can}(\phi \vee \psi) &\overset{\text{def}}{=} \langle \mathbf{if}\ *\ \mathbf{then}\ S_\phi\ \mathbf{else}\ S_\psi, T_\phi\, T_\psi \rangle \\
\mathsf{can}(\phi \frown \psi) &\overset{\text{def}}{=} \langle S_\phi; S_\psi, T_\phi\, T_\psi \rangle & \mathsf{can}(\mu X.\phi) &\overset{\text{def}}{=} \langle m_X(), T_\phi\, \{m_X\{S_\phi\}\} \rangle \\
\mathsf{can}(X) &\overset{\text{def}}{=} \langle m_X(), \epsilon \rangle
\end{aligned}$$

The definition contains the statement **diverge**. It is definable in the **Rec** language as **diverge** $\overset{\text{def}}{=}$ *abort*(), with the declaration *abort*{*abort*()}. We assume that any table T_ϕ contains the declaration of procedure *abort*() when needed.

▶ **Example 6.3.** We translate the formula in Example 6.1:

$$\mathsf{can}(Sb^0_y \frown \mu X.(Id \vee Sb^{y+1}_y \frown X)) = \langle y := 0; m_X(),\ T \rangle$$

where $T = m_X\, \{\mathbf{if}\ *\ \mathbf{then\ skip\ else}\ y := y + 1; m_X()\}$.

▶ **Proposition 6.4.** *We have* $\mathcal{S}_{tr}[\![\mathbf{diverge}]\!] = \varnothing$.

▶ **Proposition 6.5.** *Let ϕ be an open trace formula, let T'_ϕ be declarations of its unbound recursion variables, and let* $\mathsf{can}(\phi) = \langle S_\phi, T_\phi \rangle$. *Then* $\langle S_\phi, T_\phi\, T'_\phi \rangle$ *is a well-defined* **Rec*** *program.*

Evaluation of procedure calls and Boolean guards introduce *stuttering steps* as compared to the corresponding logical operators of least fixed point recursion and disjunction, respectively. Hence, canonical programs obtained from a formula ϕ are statements, whose trace semantics is equal to the one of ϕ, but *modulo stuttering*: The two trace sets are equal when abstracting away the stuttering steps. Further, we say that statement S_1 *refines* statement S_2, written $S_1 \preceq S_2$, when $\mathcal{S}_{tr}[\![S_1]\!] \subseteq \mathcal{S}_{tr}[\![S_2]\!]$.

▶ **Definition 6.6** (Stuttering Equivalence). *Let $\tilde{\sigma}$ be the* stutter-free *version of a trace σ, i.e., where any subtrace of the form $s \cdot s \cdots s$ has been replaced with s. Define $\widetilde{A} = \{\tilde{\sigma} \mid \sigma \in A\}$. We say two trace sets A, B are* stutter-equivalent, *written $A \,\tilde{=}\, B$, if $\widetilde{A} = \widetilde{B}$.*

It is easy to see that $A = B$ implies $A \,\tilde{=}\, B$. Let $\tilde{\models}$ and $\tilde{\preceq}$ denote entailment between formulas and refinement between statements, respectively, both modulo stuttering equivalence.

Unsurprisingly, the characterization of canonical programs resembles the one of strongest trace formulas, however, modulo stuttering equivalence.

▶ **Theorem 6.7** (Characterisation of Canonical Program). *Let ϕ be a closed trace formula, and let* $\mathsf{can}(\phi) = \langle S_\phi, T_\phi \rangle$. *Then* $\mathcal{S}_{tr}[\![S_\phi]\!] \,\tilde{=}\, \|\phi\|$.

Finally, we can establish that $\mathsf{stf}(\cdot)$ and $\mathsf{can}(\cdot)$ form a Galois connection w.r.t. the partial orders $\tilde{\models}$ on formulas and $\tilde{\preceq}$ on statements.

▶ **Corollary 6.8.** *Let ϕ be a closed trace formula, and let* $\mathsf{can}(\phi) = \langle S_\phi, T_\phi \rangle$. *Then, for every statement S, we have:* $\mathsf{stf}(S) \,\tilde{\models}\, \phi$ *iff* $S \,\tilde{\preceq}\, S_\phi$.

Proof. By using Theorem 4.8 and Theorem 6.7. ◀

7 Related Work

We do not discuss higher-order logical frameworks [4, 27], because they serve a different purpose: Their expressiveness can be used to mechanize verification frameworks, including syntax, semantics, and calculus. For example, a mechanization of contract-based deductive verification is in [34]. One could do that also for the approach in the present paper.

Stirling [32, p. 528, footnote 2] suggests that the μ-calculus can be generalized to non-unary predicates, but does not develop this possibility further. In [24], Müller-Olm proposes a modal fixed point logic with chop, which can characterize any context-free process up to bisimulation or simulation. The logic is shown to be strictly more expressive than the modal μ-calculus. Lange & Somla [21] relate propositional dynamic logic over context-free programs with Müller-Olm's logic and show the former to be equi-expressive with a fragment of the latter. Fredlund et al. [13] present a verification tool for the ERLANG language based on first-order μ-calculus with actions [9]. Rosu et al. [6, 31] propose *matching logic* (ML) as a unifying logic for specifying and reasoning about the static structure and dynamic behaviour of programs. It uses patterns and constraints to uniformly represent mathematical domains, data types, and transition systems, whose properties can be reasoned about using one proof system. Our trace logic bares certain similarities with ML, both allowing recursion, but our logic is syntax-driven and has a fixed semantics in the domain of program traces, while ML is a semantic approach. We are not aware of a similar result as Corollary 6.8 in ML.

In contrast to the above mentioned work, we separate programs from fixed point formulas, and relate them in the form of judgments. Our logic has a single binary operator Sb_x^a over arithmetic expressions a and program variables x, together with the chop operator $\frown$. The latter models composition of binary relations in the denotational semantics. Our logic is sufficient to characterize any **Rec** program. Specifically, μ-formulas can serve as contracts of recursive procedures. More importantly, our approach leads to a *compositional* calculus, where all rules but the consequence rule are analytic.

Kleene algebra with tests (KAT) [19] are an equational algebraic theory that has been shown to be as expressive as propositional while programs. They were mechanized in an interactive theorem prover [30] and are able to express at least Hoare-style judgments [20]. The research around KATs focuses on *propositional* while programs: we are not aware of results relating KAT with recursive stateful programs. Specifically, our result that procedure contracts can be expressed purely in terms of trace formulas (Theorem 4.8) has not been obtained by algebraic approaches.

Expressive trace-based specification languages are rare in program verification. The trace logic of Barthe et al. [2] is a many-sorted *first-order logic*, equipped with an arithmetic theory of explicit trace positions to define program semantics. It is intended to model program verification in first-order logic for processing in automated theorem provers. Like $\mathcal{S}_{tr}[\![]\!]$, their program semantics is compositional; however, it uses explicit time points instead of algebraic operators. An extension of Hoare logic with trace specifications is presented by Ernst et al. [11]. Hoare triples are extended with regular expressions recording events emitted before the execution of the command and the events emitted by its execution. Our trace logic is more expressive. Also the first-order *temporal logic of nested words* (NWTL) of Alur et al. [1] permits to specify certain execution patterns within symbolic traces. It is orthogonal to our approach, being based on nested event pairs and temporal operators, instead of least fixed points and chop. NWTL is equally expressive over nested words as first-order logic. The intended computational model is not **Rec**, but non-deterministic Büchi automata over nested words for which it is complete. *Temporal stream logic* of Finkbeiner et al. [12], like our trace logic, has state updates Sb_x^t (with a different syntax) and state predicates, but it is based on linear temporal logic and has no fixed points or chop. Again, the intended computational model is an extension of Büchi automata, the verification target are FPGA programs.

Cousot & Cousot [8] define a trace-based semantics for modal logics where (infinite) traces are equipped with past, present, and future. Their main focus is to relate model checking and static analysis to abstract interpretation – the trace-based semantics is the basis for it. In contrast, our paper relates a computation model to a logic. Like our semantics, theirs is compositional and the operators mentioned in their paper could inspire abstractions of our trace logic, cf. Section 8.1 below.

Nakata & Uustalu [25] present a trace-based co-inductive operational semantics with chop for an imperative programming language with loops. Following up on this work, [10] extended the approach to an asynchronous concurrent language, but neither of these uses fixed points, so that the specification language is incomplete. Also the calculus is not compositional.

Closest to the present work is our earlier work on trace-based deductive verification [5], where we used a similar trace logic than here. However, neither the semantics nor the calculus were compositional. Furthermore, we proved soundness there, but left completeness as an open question.

8 Future Work

8.1 Abstract Specifications and Extension of Recursive Programs

It is desirable to formulate specifications in a more abstract manner than the programs whose behavior they are intended to capture. For example, if we reinstate the atomic trace formula p in our logic, we can easily express the set of all finite traces as $\|\mathit{true}\| = \mathbf{State}^+$. Then we can define a binary connective as $\phi \cdot\cdot \psi \overset{\mathsf{def}}{=} \phi \frown \mathit{true} \frown \psi$ as "any finite (possibly, empty) computation may occur between ϕ and ψ". For example, the trace formula $\mu X.(X \cdot\cdot X)$ expresses that a procedure m_X calls itself at least twice recursively, at the beginning and at the end of its body, respectively.

A more general approach to introduce non-determinism to the logic is to define for a Boolean expression b a *binary relation* R_b with semantics $R_b(s,s') \overset{\mathsf{def}}{\Longleftrightarrow} \mathcal{B}[\![b]\!]\,(s) = \mathbf{tt}$. The atomic formula p is then *definable* as: $p \overset{\mathsf{def}}{=} R_p \frown R^+_{\mathit{true}}$. The advantage of basing p on a binary relation is that it is more easily represented as a canonical program. To this end, we introduce an atomic statement **havoc** with the trace semantics $\mathcal{S}_{tr}[\![\mathbf{havoc}]\!] = \{s \cdot s' \mid s, s' \in \mathbf{State}\}$, i.e. in any given state s, executing **havoc** results in an arbitrary successor state. The proof rule for **havoc** is the axiom $\Gamma \vdash \mathbf{havoc} : R_{\mathit{true}}$ and obviously $\mathsf{stf}(\mathbf{havoc}) \overset{\mathsf{def}}{=} R_{\mathit{true}}$. Then p is characterized by $\mathsf{can}(p) \overset{\mathsf{def}}{=} \langle \mathbf{if}\ p\ \mathbf{then}\ \mathit{havoc}()\ \mathbf{else\ diverge}, T\rangle$, where T contains the declaration $\mathit{havoc}\ \{\mathbf{if}\ *\ \mathbf{then\ havoc\ else\ skip}; \mathit{havoc}()\}$.

Interestingly, R_{true} (and, therefore, **havoc** on the side of programs) permits to define *concatenation* of trace formulas $\phi \cdot \psi$ with the obvious semantics:

$$\|\phi \cdot \psi\| \overset{\mathsf{def}}{=} \{s_0 \cdots s_n \cdot s'_0 \cdots s'_m \mid s_0 \cdots s_n \in \|\phi\|,\, s'_0, \cdots s'_m \in \|\psi\|\} = \|\phi \frown R_{\mathit{true}} \frown \psi\|$$

8.2 Proving Consequence of Trace Formulas

In general this is a difficult problem that requires fixed point induction, but the derivations needed in practice might be relatively simple, as the following example shows.

▶ **Example 8.1.** Consider the two trace formulas:

$$\begin{aligned}\mathsf{stf}(\mathit{down}()) &= \mathit{Id}^\frown \mu X_{\mathit{down}}.\big((x > 0 \wedge \mathit{Id}^\frown \mathit{Sb}_x^{x-2\frown}\mathit{Id}^\frown X_{\mathit{down}}) \vee (x \le 0 \wedge \mathit{Id}^\frown \mathit{Id})\big)\\ \mathit{Dec}_x^+ &= \mu X_{\mathit{dec}}.(\mathit{Dec}_x{}^\frown X_{\mathit{dec}} \vee \mathit{Dec}_x)\end{aligned}$$

from Examples 2.3 and 4.7, respectively. We expect the trace formula implication $\mathsf{stf}(\mathit{down}()) \Rightarrow \mathit{Id}^\frown \mathit{Dec}_x^+$ to be provable, because of Theorem 4.8.

It turns out that the following fixed point induction rule and consequence rule, combined with straightforward *first-order* consequence and logic rules, are sufficient to prove the claim:

$$\text{FP-IND}\ \frac{\Gamma \vdash \phi \Rightarrow \psi}{\Gamma \vdash \mu X.\phi \Rightarrow \mu X.\psi} \qquad \text{CONS-LEFT}\ \frac{\Gamma, \phi \vdash \psi \qquad \Gamma, \phi' \vdash \psi'}{\Gamma, \phi^\frown \phi' \vdash \psi^\frown \psi'}$$

In fact, the proof is considerably shorter than proving the judgment $\mathit{down}() : \mathit{Id}^\frown \mathit{Dec}_x^+$ in the calculus of Section 5, which is as well possible.

A proof system for trace formula implication that can prove the above as well as many other non-trivial examples is given in [17].

8.3 Non-terminating Programs

Our results so far are limited to *terminating* programs, i.e. to sets of finite traces. To extend the calculus with a termination measure, such that a proof of $S : \phi$ not only shows correctness of S relative to ϕ, but also ensures it produces only finite traces, is easy.

However, the trace-based setup permits, in principle, also to prove properties of *non-terminating* programs. To this end, it is necessary to extend the logic with operators whose semantics contains infinite traces. One obvious candidate are *greatest fixed points* [32]. The downside to this approach is that nested fixed points of opposite polarity are difficult to understand, as is well known from μ-calculus. Is there a restriction of mixed fixed point formulas that *naturally* corresponds to a certain class of programs? The theory of strongest trace formulas and canonical programs might guide the search for such fragments.

9 Conclusion

We presented a fixed point logic that characterizes recursive programs with non-deterministic guards. Both, programs and formulas have the same kind of trace semantics, which, like the calculus for proving judgments, is fully compositional in the sense that the definitions and rules embody no context. The faithful embedding of programs into a logic seems to suggest that we merely replace one execution model (programs) with another (trace formulas). So why is it worth having such an expressive specification logic? We can see four reasons:

First, the logic renders itself naturally to extension and abstraction that cannot be easily mimicked by programs or that are much less natural for programs. This is corroborated by the discussion in Section 8.1, but also by the case of conjunction: It is trivial to add conjunction $\phi \wedge \psi$ to the trace logic and to a calculus for trace formulas, but conjunction has no natural program counter-part. Yet it permits to specify certain hyper-properties, i.e., properties relating sets of traces.

Second, the logic offers reasoning patterns that are easily justified algebraically, such as projection, replacement of equivalents, strengthening, distribution, etc., that are not obvious in the realm of programs.

Third, the concept of *strongest trace formula* leads to a characterization of valid judgments and, thereby, enables a simple completeness proof.

And finally, the duality between programs and formulas permits to prove judgments by freely mixing two styles of reasoning: with the rules of the calculus in Figure 5, or using a calculus for the consequence of trace formulas. For example, a judgment such as $\mathit{down}() : \mathit{Id}^\frown \mathit{Dec}_x^+$ can be proved as in Example 8.1 or directly with the rules in Figure 5, but also by *mixing* both styles.

A perhaps surprising feature of our trace logic is the fact that no explicit notion of *procedure contract* is required to achieve procedure-modular verification: Instead, strongest trace formulas and statement variables are employed. This results in a novel (Call) rule that works with *symbolic continuations* realized by statement variables.

References

1 Rajeev Alur, Marcelo Arenas, Pablo Barceló, Kousha Etessami, Neil Immerman, and Leonid Libkin. First-order and temporal logics for nested words. *Log. Methods Comput. Sci.*, 4(11):1–44, 2008. doi:10.2168/LMCS-4(4:11)2008.

2 Gilles Barthe, Renate Eilers, Pamina Georgiou, Bernhard Gleiss, Laura Kovács, and Matteo Maffei. Verifying relational properties using trace logic. In Clark W. Barrett and Jin Yang, editors, *Formal Methods in Computer Aided Design, FMCAD*, pages 170–178, San Jose, CA, USA, 2019. IEEE. doi:10.23919/FMCAD.2019.8894277.

3 Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. ACSL: ANSI/ISO C Specification. Technical Report Version 1.17, CEA and INRIA, 2021. URL: https://frama-c.com/download/frama-c-acsl-implementation.pdf.

4 Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development—Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, Berlin Heidelberg, 2004. doi:10.1007/978-3-662-07964-5.

5 Richard Bubel, Dilian Gurov, Reiner Hähnle, and Marco Scaletta. Trace-based deductive verification. In *Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2023)*, volume 94 of *EPiC Series in Computing*, pages 73–95. EasyChair, 2023. doi:10.29007/VDFD.

6 Xiaohong Chen, Dorel Lucanu, and Grigore Rosu. Matching logic explained. *Journal of Logical and Algebraic Methods in Programming*, 120:100638, 2021. doi:10.1016/J.JLAMP.2021.100638.

7 Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

8 Patrick Cousot and Radhia Cousot. Temporal abstract interpretation. In Mark N. Wegman and Thomas W. Reps, editors, *Proc. 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, USA*, pages 12–25. ACM, 2000. doi:10.1145/325694.325699.

9 Mads Dam and Dilian Gurov. μ-calculus with explicit points and approximations. *J. of Logic and Computation*, 12(2):255–269, April 2002. doi:10.1093/LOGCOM/12.2.255.

10 Crystal Chang Din, Reiner Hähnle, Einar Broch Johnsen, Violet Ka I Pun, and Silvia Lizeth Tapia Tarifa. Locally abstract, globally concrete semantics of concurrent programming languages. In Cláudia Nalon and Renate Schmidt, editors, *Proc. 26th Intl. Conf. on Automated Reasoning with Tableaux and Related Methods*, volume 10501 of *LNCS*, pages 22–43, Cham, September 2017. Springer. doi:10.1007/978-3-319-66902-1_2.

11 Gidon Ernst, Alexander Knapp, and Toby Murray. A Hoare logic with regular behavioral specifications. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation, 11th Intl. Symp., ISoLA, Rhodes, Greece, Proc. Part I*, volume 13701 of *LNCS*, pages 45–64. Springer, 2022. doi:10.1007/978-3-031-19849-6_4.

12 Bernd Finkbeiner, Felix Klein, Ruzica Piskac, and Mark Santolucito. Temporal stream logic: Synthesis beyond the bools. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification: 31st Intl. Conf., CAV, Part I*, volume 11561 of *LNCS*, pages 609–629, New York City, NY, USA, 2019. Springer. doi:10.1007/978-3-030-25540-4_35.

13 Lars-Åke Fredlund, Dilian Gurov, Thomas Noll, Mads Dam, Thomas Arts, and Gennady Chugunov. A verification tool for Erlang. *Journal of Software Tools for Technology Transfer*, 4(4):405–420, August 2003. `doi:10.1007/S100090100071`.

14 Dilian Gurov and Reiner Hähnle. An expressive trace logic for recursive programs. *CoRR*, abs/2411.13125, 2024. `doi:10.48550/arXiv.2411.13125`.

15 Reiner Hähnle and Marieke Huisman. Deductive verification: from pen-and-paper proofs to industrial tools. In Bernhard Steffen and Gerhard Woeginger, editors, *Computing and Software Science: State of the Art and Perspectives*, volume 10000 of *LNCS*, pages 345–373. Springer, Cham, Switzerland, 2019.

16 Joseph Y. Halpern and Yoav Shoham. A propositional modal logic of time intervals. *Journal of the ACM*, 38(4):935–962, 1991. `doi:10.1145/115234.115351`.

17 Niklas Heidler and Reiner Hähnle. A Sequent Calculus for Trace Formula Implication. *arXiv CoRR*, May 2025. `doi:10.48550/arXiv.2505.03693`.

18 C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In Erwin Engeler, editor, *Symposium on Semantics of Algorithmic Languages*, volume 188 of *Lecture Notes in Mathematics*, pages 102–116. Springer, Berlin, Heidelberg, 1971. `doi:10.1007/BFb0059696`.

19 Dexter Kozen. Kleene algebra with tests. *ACM Transactions on Programming Languages and Systems*, 19(3):427–443, 1997. `doi:10.1145/256167.256195`.

20 Dexter Kozen. On Hoare logic and Kleene algebra with tests. *ACM Transactions on Computational Logic*, 1(1):60–76, 2000. `doi:10.1145/343369.343378`.

21 Martin Lange and Rafał Somla. Propositional dynamic logic of context-free programs and fixpoint logic with chop. *Information Processing Letters*, 100(2):72–75, 2006. `doi:10.1016/J.IPL.2006.04.019`.

22 Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, Daniel M. Zimmerman, and Werner Dietl. *JML Reference Manual*, May 2013. Draft revision 2344. URL: `http://www.eecs.ucf.edu/~leavens/JML//OldReleases/jmlrefman.pdf`.

23 Angelika Mader. *Verification of modal properties using Boolean equation systems.* PhD thesis, Technical University Munich, 1997.

24 Markus Müller-Olm. A modal fixpoint logic with chop. In *Theoretical Aspects of Computer Science (STACS 1999)*, volume 1563 of *LNCS*, pages 510–520, Berlin Heidelberg, 1999. Springer. `doi:10.1007/3-540-49116-3_48`.

25 Keiko Nakata and Tarmo Uustalu. Trace-based coinductive operational semantics for While. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5674 of *LNCS*, pages 375–390, Berlin Heidelberg, 2009. Springer. `doi:10.1007/978-3-642-03359-9_26`.

26 Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer.* Undergraduate Topics in Computer Science. Springer, London, 2007. `doi:10.1007/978-1-84628-692-6`.

27 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, Berlin Heidelberg, 2002. `doi:10.1007/3-540-45949-9`.

28 David Michael Ritchie Park. Finiteness is mu-ineffable. *Theoretical Computer Science*, 3(2):173–181, 1976. `doi:10.1016/0304-3975(76)90022-0`.

29 Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60–61:17–139, 2004.

30 Damien Pous. Kleene algebra with tests and Coq tools for while programs. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving, 4th Intl. Conf. ITP, Rennes, France*, volume 7998 of *LNCS*, pages 180–196. Springer, 2013. `doi:10.1007/978-3-642-39634-2_15`.

31 Grigore Rosu. Matching logic. *Logical Methods in Computer Science*, 13(4), 2017. `doi:10.23638/LMCS-13(4:28)2017`.

32 Colin Stirling. Modal and temporal logics. In *Handbook of Logic in Computer Science (Vol. 2): Background: Computational Structures*, pages 477–563, USA, 1993. Oxford University Press, Inc.

33 Alfred Tarski. A lattice-theoretical fixedpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

34 David von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13):1173–1214, 2001. `doi:10.1002/CPE.598`.

A Selected Proofs

A.1 Strongest Trace Formula

Proof of Lemma 4.11. The proof proceeds by induction on the structure of S.

Case $S = \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2$. By the induction hypothesis, $\|\mathsf{stf}(S_i)\|_{\mathcal{V}_\rho} = \mathcal{S}^0_{tr}[\![S_i]\!]_\rho$ for $i = 1, 2$. We have:

$$\begin{array}{rl} & \left\|\mathsf{stf}'(\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2)\right\|_{\mathcal{V}_\rho} \\ = & \left\|(b \wedge \mathit{Id}^\frown \mathsf{stf}'(S_1)) \vee (\neg b \wedge \mathit{Id}^\frown \mathsf{stf}'(S_2))\right\|_{\mathcal{V}_\rho} \\ = & \left\|(b \wedge \mathit{Id}^\frown \mathsf{stf}'(S_1))\right\|_{\mathcal{V}_\rho} \cup \left\|(\neg b \wedge \mathit{Id}^\frown \mathsf{stf}'(S_2))\right\|_{\mathcal{V}_\rho} \\ = & (\sharp \left\|\mathsf{stf}'(S_1)\right\|_{\mathcal{V}_\rho})|_b \cup (\sharp \left\|\mathsf{stf}'(S_2)\right\|_{\mathcal{V}_\rho})|_{\neg b} \\ = & (\sharp \mathcal{S}^0_{tr}[\![S_1]\!]_\rho)|_b \cup (\sharp \mathcal{S}^0_{tr}[\![S_2]\!]_\rho)|_{\neg b} \\ = & \mathcal{S}^0_{tr}[\![\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2]\!]_\rho \end{array}$$

Case $S = m()$. We have:

$$\begin{array}{rl} & \left\|\mathsf{stf}'(m())\right\|_{\mathcal{V}_\rho} \\ = & \|X_m\|_{\mathcal{V}_\rho} \\ = & \mathcal{V}_\rho(X_m) \\ = & \rho(m) \\ = & \mathcal{S}^0_{tr}[\![m()]\!]_\rho \end{array}$$

◀

Proof of Theorem 4.8. Since $\mathsf{stf}'(S)$ and $\mathsf{stf}(X, S)$ are defined identically, except for the case $S = m()$, the proof is the same as for Lemma 4.11, except for that case:

Case $S = m()$. We only sketch the proof here. We translate the formula $\mathsf{stf}(\varnothing, m())$ into a modal equation system $\mathsf{mes}(\mathsf{stf}(\varnothing, m()))$. This results in the formula X_m, defined in the context of a system of modal equations: for each fixed point operator μX_i in $\mathsf{stf}(\varnothing, m())$, there is an equation $X_i = \mathit{Id}^\frown \mathsf{stf}'(S_i)$, whenever m_i is declared as $m_i \,\{S_i\}$ in T. Next, from the standard semantics of modal equation systems, and by Lemma 4.11, it follows that the least solution $\mathcal{V}_0$ of the modal equation system is equal (on the names of the procedures called recursively by m) to the valuation $\mathcal{V}_{\rho_0}$ induced by the interpretation ρ_0 defined by the procedure table T. Finally, by Lemma 4.11, we have:

$$\begin{array}{rl} & \|\mathsf{stf}(\varnothing, m())\| \\ = & \|X_m\|_{\mathcal{V}_0} \\ = & \|X_m\|_{\mathcal{V}_{\rho_0}} \\ = & \mathcal{V}_{\rho_0}(X_m) \\ = & \rho_0(m) \\ = & \mathcal{S}^0_{tr}[\![m()]\!]_{\rho_0} \\ = & \mathcal{S}_{tr}[\![m()]\!] \end{array}$$

◀

A.2 Soundness of the Calculus

For the proof we need to relate traces restricted by a condition b to trace formulas. In the following proposition, the intuition for $\|\neg b \vee \phi\|$ is that it ignores any trace, that is, it is trivially true for any trace, where b does not hold in the beginning.

▶ **Proposition A.1** (State formulas in judgments). *Let b be a Boolean expression, ϕ a trace formula. Then $(\mathcal{S}_{tr}[\![S]\!])|_b \subseteq \|\phi\|$ iff $\mathcal{S}_{tr}[\![S]\!] \subseteq \|\neg b \vee \phi\|$.*

Proof. "Only If" direction: Assume $(\mathcal{S}_{tr}[\![S]\!])|_b \subseteq \|\phi\|$ and there is a trace $s \cdot \sigma \in \mathcal{S}_{tr}[\![S]\!]$ that is not in $\|\neg b \vee \phi\| = \|\neg b\| \cup \|\phi\|$, hence, $s \cdot \sigma \notin \|\neg b\|$ and $s \cdot \sigma \notin \|\phi\|$. But $s \cdot \sigma \notin \|\neg b\|$ implies $s \cdot \sigma \in (\mathcal{S}_{tr}[\![S]\!])|_b \subseteq \|\phi\|$: contradiction.

"If" direction: Assume $\mathcal{S}_{tr}[\![S]\!] \subseteq \|\neg b \vee \phi\|$ and there is a trace $s \cdot \sigma \in (\mathcal{S}_{tr}[\![S]\!])|_b$ that is not in $\|\phi\|$. From the assumption and $(\mathcal{S}_{tr}[\![S]\!])|_b \subseteq \mathcal{S}_{tr}[\![S]\!]$ we obtain $s \cdot \sigma \in \|\neg b \vee \phi\|$. However, since $\mathcal{B}[\![b]\!](s) = \mathbf{tt}$ we must have in fact $s \cdot \sigma \in \|\phi\|$: contradiction. ◀

Proof of Theorem 5.6. The proof system is sound, since every rule of the system is *locally sound*, in the sense that its conclusion is valid whenever all its premises are valid. We shall prove local soundness of each rule. Without loss of generality, we ignore Γ in most cases.

Rule IF. Let $\mathcal{I}$ be an arbitrary interpretation. Using Proposition A.1, we have:

$$\begin{array}{ll}
 & \models_{\mathcal{I}} \mathbf{skip}; S_1 : \neg b \vee \phi \text{ and } \models_{\mathcal{I}} \mathbf{skip}; S_2 : b \vee \phi \\
\Leftrightarrow & \mathcal{S}_{tr}[\![\mathbf{skip}; S_1]\!]_{\mathcal{I}} \subseteq \|\neg b \vee \phi\| \text{ and} \\
 & \mathcal{S}_{tr}[\![\mathbf{skip}; S_2]\!]_{\mathcal{I}} \subseteq \|b \vee \phi\| \\
\Leftrightarrow & \sharp\mathcal{S}_{tr}[\![S_1]\!]_{\mathcal{I}} \subseteq \|\neg b \vee \phi\| \text{ and } \sharp\mathcal{S}_{tr}[\![S_2]\!]_{\mathcal{I}} \subseteq \|b \vee \phi\| \\
\Leftrightarrow & (\sharp\mathcal{S}_{tr}[\![S_1]\!]_{\mathcal{I}})|_b \subseteq \|\phi\| \text{ and } (\sharp\mathcal{S}_{tr}[\![S_2]\!]_{\mathcal{I}})|_{\neg b} \subseteq \|\phi\| \\
\Leftrightarrow & ((\sharp\mathcal{S}_{tr}[\![S_1]\!]_{\mathcal{I}})|_b \cup (\sharp\mathcal{S}_{tr}[\![S_2]\!]_{\mathcal{I}})|_{\neg b}) \subseteq \|\phi\| \\
\Leftrightarrow & \mathcal{S}_{tr}[\![\mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2]\!]_{\mathcal{I}} \subseteq \|\phi\| \\
\Leftrightarrow & \models_{\mathcal{I}} \mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2 : \phi
\end{array}$$

where we use that $\mathcal{S}_{tr}[\![\mathbf{skip}; S]\!]_{\mathcal{I}} = \sharp\mathcal{S}_{tr}[\![S]\!]_{\mathcal{I}}$, and therefore:

$$\begin{array}{ll}
 & \models_{\mathcal{I}} \mathbf{skip}; S_1 : \neg b \vee \phi \text{ and } \models_{\mathcal{I}} \mathbf{skip}; S_2 : b \vee \phi \\
\Leftrightarrow & \models_{\mathcal{I}} \mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2 : \phi
\end{array}$$

Rule CALL. For the proof of the rule we employ the principle of Fixed Point Induction. To simplify the presentation, we assume there is only one procedure $m()$, declared as $m\,\{S_m\}$ in T. The general case follows from Bekič's Principle. The notation $\rho[m \mapsto \gamma]$ specifies the interpretation that is identical to ρ, except for $\rho(m) = \gamma$.

$$\begin{array}{ll}
 & Y_m : \phi_m \models S_m[\mathbf{skip}; Y_m/m()] : \phi_m \\
\Leftrightarrow & \forall \mathcal{I}.\ (\mathcal{S}_{tr}[\![Y_m]\!]_{\mathcal{I}} \subseteq \|\phi_m\| \\
 & \quad \Rightarrow \mathcal{S}_{tr}[\![S_m[\mathbf{skip}; Y_m/m()]]\!]_{\mathcal{I}} \subseteq \|\phi_m\|) \\
\Leftrightarrow & \forall \gamma.\ (\gamma \subseteq \|\phi_m\| \Rightarrow \mathcal{S}^0_{tr}[\![S_m]\!]_{\rho[m \mapsto \sharp\gamma]} \subseteq \|\phi_m\|) \\
\Leftrightarrow & \forall \gamma.\ (\sharp\gamma \subseteq \|Id^\frown\phi_m\| \Rightarrow \sharp\mathcal{S}^0_{tr}[\![S_m]\!]_{\rho[m \mapsto \sharp\gamma]} \subseteq \|Id^\frown\phi_m\|) \\
\Leftrightarrow & \forall \gamma.\ (\gamma \subseteq \|Id^\frown\phi_m\| \Rightarrow \sharp\mathcal{S}^0_{tr}[\![S_m]\!]_{\rho[m \mapsto \gamma]} \subseteq \|Id^\frown\phi_m\|) \\
\Rightarrow & \rho_0(m) \subseteq \|Id^\frown\phi_m\| \\
\Leftrightarrow & \mathcal{S}^0_{tr}[\![m()]\!]_{\rho_0} \subseteq \|Id^\frown\phi_m\| \\
\Leftrightarrow & \mathcal{S}_{tr}[\![m()]\!] \subseteq \|Id^\frown\phi_m\| \\
\Leftrightarrow & \models m() : Id^\frown\phi_m
\end{array}$$

◀

A.3 Completeness of the Calculus

Proof of Theorem 5.7. The proof proceeds by induction on the structure of S. However, In the case of a call the statement does not necessarily get smaller, because the body of m is expanded. So we need to argue that the induction is well-founded. Indeed, the lexicographic order on $\langle N - |\Gamma|, |S|\rangle$ (obviously, the first component is never negative) always decreases. Since Γ is irrelevant for all cases except $S = m()$, we simplify the claim accordingly for these.

Case $S =$ if b then S_1 else S_2. Assume by the induction hypothesis that $\vdash S_1 : \mathsf{stf}(S_1)$ and $\vdash S_2 : \mathsf{stf}(S_2)$ can be proven. We also use that $\phi \models (p \vee (\neg p \wedge \phi))$ is a valid consequence. We have:

$$\begin{array}{ll}
& \vdash \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 : \mathsf{stf}(\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2) \\
\Leftrightarrow & \vdash \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 : (b \wedge \mathit{Id}^\frown\mathsf{stf}(S_1)) \vee \\
& \qquad\qquad\qquad\qquad (\neg b \wedge \mathit{Id}^\frown\mathsf{stf}(S_2)) \\
\Leftarrow & \{\text{By rule (If) combined with rule (Or)}\} \\
& \vdash \textbf{skip}; S_1 : \neg b \vee (b \wedge \mathit{Id}^\frown\mathsf{stf}(S_1)) \text{ and} \\
& \vdash \textbf{skip}; S_2 : b \vee (\neg b \wedge \mathit{Id}^\frown\mathsf{stf}(S_2)) \\
\Leftarrow & \{\text{By rule (Cons)}\} \\
& \vdash \textbf{skip}; S_1 : \mathit{Id}^\frown\mathsf{stf}(S_1) \text{ and } \vdash \textbf{skip}; S_2 : \mathit{Id}^\frown\mathsf{stf}(S_2) \\
\Leftrightarrow & \{\text{By rule (Seq) combined with rule (Skip) and} \\
& \text{the induction hypothesis}\} \\
& \mathsf{true}
\end{array}$$

Case $S = m()$. There are two subcases: either $m \in \{m_1, \ldots, m_n\}$ or not. In the first case, $S[\textbf{skip}; Y_{m_1}/m_1(), \ldots, \textbf{skip}; Y_{m_n}/m_n()] = \textbf{skip}; Y_m$.

In the second case, $S[\textbf{skip}; Y_{m_1}/m_1(), \ldots, \textbf{skip}; Y_{m_n}/m_n()] = m()$. In both cases, we have $\mathsf{stf}(m()) = \mathit{Id}^\frown\phi_m$.

Subcase skip; Y_m. We have to prove the judgment $\Gamma \vdash \textbf{skip}; Y_m : \mathit{Id}^\frown\phi_m$. Using rules (Seq) and (Skip), this reduces to $\Gamma \vdash Y_m : \phi_m$. Because of the assumption $m \in \{m_1, \ldots, m_n\}$ we have $Y_m : \phi_m \in \Gamma$, so the proof is finished.

Subcase $m()$. We have to prove the judgment $\Gamma \vdash m() : \mathit{Id}^\frown\phi_m$. Rule (Call) is applicable due to assumption $m \notin \{m_1, \ldots, m_n\}$. Let

$$S'_m \stackrel{\mathsf{def}}{=} S_m[\textbf{skip}; Y_m/m(), \textbf{skip}; Y_{m_1}/m_1(), \ldots, \textbf{skip}; Y_{m_n}/m_n()]$$

The obtained premise yields the new claim to prove:

$$\Gamma \cup \{Y_m : \phi_m\} \vdash S'_m : \mu X_m. \mathsf{stf}(\{m\}, S_m)$$

We apply rule (Unfold) on the right and obtain:

$$\Gamma \cup \{Y_m : \phi_m\} \vdash S'_m : \mathsf{stf}(\{m\}, S_m)[\phi_m/X_m]$$

By induction and by the definition of $\mathsf{stf}(S)$ we finish the proof up to subgoals of the form (for some $m' \neq m$):

$$\Gamma \cup \{Y_m : \phi_m\} \vdash m'() : \mathsf{stf}(\{m\}, m')[\phi_m/X_m]$$

Unfortunately, the structure of the formula on the right does not conform to $\mathsf{stf}(m')$ as required. But, observing that $\|\mathsf{stf}(S)\| = \|\mathsf{stf}(\overline{X}, S)\|$, as well as soundness of unfolding, we can use Cons to obtain:

$$\Gamma \cup \{Y_m : \phi_m\} \vdash m'() : \mathsf{stf}(m')$$

This follows from the induction hypothesis, because of $N - |\Gamma| > N - |\Gamma \cup \{Y_m : \phi_m\}|$. ◀

A.4 Canonical Programs

Proof of Proposition 6.5. We need to show that for any call $m_X()$ in $\mathsf{can}(\phi)$ there is exactly one declaration of m_X in T_ϕ. This is a straightforward structural induction. ◀

▶ **Lemma A.2.** *Let ϕ be an open trace formula not containing fixed point binders ($\mu X.$), and let $\mathsf{can}(\phi) = \langle S_\phi, T_\phi \rangle$. Then, we have: $\mathcal{S}^0_{tr}[\![S_\phi]\!]_\rho \,\widehat{=}\, \|\phi\|_{\mathcal{V}_\rho}$ for all interpretations $\rho : M \to 2^{\mathbf{State}^+}$ of the procedures M declared in T_ϕ, and (induced) valuations $\mathcal{V}_\rho : \mathsf{RVar} \to 2^{\mathbf{State}^+}$ defined by $\mathcal{V}_\rho(X_m) \overset{\mathsf{def}}{=} \rho(m)$.*

Proof. We proceed by structural induction on ϕ.

Case $\phi = p \wedge \psi$. By the induction hypothesis, $\mathcal{S}^0_{tr}[\![S_\psi]\!]_\rho \,\widehat{=}\, \|\psi\|_{\mathcal{V}_\rho}$.

$$\begin{aligned}
&\mathcal{S}^0_{tr}[\![\mathbf{if}\ p\ \mathbf{then}\ S_\psi\ \mathbf{else}\ \mathbf{diverge}]\!]_\rho\\
&\qquad = (\sharp\mathcal{S}^0_{tr}[\![S_\psi]\!]_\rho)|_p \cup (\sharp\mathcal{S}^0_{tr}[\![\mathbf{diverge}]\!]_\rho)|_{\neg p}\\
&\qquad = (\sharp\mathcal{S}^0_{tr}[\![S_\psi]\!]_\rho)|_p \qquad \text{(Proposition 6.4)}\\
&\qquad = \mathbf{State}^+|_p \cap \sharp\mathcal{S}^0_{tr}[\![S_\psi]\!]_\rho \,\widehat{=}\, \mathbf{State}^+|_p \cap \mathcal{S}^0_{tr}[\![S_\psi]\!]_\rho\\
&\qquad \widehat{=}\, \mathbf{State}^+|_p \cap \|\psi\|_{\mathcal{V}_\rho} \qquad \text{(Induction hypothesis)}\\
&\qquad = \|p \wedge \psi\|_{\mathcal{V}_\rho}
\end{aligned}$$

Case $\phi = \phi_1 \vee \phi_2$. By the induction hypothesis, we have that $\mathcal{S}^0_{tr}[\![S_{\phi_1}]\!]_\rho \,\widehat{=}\, \|\phi_1\|_{\mathcal{V}_\rho}$ and $\mathcal{S}^0_{tr}[\![S_{\phi_2}]\!]_\rho \,\widehat{=}\, \|\phi_2\|_{\mathcal{V}_\rho}$. Therefore,

$$\begin{aligned}
\mathcal{S}^0_{tr}[\![\mathbf{if}\ *\ \mathbf{then}\ S_{\phi_1}\ \mathbf{else}\ S_{\phi_2}]\!]_\rho &= \sharp\mathcal{S}^0_{tr}[\![S_{\phi_1}]\!]_\rho \cup \sharp\mathcal{S}^0_{tr}[\![S_{\phi_2}]\!]_\rho\\
\widehat{=}\ \mathcal{S}^0_{tr}[\![S_{\phi_1}]\!]_\rho \cup \mathcal{S}^0_{tr}[\![S_{\phi_2}]\!]_\rho &\,\widehat{=}\, \|\phi_1\|_{\mathcal{V}_\rho} \cup \|\phi_2\|_{\mathcal{V}_\rho} \qquad \text{(Ind. hyp.)}\\
&= \|\phi_1 \vee \phi_2\|_{\mathcal{V}_\rho}
\end{aligned}$$

Case $\phi = X$. We have:

$$\mathcal{S}^0_{tr}[\![m_X()]\!]_\rho = \rho(m_X) = \mathcal{V}_\rho(X) = \|X\|_{\mathcal{V}_\rho}$$ ◀

Proof of Theorem 6.7. By structural induction on ϕ. We only sketch the proof idea here. An argument can be made similar to the one in the last case of the proof of Theorem 4.8, by referring to the modal equation system corresponding to ϕ, and using Lemma A.2 to generalise the treatment to open formulas ϕ and statements that contain calls to procedures not declared in T_ϕ. Proposition 6.5 ensures that the program on the left is well-defined. Compositionality of the proof is justified by Bekič's Principle. ◀