A Dynamic Logic with Traces and Coinduction

Richard Bubel[1], Crystal Chang Din[1], Reiner Hähnle[1], and Keiko Nakata[2]

[1] Department of Computer Science, Technische Universität Darmstadt, Germany
{bubel,crystald,haehnle}@cs.tu-darmstadt.de
[2] FireEye Dresden, Germany
keiko.nakata@fireeye.com

Abstract. Dynamic Logic with Traces and Coinduction is a new program logic that has an explicit syntactic representation of both programs and their traces. This allows to prove properties involving programs as well as traces. Moreover, we use a coinductive semantics which makes it possible to reason about non-terminating programs and infinite traces, such as controllers and servers. We develop a sound sequent calculus for our logic that realizes symbolic execution of the programs under verification. The calculus has been developed with the goal of automation in mind. One of the novelties of the calculus is a coinductive invariant rule for while loops that is able to prove termination as well as non-termination.

1 Introduction

In this paper we define a new program logic that allows to relate programs and their traces explicitly, as well as to prove coinductive properties. In a nutshell, given a program p and a syntactic representation Θ of a set of traces of p, we build a first-order formula Ψ over Θ, giving rise to a *trace modality formula* of the form $[\![p]\!]\Psi$. Given a possibly infinite trace τ, the semantic judgment $\tau \models [\![p]\!]\Psi$ expresses that any possible trace of p extending τ must be one of the traces characterized by Ψ.

We support *coinductive reasoning*, that is, a program p needs not to terminate and a formula Ψ describes a set of possibly infinite traces. The motivation is that many practically relevant programs, for example, servers and controllers, are *designed* not to terminate. Clearly it is important to express and prove properties about such programs. Relating (abstract) programs to traces is also possible in temporal logics such as CTL* [1], but our approach is based on an expressive first-order dynamic logic [2] over an imperative programming language with standard datatypes. We aim to specify and verify complex, functional properties of the target programs. Our long-term goal is to implement the logic presented here in a verification tool such as KeY [3] that allows highly automated formal verification of real software.

There are already several extensions of first-order dynamic logic that permit to reason about temporal properties of programs [4–6] and even about trace modalities [7–9]. Our work differs from all of these approaches in two important aspects: first, we include an explicit syntactic representation of traces in our logic. This means that (potentially infinite) traces occur as syntactic entities inside formulas, not only in the semantics. Second, we use this *explicit trace representation* to enable reasoning about non-terminating programs in coinductive style. Infinite traces and coinduction allow to express and to prove in a natural manner properties of programs that are designed not to terminate.

A program logic with explicit traces for coinductive reasoning was introduced in [10, 11], however, in a more abstract setting than here: the assertion language is partially left open, the proof rules are highly non-determinstic, the notion of state is implicit.

In the present paper we merge the two lines of research just sketched ([4–9] and [10, 11]): we build a program logic that serves as a basis for *practical* reasoning over coinductive traces and proof rules. The main contributions are as follows: In Sect. 2 we introduce the syntax of *Dynamic Logic with Coinductive Traces* (DLTC) and motivate its design. In Sect. 3 we provide a formal semantics for DLTC and illustrate some of its properties. In Sect. 4 we present a sequent calculus for DLTC, and discuss soundness and completeness. The proof rules of DLTC follow the design principle employed in the state-of-art verification system KeY [3]: program rules are deterministic except for the loop invariant rule and together realize a symbolic execution engine that can eliminate programs from trace modality formulas. Also, state changes are recorded by explicit substitutions called updates. There are several new aspects to the calculus: in contrast to [10,11], we carefully distinguish between *state invariants* and *trace invariants*. This makes coinductive loop invariant reasoning modular and comprehensible. To the best of our knowledge, we present the first invariant rule for while loops that allows to prove termination as well as non-termination. In contrast to [3, 9], state updates are applied not only to state formulas, but also to explicit traces. In Sect. 5 we illustrate how the calculus is used in practice.

In this paper, we focus on a sequential target language, but our ultimate goal is to reason mechanically about real-life, concurrent programs that are designed not to terminate. We see the present work as a first step towards extending the verification tool KeY-ABS [12] for the concurrent modelling language ABS [13] to coinductive reasoning about complex properties of services. The scope of the present paper, however, is to lay the foundations.

2 Dynamic Logic with Traces and Coinduction

The verification target is a simple sequential imperative programming language, whose syntax is specified by the following context-free grammar over well-typed statements *stmt*, (boolean) expressions (*bexp*) *exp*, and program variables ℓ:

$$\begin{array}{ll} \mathit{stmt} ::= \ \ell \ \texttt{=} \ \mathit{exp} \mid \mathit{stmt}\texttt{;}\mathit{stmt} \mid \texttt{if}\ \mathit{bexp}\ \texttt{then}\ \mathit{stmt}\ \texttt{fi} \mid \texttt{while}\ \mathit{bexp}\ \texttt{do}\ \mathit{stmt}\ \texttt{od} \\ \mathit{bexp} ::= \ \mathit{exp}\ \mathit{brel}\ \mathit{exp} \qquad\qquad \mathit{brel} ::= \texttt{==} \mid \texttt{<} \mid \texttt{>} \mid \leq \mid \geq \\ \mathit{exp} ::= \ \mathit{exp}\ \texttt{+}\ \mathit{exp} \mid \mathit{exp}\ \texttt{-}\ \mathit{exp} \mid \mathit{exp}\ \texttt{*}\ \mathit{exp} \mid \mathit{exp}\ \texttt{/}\ \mathit{exp} \mid \ell \mid 0 \mid 1 \mid \cdots \\ \ell ::= \ \mathit{identifier} \end{array}$$

In Listing. 1.1, we illustrate our syntax with a non-terminating program.

$$\ell = 0;\ \texttt{while}\ \ell \geq 0\ \texttt{do}\ \ell = \ell + 1\ \texttt{od}$$

Listing 1.1. An example

We distinguish carefully between program variables $\ell \in \mathsf{PV}$, and logical (first-order) variables $x \in \mathsf{LV}$. Both may appear in logic terms, but only the latter can be quantified over and only the former may appear in programs. $\mathsf{Var} = \mathsf{PV} \cup \mathsf{LV}$ is the set of all variables. Variables have type integer and are evaluated in $\mathbb{N}$. We assume a standard first-order signature with function symbols f and predicate symbols P (including the arithmetic operators appearing in programs), each with an arity. The actual signature is unimportant and left out as a parameter from the following definitions. The syntax of dynamic logic with traces and coinduction (DLTC) is defined inductively by the following grammars. We start with terms t, state updates u, and first-order formulas φ:

$$\begin{aligned}
t &::= \ exp \mid x \mid f\texttt{(}t\texttt{,}\ldots\texttt{,}t\texttt{)} \mid \{u\}\ t \\
&\mid\ \ \texttt{if (}\varphi\texttt{) then (}t\texttt{) else (}t\texttt{)} \mid \texttt{if (}\Psi\texttt{) then (}t\texttt{) else (}t\texttt{)} \\
u &::= \ \ell \texttt{ := } t \mid \ell \texttt{ := } t\texttt{,}u \\
\varphi &::= \ P\texttt{(}t\texttt{,}\ldots\texttt{,}t\texttt{)} \mid \neg\ \varphi \mid \varphi \wedge \varphi \mid \exists x\texttt{.}\varphi \mid \{u\}\ \varphi \\
x &::= \ identifier \qquad f ::= identifier \qquad P ::= identifier \mid brel
\end{aligned}$$

The first three production rules for terms are obvious. The fourth applies *state updates* u on terms. State updates can be seen as explicit substitutions that correspond to a single state transition in a Kripke structure. We need them to specify states explicitly in traces. The last two rules define conditional terms where the guard is either a first-order formula φ or a trace modality formula Ψ, as defined below. Hence, formulas can appear inside terms. *Atomic state updates*, $\ell := t$, specify the value change of a single program variable. These can be combined into finite update sequences. The rules for first-order formulas are standard, except that we permit to apply updates on formulas, similar as for terms. We will freely use logical connectives such as $\vee$ and $\rightarrow$ and the truth constants true, false, which are definable.

Next we introduce an explicit notation for sets of traces Θ. To facilitate reasoning about traces, the language for describing traces is carefully designed to match the target programming language. We chose a minimal trace language for ease of presentation. If necessary, it can be easily extended. If the target programming language introduces new concepts this is in general necessary. For example, if we would add concurrency, an extended notion of trace is required.

$$\Theta ::= \ \ell \texttt{ := } t \mid \Theta \ast\ast\ \Theta \mid \Theta^{<\omega} \mid \Theta^{\omega} \mid \lceil\varphi\rceil \mid \texttt{finite} \mid \texttt{infinite}$$

The first rule extends the current trace with a single transition corresponding to an atomic update or an assignment statement. The second rule features the "chop" operator from [10, 11] and divides a trace into two parts, where the first part ends with the same state with which the second part. This corresponds to sequential composition, where the second part starts execution in the final state of the run of the first part. A final state does not exist when running the first part diverges—the semantics of the chop operator is carefully crafted to take non-termination into account. The next two rules add finite and infinite iteration to traces, corresponding to terminating and non-terminating loops, respectively. Iteration is not the same as concatenation, but rather of the form $\Theta ** \Theta ** \cdots$, expressing that the final state of one loop iteration is identical to the first state in the following round. The next rule lifts state formulas to traces. The intention is that $\lceil\varphi\rceil$ holds exactly in a trace of length one whose only state satisfies φ. Lifting state formulas makes it possible to express complex functional properties with traces or to represent the value of a guard expression. Finally, there are literals that represent any finite or infinite trace.

Example 1. Let $\ell \in \mathsf{PV}$, $n \in \mathsf{LV}$. Intuitively, the trace formula $\mathrm{Attain}(\ell, n) \equiv \mathsf{finite} ** \lceil \ell \mathtt{==} n \rceil ** \mathsf{infinite}$ holds in any infinite trace that contains a state where the value of ℓ attains the value assigned to n.

There is no trace construct that corresponds to branching. Disjunction and even quantification over traces can be achieved with *trace modality formulas*:

$$\begin{array}{lll}\Psi ::= & P(t, \ldots, t) & \text{where no program variables occur in } t \\ \mid & \neg\,\Psi \mid \Psi \wedge \Psi \mid \exists x\,.\Psi & \\ \mid & \Theta & \\ \mid & [\![\, stmt \,]\!]\ \Psi \mid \{u\}\ \Psi & \end{array}$$

In the following we use implication $\Psi \rightarrow \Phi$ and universal quantification $\forall x.\Psi$ as abbreviations for $\neg\Psi \vee \Phi$ resp. $\neg\exists x.\neg\Psi$. Other standard operators like $\vee$, $\leftrightarrow$ are used in a similar fashion.

In Sect. 3 we will evaluate trace modality formulas Ψ relative to a given trace τ. As it is undefined in which state in τ a program variable in Ψ is evaluated, we simply omit them from trace modality formulas which are not trace formulas. Program variables may, however, occur inside trace formulas, where their position defines the state in which they are evaluated. The formula $[\![stmt]\!]\,\Psi$ expresses that any possible trace of *stmt* extending τ must be one of the traces characterized by Ψ. To specify that the value of a program variable ℓ occurring in *stmt* has the value 42 in the final state one writes $[\![stmt]\!]\,(\mathsf{finite}{**}\lceil\ell\mathtt{==}42\rceil)$, which additionally requires that τ is finite and running *stmt* from the last state of τ is terminating. With this in mind, the usual partial and total correctness modalities $[\cdot]\cdot$ and $\langle\cdot\rangle\cdot$ of dynamic logic [2] can be expressed as $[s]\varphi \equiv [\![s]\!](\mathsf{finite} \rightarrow \mathsf{finite}{**}\lceil\varphi\rceil)$ and $\langle s\rangle\varphi \equiv [\![s]\!](\mathsf{finite} ** \lceil\varphi\rceil)$, respectively.

Example 2. We continue Ex. 1 by defining the trace modality formula

$$\Psi_0 \equiv \mathsf{finite} \rightarrow [\![p]\!]\forall n.\mathrm{Attain}(\ell, n) \tag{1}$$

where p is the program from Listing 1.1. Intuitively, it should hold, as it expresses that in all the infinite traces of p the program variable ℓ attains each $n \in \mathbb{N}$ in some state.

One might ask why we have both finite and $\mathsf{infinite}$ in our trace language since it is possible to define $\mathsf{infinite} \equiv \neg\mathsf{finite}$. Note, however, that our trace language is not closed with respect to logical connectives. For example, we cannot write something like $\Theta ** \neg\mathsf{finite}$. This is intentional, as it simplifies the calculus and stratifies the definition of traces and trace modality formulas. Finally, we introduce abbreviations for trace modality formulas: $\mathsf{any} \equiv \mathsf{True} \equiv \mathsf{finite} \vee \mathsf{infinite}$ and $\mathsf{none} \equiv \mathsf{False} \equiv \neg\mathsf{any} \equiv \neg\mathsf{True}$.

3 Semantics

A *trace* is a potentially infinite non-empty sequence of states σ, where $\sigma : \mathsf{PV} \to \mathbb{N}$. The syntax of traces is specified by: $\tau ::= \langle\sigma\rangle \mid \tau \curvearrowright \sigma$.

The equation should be read coinductively, so that a trace may be finite or infinite. Traces grow to the right, by appending latest states, i.e. $\tau \curvearrowright \sigma$. This matches the syntax of updates, where most recent updates are appended to the right. The angular brackets create a singleton trace from a given state. We define three functions on traces. The function "last" takes a finite trace and returns its latest, i.e., right-most, state. Formally,

$$\mathrm{last}(\tau) = \begin{cases} \sigma & \text{if } \tau = \langle\sigma\rangle \\ \sigma_n & \text{if } \tau = \langle\sigma_0\rangle \curvearrowright \sigma_1 \curvearrowright \ldots \curvearrowright \sigma_n \end{cases}$$

It is undefined on infinite traces. Given two traces τ and τ', both of which may be finite or infinite, $\tau \cdot \tau'$ denotes their concatenation.[3]

When reasoning about traces of sequentially composed programs the presentation is simplified a lot by the *chop* function $\underline{**}$, which can be illustrated by the following diagram:

$$\begin{array}{c} \overbrace{\langle\sigma_0\rangle \curvearrowright \cdots \curvearrowright \sigma_n \curvearrowright \sigma_{n+1} \curvearrowright \cdots}^{r;s} \\ || \\ \underbrace{\langle\sigma_0\rangle \curvearrowright \cdots \curvearrowright \sigma_n}_{r} \ \underline{**}\ \underbrace{\langle\sigma_n\rangle \curvearrowright \sigma_{n+1} \curvearrowright \cdots}_{s} \end{array}$$

We want to characterize the traces resulting from sequential composition of programs r and s. Assume that r has a finite trace with final state σ_n (below left). Sequential composition requires the first state of a trace of s to be σ_n as well (below right). In the resulting trace of $r;s$ (on top), one of the σ_n is cut out ("chopped"), by definition, that is the last state of the trace of r. The formal definition is as follows:

$$\tau \,\underline{**}\, \tau' = \begin{cases} \tau & \text{if } \tau \text{ is infinite} \\ \tau' & \text{if } \tau = \langle\sigma\rangle \\ (\langle\sigma_0\rangle \curvearrowright \sigma_1 \curvearrowright \ldots \curvearrowright \sigma_{n-1}) \cdot \tau' & \text{if } \tau = \langle\sigma_0\rangle \curvearrowright \sigma_1 \curvearrowright \ldots \curvearrowright \sigma_n \end{cases}$$

With the help of traces we can give a formal semantics to DLTC. The semantic definitions are presented in the same sequence as the syntactic constructs in

[3] When τ is infinite, $\tau \cdot \tau'$ is τ.

$$
\begin{array}{lcl}
\mathrm{val}_{D,\sigma,\beta}(\ell = e) & = & \langle\sigma\rangle \curvearrowright \sigma[\ell \mapsto \mathrm{val}_{D,\sigma,\beta}(e)] \\
\mathrm{val}_{D,\sigma,\beta}(r; s) & = & \begin{cases} \tau \mathbin{\underline{**}} (\mathrm{val}_{D,\mathrm{last}(\tau),\beta}(s)) & \text{if } \tau = \mathrm{val}_{D,\sigma,\beta}(r) \text{ is finite} \\ \mathrm{val}_{D,\sigma,\beta}(r) & \text{otherwise} \end{cases} \\
\mathrm{val}_{D,\sigma,\beta}(\mathtt{if}\ e\ \mathtt{then}\ s\ \mathtt{fi}) & = & \begin{cases} \mathrm{val}_{D,\sigma,\beta}(s) & \text{if } \mathrm{val}_{D,\sigma,\beta}(e) = \mathrm{tt} \\ \langle\sigma\rangle & \text{otherwise} \end{cases} \\
\mathrm{val}_{D,\sigma,\beta}(\mathtt{while}\ e\ \mathtt{do}\ s\ \mathtt{od}) & = & \begin{cases} \langle\sigma\rangle & \text{if } \mathrm{val}_{D,\sigma,\beta}(e) = \mathrm{ff} \\ \tau & \text{if } \mathrm{val}_{D,\sigma,\beta}(e) = \mathrm{tt}, \tau = \mathrm{val}_{D,\sigma,\beta}(s),\ \tau \text{ is infinite} \\ \tau \mathbin{\underline{**}} (\mathrm{val}_{D,\mathrm{last}(\tau),\beta}(\mathtt{while}\ e\ \mathtt{do}\ s\ \mathtt{od})) & \\ \quad \text{if } \mathrm{val}_{D,\sigma,\beta}(e) = \mathrm{tt},\ \tau = \mathrm{val}_{D,\sigma,\beta}(s),\ \tau \text{ is finite} & \end{cases}
\end{array}
$$

Fig. 1. Program semantics

$$
\begin{array}{ll}
\mathrm{val}_{D,\sigma,\beta}(\ell := t) = \lambda\sigma'.\langle\sigma'[\ell \mapsto \mathrm{val}_{D,\sigma,\beta}(t)]\rangle & \\
\mathrm{val}_{D,\sigma,\beta}(\ell := t, u) = \lambda\sigma'.\langle\sigma''\rangle \cdot (\mathrm{val}_{D,\sigma'',\beta}(u)(\sigma'')) & \text{where } \sigma'' = \sigma'[\ell \mapsto \mathrm{val}_{D,\sigma,\beta}(t)]
\end{array}
$$

Fig. 2. Semantics: Updates

Sect. 2. The semantics of the programming language and the dynamic logic is given by a valuation function $\mathrm{val}_{D,\rho,\beta}$, which sends a syntactic construct to its meaning. It is parameterised over the domain D, as well as the valuation function β on logical constants and variables. The parameter ρ represents either a state or a trace. The following coinductive definitions of $\mathrm{val}_{D,\rho,\beta}$ are simultaneous.

We start with programs. Fig. 1 gives their semantics. The semantics of a program s on an initial state σ is given by the trace which records all the intermediate states in the run of s from σ. A terminating program run produces a finite trace and a non-terminating program run produces an infinite trace. A trace necessarily contains an initial state, hence it is non-empty. Running an assignment $\ell = e$ produces a doubleton consisting of the initial state and the final state. The valuation function $\mathrm{val}_{D,\sigma,\beta}(e)$ evaluates the expression e in the state σ. For a sequence $r; s$, we first run r on the initial state σ. If the resulting trace τ is finite, then it must have a last state σ' from which we obtain a trace of s. The chop operator combines both traces. If r diverges, then τ is infinite, and we return that as the result, because s is not reached. The semantics of conditional statements is obvious. The semantics of a while statement is obtained from iterating a sequence of conditional evaluations of the body. If the guard e evaluates to falsity, then the entire run terminates immediately. If e evaluates to truth, then the loop body s is run. We have two possible cases. If the run of s diverges, then we return the infinite trace τ produced by that run. Otherwise, the trace τ resulting from the run of s is finite, and we continue with the next iteration from its last state. The definition is coinductive, allowing the body to be iterated infinitely, producing a potentially infinite trace like this (assuming $\sigma(\ell) = 3$):

$$
\begin{array}{l}
\mathrm{val}_{D,\sigma,\beta}(\ell = 0;\ \textbf{while}\ \ell \geq 0\ \textbf{do}\ \ell = \ell + 1\ \textbf{od}) \\
\quad = \langle\{\ell \mapsto 3\}\rangle \curvearrowright \{\ell \mapsto 0\} \curvearrowright \{\ell \mapsto 1\} \curvearrowright \{\ell \mapsto 2\} \curvearrowright \{\ell \mapsto 3\} \curvearrowright \ldots
\end{array}
$$

$$
\begin{array}{rcl}
\mathrm{val}_{D,\tau,\beta}(\ell := e) & \text{iff} & \tau = \langle\sigma\rangle \curvearrowright \sigma' \text{ and } \sigma' = \sigma[\ell \mapsto \mathrm{val}_{D,\sigma,\beta}(e)] \\
\mathrm{val}_{D,\tau,\beta}(\Theta_1 ** \Theta_2) & \text{iff} & \begin{cases} \mathrm{val}_{D,\tau,\beta}(\mathsf{infinite}) \text{ and } \mathrm{val}_{D,\tau,\beta}(\Theta_1), \text{ or} \\ \tau = \tau_1 \underline{**}\, \tau_2 \text{ and } \mathrm{val}_{D,\tau_1,\beta}(\Theta_1) \text{ and } \mathrm{val}_{D,\tau_2,\beta}(\Theta_2) \end{cases} \\
\mathrm{val}_{D,\tau,\beta}(\Theta^{<\omega}) & \text{iff} & \begin{cases} \tau = \langle\sigma\rangle, \text{ or} \\ \tau = \tau_1 \underline{**} \ldots \underline{**}\, \tau_{n-1} \underline{**}\, \tau_n \text{ for some } n \text{ and} \\ \qquad\qquad\qquad\qquad \forall i \leq n.\, \mathrm{val}_{D,\tau_i,\beta}(\Theta) \end{cases} \\
\mathrm{val}_{D,\tau,\beta}(\Theta^{\omega}) & \text{iff} & \tau = \tau_1 \underline{**}\, \tau_2 \underline{**} \ldots \underline{**}\, \tau_i \underline{**} \ldots \text{ and} \\
 & & \quad \forall i \geq 1.(\mathrm{val}_{D,\tau_i,\beta}(\mathsf{finite}) \text{ and } \mathrm{val}_{D,\tau_i,\beta}(\Theta)) \\
\mathrm{val}_{D,\tau,\beta}(\lceil\varphi\rceil) & \text{iff} & \tau = \langle\sigma\rangle \text{ and } \mathrm{val}_{D,\sigma,\beta}(\varphi) \\
\mathrm{val}_{D,\tau,\beta}(\mathsf{finite}) & \text{iff} & \tau \text{ is finite} \\
\mathrm{val}_{D,\tau,\beta}(\mathsf{infinite}) & \text{iff} & \tau \text{ is infinite}
\end{array}
$$

Fig. 3. Semantics: trace formulas

$$
\begin{array}{rcl}
\mathrm{val}_{D,\tau,\beta}([\![s]\!]\Psi) & \text{iff} & \begin{cases} \mathrm{val}_{D,\tau,\beta}(\Psi) & \tau \text{ is infinite} \\ \mathrm{val}_{D,\tau \underline{**} \tau',\beta}(\Psi) & \tau \text{ is finite and } \tau' = \mathrm{val}_{D,\mathrm{last}(\tau),\beta}(s) \end{cases} \\
\mathrm{val}_{D,\tau,\beta}(\{u\}\Psi) & \text{iff} & \begin{cases} \mathrm{val}_{D,\tau,\beta}(\Psi) & \tau \text{ is infinite} \\ \mathrm{val}_{D,\tau\cdot\mathrm{val}_{D,\sigma,\beta}(u)(\sigma),\beta}(\Psi) & \tau \text{ is finite and } \sigma = \mathrm{last}(\tau) \end{cases}
\end{array}
$$

Fig. 4. Semantics: trace modality Formulas

We define the semantics of updates as functions from states to (finite) traces in Fig. 2. The meaning of a single update $\ell := t$ is a function which, given a state σ', returns a singleton trace after updating σ' with the value of t at ℓ. The meaning of a multiple update is a function which, given a state σ', returns a trace, of the same length as the update, containing all the intermediate states of successively applying the update on σ'.

The evaluation function $\mathrm{val}_{D,\rho,\beta}$ for first-order formulas ϕ is straightforward. When ρ is a trace, then ϕ is evaluated in its final state $\sigma = \mathrm{last}(\rho)$. The function $\mathrm{val}_{D,\sigma,\beta}$ is defined exactly as $\mathrm{val}_{D,\beta}$ in standard first-order logic, with the obvious addition of the base case $\mathrm{val}_{D,\sigma,\beta}(\ell) = \sigma(\ell)$ when $\ell \in \mathsf{PV}$. A formula $[\![s]\!]\phi$ is true in σ if either $\mathrm{val}_{D,\sigma,\beta}(s)$ is infinite or else if ϕ is true in the last state of $\mathrm{val}_{D,\sigma,\beta}(s)$. A formula $\{u\}\varphi$ is true in σ if φ is true in the last state of $\mathrm{val}_{D,\sigma,\beta}(u)(\sigma)$. Formally, $\mathrm{val}_{D,\sigma,\beta}(\{u\}\varphi)$ iff $\mathrm{val}_{D,\sigma',\beta}(\varphi)$, where $\sigma' = \mathrm{val}_{D,\sigma,\beta}(u)(\sigma)$.

Fig. 3 gives the semantics of trace formulas. An update $\ell := e$ is true in a doubleton trace $\langle\sigma\rangle \curvearrowright \sigma'$, where σ' is obtained by updating σ at ℓ with the value of e in σ. The chop $\Theta_1 ** \Theta_2$ is true in τ, if either τ is infinite and Θ_1 is true in τ, or else τ can be split into τ_1 and τ_2 such that $\tau = \tau_1 \underline{**}\, \tau_2$ and Θ_1 (resp., Θ_2) is true in τ_1 (resp., τ_2). The finite iteration $\Theta^{<\omega}$ is true in τ, if τ can be split into n segments, each of which satisfies Θ.[4] The infinite iteration Θ^{ω} is true in τ, if τ can be split into infinitely many segments each of which is finite and satisfies Θ. The singleton formula $\lceil\varphi\rceil$ embeds a state formula φ into a trace

[4] We can additionally ask the first $n-1$ segments to be finite, without reducing the expressivity.

formula. It is true of a singleton trace $\langle\sigma\rangle$ containing a state σ which satisfies φ. We note that $\lceil\varphi\rceil^{\omega}$ is equivalent to $\lceil\varphi\rceil$, because $\langle\sigma\rangle \underline{**} \langle\sigma\rangle$ collapses to $\langle\sigma\rangle$. The trace formula finite ($\mathsf{infinite}$) is true in τ if τ is finite (infinite).

Fig. 4 gives the semantics of trace modality formulas. For space reasons, we only present dynamic logic operators (first-order operators are defined as usual). The formula $\mathrm{val}_{D,\tau,\beta}([\![s]\!]\Psi)$ is true, whenever Ψ is true in τ and τ is infinite. If τ is finite then s is executed from its last state. The resulting trace is sequentially composed with τ, and in that trace Ψ must be true. The semantics of formula $\{u\}\Psi$ is similar, only that the extension of τ is based on evaluating u.

Various equivalences and implications hold for state and trace formulas, e.g., the chop operator is associative, i.e. for any τ, $\mathrm{val}_{D,\tau,\beta}(\Theta_0 ** (\Theta_1 ** \Theta_2))$ iff $\mathrm{val}_{D,\tau,\beta}((\Theta_0 ** \Theta_1) ** \Theta_2)$. Also, a finite iteration is equivalent to either zero iterations or a finite iteration followed by a single iteration, i.e. for any τ, $\mathrm{val}_{D,\tau,\beta}(\Theta^{<\omega})$ iff $\mathrm{val}_{D,\tau,\beta}(\lceil\mathsf{true}\rceil \vee (\Theta^{<\omega} ** \Theta))$. Such properties give rise to simplification rules, some of which are further discussed in Sect. 4.3 below.

4 Calculus

We present a Gentzen-style sequent calculus for reasoning about trace formulas. Four kinds of rule sets can be distinguished: (i) program rules, which are responsible to remove programs from trace formulas; (ii) update simplification rules, which are applied to formulas preceded by an update and compute its weakest precondition; (iii) rules to reason about validity of trace formulas, and (iv) standard first-order rules. We focus on the first three categories as the last one is standard. Even though there is a considerable number of rules, the calculus is quite modular and amenable to automation. This is because the rules in categories (i)–(iii) are meant to be applied consecutively, i.e, first those in (i), then (ii), etc. In addition, many rules are deterministic (e.g., there is exactly one program rule for all but one programming construct) and they preserve validity (see Sect. 4.5). It is not necessary to backtrack in a proof.

4.1 Notation

The syntax for sequents is as usual:

$$\Gamma ::= \epsilon \mid \Psi, \Gamma \qquad \mathit{seq} ::= \Gamma \Rightarrow \Gamma$$

where Γ is a set of trace modality formulas with ϵ denoting the empty set. The left side of a sequent is called *antecedent* and the right side *succedent.* A sequent rule schema is written as

$$\mathsf{name}\ \frac{\overbrace{\Gamma_1 \Rightarrow \Delta_1 \ldots \Gamma_n \Rightarrow \Delta_n}^{\mathit{premises}}}{\underbrace{\Gamma \Rightarrow \Delta}_{\mathit{conclusion}}}$$

where name denotes the rule's name and $\Gamma, \Gamma_i, \Delta, \Delta_i, i \in \{1 \ldots n\}$ are schema variables matching sets of trace modality formulas. The meaning of a sequent $\Gamma \Rightarrow \Delta$ with $\Gamma = \{\Phi_1, \ldots, \Phi_n\}, \Delta = \{\Psi_1, \ldots \Psi_n\}$ is defined as

$$\mathrm{val}_{D,\tau,\beta}(\Gamma \Rightarrow \Delta) = \mathrm{val}_{D,\tau,\beta}((\Phi_1 \ \wedge \ldots \wedge \ \Phi_n) \rightarrow (\Psi_1 \ \vee \ldots \vee \ \Psi_m))$$

A sequent is *valid* if $\mathrm{val}_{D,\tau,\beta}(\Gamma \Rightarrow \Delta)$ is true for any domain D, trace τ, and variable assignment β.

A proof is a tree whose nodes are labeled with sequents. The leaves are called *proof goals*. A sequent rule is applicable to a proof if its conclusion matches the sequent of a proof goal. Applying a sequent rule on a goal g adds n (number of premises) new nodes as children to g, such that there is exactly one child for each premise labeled with the premise's instantiated sequent. Rules with no premises are called *axioms*. A goal is *closed* if any of the last rule applications on its proof branch where it appears has been an axiom application. A proof is closed when all its goals are closed. In addition to sequent rules there are rewrite rules, written as follows:

$$\mathsf{name} \qquad \xi \rightsquigarrow \xi' \quad (\overset{\rightsquigarrow}{=} \text{ is used instead of } \rightsquigarrow \text{ in case of directed equations})$$

where ξ, ξ' are trace modality formulas, trace formulas, state formulas or terms. A rewrite rule can be applied on any (sub-)formula or (sub-)term that matches the rule's left-hand side. Applying a rewrite rule replaces the matched subexpression by the accordingly instantiated right-hand side.

4.2 Program Rules

The program rules eliminate programs from formulas. The calculus follows the symbolic execution paradigm, i.e., programs are symbolically executed from left to right. To achieve this the rules always work on the first statement. For completeness we need at least one rule for each statement category. One rule per statement suffices, as our language has only side-effect free expressions, which do not need to be decomposed. In case of the while loops we provide two rules, one that unwinds one loop iteration and an invariant rule that comes with built-in coinduction. We give here only the rule versions for the main formula in the succedent, but analogous rules for the antecedent exist.

$$\mathsf{assign}\ \frac{\Gamma \Rightarrow \{u, \ell := e\}[\![r]\!]\Psi, \Delta}{\Gamma \Rightarrow \{u\}[\![\ell = e;\ r]\!]\Psi, \Delta}$$

Rule assign is applicable on any top level modality (possibly below an update) whose first statement is an assignment. The assignment is turned into an update and sequentially composed with a preceding update (if one exists).

$$\mathsf{ifThen}\ \frac{\Gamma, \{u\}(\mathsf{any} ** \lceil e \rceil) \Rightarrow \{u\}[\![s;\ r]\!]\Psi, \Delta \qquad \Gamma, \{u\}(\mathsf{any} ** \lceil \neg e \rceil) \Rightarrow \{u\}[\![r]\!]\Psi, \Delta}{\Gamma \Rightarrow \{u\}[\![\mathtt{if}\ e\ \mathtt{then}\ s\ \mathtt{fi};r]\!]\Psi, \Delta}$$

The rule for the conditional splits the proof into two branches; one for the case where its guard e evaluates to true (and the conditional's then-block is executed), the other for the case where e evaluates to false. Trace modality formulas are evaluated realtive to a trace. This is reflected in the expressions $\mathsf{any} ** \lceil e \rceil$ and $\mathsf{any} ** \lceil \neg e \rceil$, which ensure that the guard is evaluated in the last state of the trace. In case of an infinite trace the guards in the antecedents of both premises are trivially true and $\{u\}[\![r]\!]\Psi \equiv \{u\}[\![r;s]\!]\Psi \equiv \Psi$, hence, the two branches coincide.

After a program has been fully symbolically executed, the modality is eliminated by applying the rule

$$\mathsf{emptyModality}\ \frac{\Gamma \Rightarrow \{u\}\Psi, \Delta}{\Gamma \Rightarrow \{u\}[\![\,]\!]\Psi, \Delta}$$

The most complex program rules implement symbolic execution of loops. Rule

$$\mathsf{unwind}\ \frac{\Gamma \Rightarrow \{u\}[\![\mathtt{if}\ e\ \mathtt{then}\ (\mathtt{s}; \mathtt{while}\ e\ \mathtt{do}\ s\ \mathtt{od})\ \mathtt{fi};\ r]\!]\Psi, \Delta}{\Gamma \Rightarrow \{u\}[\![\mathtt{while}\ e\ \mathtt{do}\ s\ \mathtt{od};\ r]\!]\Psi, \Delta}$$

unwinds the loop, encoded as a program transformation capturing the operational small-step semantics. Whenever a loop is not bound by a fixed number of iterations, this rule is obviously incomplete. Instead of introducing a separate coinduction rule, we present a loop invariant rule with built-in coinduction:

$$\mathsf{whileInv}\ \frac{\begin{array}{l}\Gamma \Rightarrow \{u\}(\mathit{UpTo} ** \lceil \mathit{SInv} \rceil), \Delta \\ \mathit{Round}^{<\omega} ** \mathit{TGuard} \Rightarrow [\![s]\!]\big((\mathsf{finite} \rightarrow (\mathit{Round}^{<\omega} ** \lceil \mathit{SInv} \rceil)) \wedge (\mathsf{infinite} \rightarrow \mathit{RDiv})\big) \\ \mathit{Round}^{<\omega} ** \mathit{TGuard} ** \lceil \mathit{Div} \rceil \Rightarrow [\![s]\!](\mathsf{infinite} \vee \mathsf{finite} ** \lceil e \rceil ** \lceil \mathit{Div} \rceil) \\ \mathit{Round}^{<\omega} ** \mathit{TGuard} ** \lceil \neg \mathit{Div} \wedge \mathit{decr}{=}{=}\mathit{old} \rceil \Rightarrow \\ \qquad\qquad\qquad [\![s]\!](\mathsf{finite} ** \lceil \mathit{decr} < \mathit{old} \wedge \neg \mathit{Div} \rceil) \\ \mathit{UpTo} ** \lceil \neg \mathit{Div} \rceil ** \mathit{Round}^{<\omega} ** \lceil \mathit{SInv} \wedge \neg e \rceil \Rightarrow [\![r]\!]\Psi \\ \mathit{UpTo} ** \lceil \mathit{Div} \rceil ** \mathit{Round}^{\omega} \vee \mathit{UpTo} ** \lceil \mathit{Div} \rceil ** \mathit{RDiv} \Rightarrow \Psi \end{array}}{\Gamma \Rightarrow \{u\}[\![\mathtt{while}\ e\ \mathtt{do}\ s\ \mathtt{od};\ r]\!]\Psi, \Delta}$$

To instantiate the rule, a number of formulas and expressions must be specified: trace formula UpTo characterizes the traces to be considered up to executing the loop for the first time. State formula SInv denotes the state loop invariant, corresponding to invariants known from Hoare-style program logics. State formula Div characterizes the entry states of the loop under which it does not terminate, while RDiv is a trace formula describing a non-terminating loop body. The variant term decr is a natural number term and is strictly decreased in each loop iteration. Trace formula $\mathit{TGuard} := \lceil \mathit{SInv} \wedge e \rceil$ expresses that the state loop invariant and loop condition evaluate to true: they describe the states just before a loop iteration. Trace formula $\mathit{Round} := \mathit{TGuard} ** \mathit{TInv}$ describes all traces in which the loop guard holds initially and the whole trace (including the initial state) satisfy the "shape" described by the trace invariant formula TInv.

The loop invariant rule combines reasoning about terminating and non-terminating loops in one single rule. In particular it allows to reason about loops that diverge for some but not all initial states. To achieve this, the rule splits the proof into six branches: (i) The first branch shows that the *state loop invariant* formula *SInv* holds initially. (ii) The second branch ensures that both (state and trace) loop invariant formulas are preserved by the loop body provided that it terminates. If the loop body s is executed finitely under a trace which describes a finite number of loop unwindings ($Round^{<\omega}$) and ending in a state satisfying the state loop invariant and the loop guard then the resulting trace again satisfies the trace invariant $Round^{<\omega}$ and the state loop invariant. If the execution of the loop body diverges, the resulting trace is specified by *RDiv*. (iii) The third branch states that the diverging formula is correct by requiring that any loop iteration executed in a diverging state ends in a state which satisfies the loop condition. (iv) The fourth branch ensures that the variant term is decreased in each state where the loop terminates. (v) The fifth branch requires to prove that in case of a terminating loop the postcondition holds after executing the remaining program, while (vi) the sixth branch requires to show that in case of non-termination the postcondition holds under the produced infinite trace.

Example 3. We apply whileInv to the loop in Listing 1.1. The conclusion is

$$\mathsf{finite} \Rightarrow \{\ell := 0\}[\![\texttt{while}\ (\ell \geq 0)\ \texttt{do}\ \ell = \ell + 1\ \texttt{od}]\!]\Psi_1 \tag{2}$$

where $\Psi_1 \equiv \forall n.\mathrm{Attain}(\ell, n)$. If the loop body contains no loop itself, it is safe to set $RDiv \equiv \lceil \mathsf{false} \rceil$. The remaining schema variables are instantiated as follows:

$$\begin{array}{lll lll lll} UpTo & \equiv & \mathsf{finite} ** \lceil \ell{=}{=}0 \rceil & SInv & \equiv & \ell \geq 0 & TGuard & \equiv & \lceil \ell \geq 0 \rceil \\ Round & \equiv & \lceil \ell \geq 0 \rceil ** \ell := \ell + 1 & TInv & \equiv & \ell := \ell + 1 & Div & \equiv & \mathsf{true} \end{array}$$

The instantiation of *UpTo* is obvious and we expect that it can be automatically computed from the antecedent and the current update in most cases. The state invariant is as it would be in Hoare logic. From it we compute *TGuard*. The decreasing term is arbitrary, because the loop does not terminate in any state, hence $Div \equiv$ true. The new aspect is the trace invariant. We chose a precise shape given by the state transition that corresponds exactly to the assignment, but a more abstract trace invariant would also be possible. Finally, *Round* is obtained from *TGuard* and *TInv*.

4.3 Simplification Rules for Trace Formulas

In Fig. 5 we show a selection of simplification rules. The elimInf rules eliminate tailing chops which "follow" an already infinite trace. By repeated application of these rules we ensure that trace formulas describing an infinite trace end either with an infinite repetition operator ω or with the formula infinite. The other elimination rules like elimSingleStateRepInf are used to simplify trace formulas. For finite traces they ensure together with the unwindFinRep rules that the last chop ends with one of the trace formula finite, $\lceil \varphi \rceil$, u.

In summary, repeated application of the simplification rules establishes a normal form that permits syntactic recognition of finite or infinite trace formulas. This normal form is exploited in the rules for update simplification below.

$$\mathsf{elimInf}_1 \quad \Theta_{hd} ** (\Theta_1 ** \ell := e ** \Theta_2)^\omega ** \Theta_{\mathsf{tail}} \rightsquigarrow \Theta_{hd} ** (\Theta_1 ** \ell := e ** \Theta_2)^\omega$$

$\Theta_{hd}, \Theta_1, \Theta_2$ are optional

$$\mathsf{elimInf}_2 \quad \Theta_{hd} ** \mathsf{infinite} ** \Theta_2 \rightsquigarrow \Theta_{hd} ** \mathsf{infinite}$$

Θ_{hd} is optional

$$\mathsf{elimInf}_3 \ \frac{\Gamma \Rightarrow \{u\}(\Theta_{hd} ** \mathsf{finite} ** \Theta_{\mathsf{tail}}), \Delta \qquad \Gamma \Rightarrow \{u\}(\Theta_{hd} ** \mathsf{infinite}), \Delta}{\Gamma \Rightarrow \{u\}(\Theta_{hd} ** \mathsf{any} ** \Theta_{\mathsf{tail}}), \Delta}$$

Θ_{hd}, u are optional (similar for antecedent)

$$\mathsf{elimFalseRight}\ \frac{\Gamma \Rightarrow \{u\}(\Theta_{hd} \wedge \mathsf{infinite}), \Delta}{\Gamma \Rightarrow \{u\}(\Theta_{hd} ** \lceil \mathsf{false} \rceil ** \Theta_{tl}), \Delta} \qquad \mathsf{elimTrueRight}\ \frac{\Gamma \Rightarrow \{u\}(\Theta_{hd} ** \Theta_{tl}), \Delta}{\Gamma \Rightarrow \{u\}(\Theta_{hd} ** \lceil \mathsf{true} \rceil ** \Theta_{tl}), \Delta}$$

$\Theta_{hd}, \Theta_{tl}, u$ are optional (similar rules for the antecedent)

$$\mathsf{elimSingleStateRepInf} \quad \lceil \varphi \rceil^\omega \overset{\leftrightarrows}{=} \lceil \varphi \rceil \qquad \mathsf{unwindInfRep} \quad \Theta^\omega \overset{\leftrightarrows}{=} \Theta \ ** \ \Theta^\omega$$

$$\mathsf{unwindFinRepLeft}\ \frac{\Gamma, \{u\}\lceil \mathsf{true} \rceil \Rightarrow \Delta \quad \Gamma, \{u\}(\Theta^{<\omega} \ ** \ \Theta) \Rightarrow \Delta}{\Gamma, \{u\}\Theta^{<\omega} \Rightarrow \Delta} \qquad \mathsf{unwindFinRepRight}\ \frac{\Gamma \Rightarrow \{u\}(\Theta^{<\omega} \ ** \ \Theta), \ \{u\}\lceil \mathsf{true} \rceil, \Delta}{\Gamma \Rightarrow \{u\}\Theta^{<\omega}, \Delta}$$

u is optional

Fig. 5. Selection of simplification rules for trace formulas

$$\mathsf{propagateEqLeft}\ \frac{\Gamma, \Theta_{hd} ** \lceil \ell\texttt{==}t \rceil ** \ell := e[l/t] ** \Theta_{tl} \Rightarrow \Delta}{\Gamma, \Theta_{hd} ** \lceil \ell\texttt{==}t \rceil ** \ell := e ** \Theta_{tl} \Rightarrow \Delta}$$

Θ_{hd}, Θ_{tl} are optional; ℓ does not occur in t (similar for right side)

$$\mathsf{propagateUpdateEffectLeft}\ \frac{\Gamma, \Theta_{hd} ** \ell := e ** \lceil \ell\texttt{==}e \rceil ** \Theta_{tl} \Rightarrow \Delta}{\Gamma, \Theta_{hd} ** \ell := e ** \Theta_{tl} \Rightarrow \Delta} \qquad \mathsf{captureVariableValueLeft}\ \frac{\Gamma, \exists k.(\Theta_{hd} ** \lceil \ell\texttt{==}k \rceil ** \ell := e ** \Theta_{tl}) \Rightarrow \Delta}{\Gamma, \Theta_{hd} ** \ell := e ** \Theta_{tl} \Rightarrow \Delta}$$

Θ_{hd}, Θ_{tl} are optional; ℓ does not occur in e (similar for right side)

Fig. 6. Rules relating state information across traces

Besides simplification rules, further rules for reasoning about trace formulas exist. Some are shown in Fig. 6. They are used to propagate information inside a trace formula. For instance, captureVariableValueLeft introduces a rigid logical variable k to remember the value that a program variable ℓ has before an update.

4.4 Update Application Rules

Exhaustive application of program and trace formula rules results in a recognizably infinite or finite trace modality formula. In the latter case, u or $\lceil \varphi \rceil$ is the final subexpression. Hence, it is sufficient to provide update rules of trace modality formulas for those cases. Fig. 7 shows a selection of update application rules for trace modality formulas and state formulas. For a full set of update application rules for state formulas, see [14].

Perhaps surprisingly, $\{u\}\lceil \varphi \rceil$ is evaluated to false. The reason is that the application of u results in at least a doubleton trace, but $\lceil \varphi \rceil$ can only be true

$$\{u\}(\Phi \land \Psi) \overset{\rightsquigarrow}{=} \{u\}\Phi \land \{u\}\Psi \qquad \{u\}\neg\Phi \overset{\rightsquigarrow}{=} \neg\{u\}\Phi \qquad \{u\}\lceil\phi\rceil \overset{\rightsquigarrow}{=} \mathsf{False}$$

$$\{u\}P(t_1, \ldots, t_n) \overset{\rightsquigarrow}{=} P(\{u\}t_1, \ldots, \{u\}t_n)$$

$$\{u\}\exists\, x.\Phi \overset{\rightsquigarrow}{=} \exists\, x.\{u\}\Phi \;\; (x \text{ does not occur } \; u) \qquad \{u, \ell := e\}\Phi \overset{\rightsquigarrow}{=} \{u\}\{\ell := e\}\Phi$$

$$\{\ell := e\}(\Theta \ast\ast \lceil\varphi\rceil) \overset{\rightsquigarrow}{=} \text{if } (\mathsf{finite}) \text{ then } ((\{\ell := e\}\Theta) \land (\mathsf{finite} \ast\ast \lceil\{\ell := e\}\varphi\rceil)) \text{ else } (\Theta)$$

$$\{\ell := e\}(\Theta \ast\ast \ell := t) \overset{\rightsquigarrow}{=} \text{if } (\mathsf{finite}) \text{ then } (\Theta \land (\mathsf{finite} \ast\ast \lceil(\{\ell := e\}\ell) = t\rceil)) \text{ else } (\Theta)$$

(a) Update application on trace modality formulas

$$\{u, \ell := e\}\ell' \overset{\rightsquigarrow}{=} \begin{cases} e & \text{, if } \ell = \ell' \\ \{u\}\ell' & \text{, otherwise} \end{cases}$$

(b) Update application on state formulas

Fig. 7. Selection of update application rules

in a singleton trace. Another interesting observation is the difference between trace modality formulas $\{\ell := e\}(\Theta \ast\ast \lceil\varphi\rceil)$ and $\{\ell := e\}(\Theta \ast\ast \ell' := t)$ in case of finite traces. Application of a single update extends a given trace τ by exactly one state resulting in τ'. Because of the semantics of chop and the fact that $\lceil\varphi\rceil$ does not actually extend a trace, the first formula is evaluated to true if Θ is true in τ' and in its last state φ holds. On the other hand, the formula $(\Theta \ast\ast \ell' := t)$ describes a trace strictly longer than Θ. This means update application extends the trace beyond the reach of Θ which, therefore, needs to hold only in τ.

4.5 Soundness and Discussion of Completeness

Theorem 1 (Soundness). *All rules preserve validity, i.e., if all premises are valid, so is the conclusion.*

A standard argument yields

Corollary 1. *The DLTC calculus is sound: only valid sequents are provable.*

Another issue is completeness. As we reason about total correctness of Turing-complete programs, this is at best a relative notion. In interactive theorem proving pragmatic completeness (can we prove interesting properties of realistic programs?) is typically more relevant. Nevertheless, it would be interesting to know whether it is possible to prove completeness in the style of [10, 11], relative to certain oracles. One reason for completeness was in particular that the authors could escape to the meta-level for reasoning about theories like general properties (simplifications) of trace formulas. The crucial question will be whether the rules for propagating state information within trace formulas (Fig. 6) are sufficiently complete in the sense that we can reason about the concrete value of a program variable at a given position in a trace formula. In the end this boils down to the completeness of our update simplification rules. As described in the previous section, these rules are obviously complete for finite end pieces. For finite traces the relative completeness should follow directly from the relative completeness of the JavaDL calculus [3] as in this case our loop invariant rule is practically identical with their loop invariant rule.

Another issue is whether an explicit coinduction rule is needed in general or if coinduction can always be reduced to induction, as it is the case in Sect. 5.

5 An Example

We demonstrate how to use the calculus by proving formula (1) (Example 2, pg. 4). Due to space limitations we just point out the most interesting aspects.

After applying a propositional rule and assign we obtain a subgoal that is the conclusion in Ex. 3 which also describes the chosen rule instantiation. Since the program diverges, the fourth and fifth branch can easily be closed. The most interesting case is the sixth branch whose proof obligation after simplification is

$$\mathsf{finite} ** \lceil \ell{==}0 \rceil ** (\lceil \ell \geq 0 \rceil ** \ell := \ell + 1)^{\omega} \Rightarrow \forall n.(\mathsf{finite} ** \lceil \ell == n \rceil ** \mathsf{infinite}) \quad (3)$$

To prove the universally quantified formula in the succedent one needs induction. Our calculus contains a standard induction schema for natural numbers. We use the following induction formula where n is the induction variable:

$$\mathsf{finite}{**}\lceil \ell{==}0 \rceil{**}(\lceil \ell \geq 0 \rceil{**}\ell := \ell{+}1)^{\omega} \to \mathsf{finite}{**}\lceil \ell{==}n \rceil{**}(\lceil \ell \geq 0 \rceil{**}\ell := \ell{+}1)^{\omega}$$

Proving the base and step case pose no problem. We obtain

$$\begin{aligned} &\mathsf{finite} ** \lceil \ell{==}0 \rceil ** (\lceil \ell \geq 0 \rceil ** \ell := \ell + 1)^{\omega} \to \\ &\quad \forall n.(\mathsf{finite} ** \lceil \ell{==}n \rceil ** (\lceil \ell \geq 0 \rceil ** \ell := \ell + 1)^{\omega}) \end{aligned}$$

This is sufficient to prove (3) by weakening.

The example nicely demonstrates the strength of our logic: we are able to reason about a reachability property which requires arbitrary long traces. This is possible because our logic soundly incorporates coinductive reasoning of infinite traces. (Notice that a trace being infinite amounts to a trace being longer than any arbitrary natural numbers.)

6 Related and Future Work

Most related work was already discussed in the introduction. To characterize internal by observable behavior is the general concern of *abstract semantics* of programming languages; see for example [15] in the case of Java. Here, we are not necessarily interested in characterizing the program behavior fully; rather, we aim to verify a specific property, whereas abstract semantics strives to establish a meta theorem on semantic equivalence for a given programming language.

Interactive proof assistants such as Coq, Isabelle and Agda support coinduction and corecursion in the setting of a full-fledged higher-order logic.

In [16] an automatic method is presented to prove non-termination of programs based on solving constraints over unreachable parts of the state space.

The next obvious step in future work is to implement DLTC and its calculus, for example, on the basis of KeY [3, 12]. As mentioned in Sect. 4.5 we will investigate and prove relative completeness of our calculus.

7 Conclusion

We presented DLTC, a program logic that allows us to reason about programs and explicit, possibly infinite, traces. We also gave a sound sequent calculus for DLTC that is ready for implementation in a semi-automated theorem prover. One innovation of the calculus is an invariant rule for while loops that permits to prove properties of terminating and non-terminating loops at the same time. For non-termination we cover the case where the guard never becomes false as well as the case where the loop body may not terminate. Other innovative features of the calculus include propagation of symbolic states over traces and the capability to reduce coinductive to inductive statements.

Acknowledgement The work has been supported by the ERDF funded ICT R&D national programme project "Coinduction" and the Estonian Science Foundation grant no. 9398 and by the EU project FP7-610582 *Envisage: Engineering Virtualized Services* (`http://www.envisage-project.eu`).

References

1. Emerson, E.A., Halpern, J.Y.: "Sometimes" and "Not Never" revisited: On branching versus linear time temporal logic. Journal of the ACM **33** (1986) 151 – 178
2. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. Foundations of Computing. MIT Press (October 2000)
3. Beckert, B., Hähnle, R., Schmitt, P., eds.: Verification of Object-Oriented Software: The KeY Approach. Volume 4334 of LNCS. Springer (2006)
4. Beckert, B., Mostowski, W.: A program logic for handling Java Card's transaction mechanism. In Pezzé, M., ed.: Proceedings, Fundamental Approaches to Software Engineering (FASE), Poland. Volume 2621 of LNCS., Springer (2003) 246–260
5. Hähnle, R., Mostowski, W.: Verification of safety properties in the presence of transactions. In Barthe, G., Burdy, L., Huisman, M., Lanet, J.L., Muntean, T., eds.: Post Conf. Proc. of CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices. Volume 3362 of LNCS., Springer (2005) 151–171
6. Schellhorn, G., Tofan, B., Ernst, G., Reif, W.: Interleaved programs and rely-guarantee reasoning with ITL. In Combi, C., Leucker, M., Wolter, F., eds.: 18th Intl. Symp. on Temporal Representation and Reasoning, IEEE (2011) 99–106
7. Beckert, B., Schlager, S.: A sequent calculus for first-order dynamic logic with trace modalities. In Goré, R., Leitsch, A., Nipkow, T., eds.: Automated Reasoning, First Intl. Joint Conf., IJCAR. Volume 2083 of LNCS., Springer (2001) 626–641
8. Platzer, A.: A Temporal Dynamic Logic for Verifying Hybrid System Invariants. In Artëmov, S.N., Nerode, A., eds.: Logical Foundations of Computer Science, Intl. Symp., LFCS, New York, USA. Volume 4514 of LNCS., Springer (2007) 457–471

9. Beckert, B., Bruns, D.: Dynamic logic with trace semantics. In Bonacina, M.P., ed.: Automated Deduction, 24th International Conference on Automated Deduction, Lake Placid, USA. Volume 7898 of LNCS., Springer (2013) 315–329
10. Nakata, K., Uustalu, T.: A Hoare logic for the coinductive trace-based big-step semantics of While. In Gordon, A.D., ed.: Programming Languages and Systems, 19th European Symp. on Programming, ESOP, Paphos, Cyprus. Volume 6012 of LNCS., Springer (2010) 488–506
11. Nakata, K., Uustalu, T.: A Hoare logic for the coinductive trace-based big-step semantics of While. Logical Methods in Computer Science **11**(1) (2015) 1–32
12. Chang Din, C., Bubel, R., Hähnle, R.: KeY-ABS: A deductive verification tool for the concurrent modelling language ABS. In Felty, A., Middeldorp, A., eds.: Proc. 25th Intl. Conf. on Automated Deduction (CADE). LNCS, Springer (2015)
13. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for abstract behavioral specification. In Aichernig, B.K., de Boer, F., Bonsangue, M.M., eds.: Proc. 9th International Symposium on Formal Methods for Components and Objects. Volume 6957 of LNCS., Springer (2011) 142–164
14. Rümmer, P.: Sequential, parallel, and quantified updates of first-order structures. In Hermann, M., Voronkov, A., eds.: Proc. Logic for Programming, Artificial Intelligence and Reasoning, Cambodia. Volume 4246 of LNCS., Springer (2006) 422–436
15. Jeffrey, A., Rathke, J.: Java Jr: Fully Abstract Trace Semantics for a Core Java Language. In Sagiv, S., ed.: Programming Languages and Systems, 14th Europ. Symp. on Programming, ESOP. Volume 3444 of LNCS., Springer (2005) 423–438
16. Velroyen, H., Rümmer, P.: Non-termination checking for imperative programs. In Beckert, B., Hähnle, R., eds.: Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy. Volume 4966 of LNCS., Springer (2008) 154–170