Proving Temporal Properties by Abstract Interpretation

Samuel Marco Ueltschi

September 6, 2017

Contents

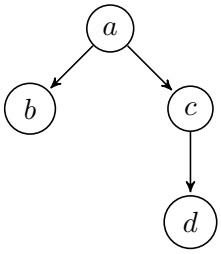


Figure 1: A basic state transition system

1 Introduction

Motivation etc.
This section introduces the necessary background for understanding the main concepts described in this thesis. Note that we assume that the reader already has a basic understanding of abstract interpretation. An detailed introduction into the theory of abstract interpretation can be found in [TODO].

2 State Transition Systems

To be able to analyse the behaviour of a program, it is necessary to express said behavior through a mathematical model. We model the operational semantics of programs using transition systems. This is based on the definitions presented in [1].

Definition 2.1 *Transition System A transition system is a tuple* $\langle \Sigma, \tau \rangle$ *where* Σ *is the set of all states in the system and* $\tau \in \Sigma \times \Sigma$ *is the so called transition relations that defines how one can transition from one state to the other.*

Transition systems allow us to model the semantics of a program independently of the programming language in which it was written. By expressing the possible transition between states in terms of a relation, it is also possible to capture nondeterminism. Figure 1 shows a simple transition system represented as directed graphs. States are represented as nodes and state transitions as directed edges.

We introduce the following auxiliary functions over states of a transition systems which will become useful in section (TODO ref XY) where we defined the semantics of CTL operators in terms of transition systems.

Definition 2.2 *Given a transition system* $\langle \Sigma, \tau \rangle$*.* $pre\colon \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ *maps a set of states* $X \in \mathcal{P}(\Sigma)$ *to the set of their predecessors with respect to the*

program transition relation τ:

$$pre(X) \stackrel{\text{def}}{=} \{s \in \Sigma \mid \exists s' \in X : \langle s, s' \rangle \in \tau\} \tag{1}$$

Definition 2.3 *Given a transition system $\langle \Sigma, \tau \rangle$. $\widetilde{pre}$: $\mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ maps a set of states $X \in \mathcal{P}(\Sigma)$ to the set of their predecessors with respect to the program transition relation τ with the limitation that only those predecessor states are selected which exclusively transition to states in X:*

$$\widetilde{pre}(X) \stackrel{\text{def}}{=} \{s \in \Sigma \mid \forall s' \in X : \langle s, s' \rangle \in \tau \Rightarrow s' \in X\} \tag{2}$$

To get an intuition for the difference between $\widetilde{\text{pre}}$ and pre, consider the state transition system depicted in figure 1. There it holds that $\text{pre}(\{b, d\}) = \{a, c\}$ because a is the predecessor of b and c the predecessor of d. However note that $\widetilde{\text{pre}}(\{b, d\}) = \{c\}$ since only c has transitions that exclusively end up in either b or d. Consequently it holds that $\widetilde{\text{pre}}(\{b, c\}) = \{a\}$ because a transitions exclusively to either b or c.

3 Computation Tree Logic (CTL)

Computation Tree Logic (CTL) is a logic which allows us to state properties about possible execution traces of state transition systems. In the context of this thesis, CTL is used to express temporal properties about the runtime behaviour of programs. This section gives a brief introduction into the syntax and semantic of CTL. Further information about CTL can be found in [2].

3.1 Syntax

The syntax of a CTL formula is given by the following grammar definition.

$$\begin{aligned}
\Phi ::= \; & \\
& a \mid \\
& \neg\Phi \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \\
& \forall \bigcirc \Phi \mid \exists \bigcirc \Phi \mid \\
& \forall \Diamond \Phi \mid \exists \Diamond \Phi \mid \\
& \forall \Box \Phi \mid \exists \Box \Phi \mid \\
& \forall (\Phi \; U \; \Phi) \mid \exists (\Phi \; U \; \Phi)
\end{aligned}$$

The term a is a placeholder for arbitrary atomic propositions.

3.2 Semantic

We now define the satisfaction relation $\models$ between states $\sigma \in \Sigma$ and CTL formulae. The satisfaction relation for atomic propositions depends on the semantics of the underlying logic for atomic propositions.

$$
\begin{aligned}
\sigma \models \neg\Phi &\iff \text{not } \sigma \models \Phi \\
\sigma \models \Phi_1 \wedge \Phi_2 &\iff (\sigma \models \Phi_1) \text{ and } (\sigma \models \Phi_1) \\
\sigma \models \Phi_1 \vee \Phi_2 &\iff (\sigma \models \Phi_1) \text{ or } (\sigma \models \Phi_1) \\
\sigma \models \forall \bigcirc \Phi &\iff \forall \pi \in Paths(\sigma) \colon (\pi[1] \models \Phi) \\
\sigma \models \exists \bigcirc \Phi &\iff \exists \pi \in Paths(\sigma) \colon (\pi[1] \models \Phi) \\
\sigma \models \forall(\Phi_1 \; U \; \Phi_2) &\iff \forall \pi \in Paths(\sigma) \colon (\exists j \geq 0 \colon \pi[j] \models \Phi_2 \wedge (\forall 0 \leq k < j \colon \pi[k] \models \Phi_1)) \\
\sigma \models \exists(\Phi_1 \; U \; \Phi_2) &\iff \exists \pi \in Paths(\sigma) \colon (\exists j \geq 0 \colon \pi[j] \models \Phi_2 \wedge (\forall 0 \leq k < j \colon \pi[k] \models \Phi_1)) \\
\sigma \models \forall\Box\Phi &\iff \forall \pi \in Paths(\sigma) \colon (\forall j \geq 0 \colon \pi[j] \models \Phi) \\
\sigma \models \exists\Box\Phi &\iff \exists \pi \in Paths(\sigma) \colon (\forall j \geq 0 \colon \pi[j] \models \Phi)
\end{aligned}
$$

The states $\sigma \in \Sigma$ are part of a state transition system $\langle \Sigma, \tau \rangle$ and $Paths(\sigma_0)$ is the set of all paths $\pi = \sigma_0\sigma_1\sigma_2 \ldots$ starting from σ_0 with $\pi[j] = \sigma_j$. The CTL formulae $\forall\Diamond\Phi$ and $\exists\Diamond\Phi$ are not defined for $\models$ as they are equivalent to $\forall(true \; U \; \Phi)$ and $\exists(true \; U \; \Phi)$. Furthermore the following useful equivalence relations exists which can be used to relate existential to universal CTL formulae.

$$
\begin{aligned}
\exists \bigcirc \Phi &\equiv \neg\forall \bigcirc (\neg\Phi) \\
\exists\Diamond\Phi &\equiv \neg\forall\Box(\neg\Phi) \\
\exists\Box\Phi &\equiv \neg\forall\Diamond(\neg\Phi)
\end{aligned}
$$

3.3 Recurrence and Guarantee Properties

TODO

4 Ranking Functions

The traditional approach for proving termination is based on inferring *ranking functions* [3] [4]. A ranking function is a partial function from program states to a well-ordered set (e.g. the natural numbers). To prove termination, the values of the ranking functions must decrease during program execution. Therefore the value that a *ranking function* assigns to a state is an upper

bound on the number of steps until the program terminates. Cousot and Cousot prove the existence of a *most precise ranking function* then can be derived by abstract interpretation [5]. The theory of abstract interpretation makes it possible to express various aspects of the semantics of a program. In that context the *most precise ranking function* for termination is called the *termination semantics*.

Definition 4.1 *The termination semantics is a ranking function* $\tau^t \in \Sigma \to \mathbb{N}$. *A program starting from some state* $\sigma \in \Sigma$ *terminates if and only if* $\sigma \in dom(\tau^t)$.

By definition of the *termination semantics*, a program will terminate if its initial state is in the domain of the ranking function. In other words, if the termination semantics assigns a natural value to the initial state that is an upper bound on the number of steps until termination.

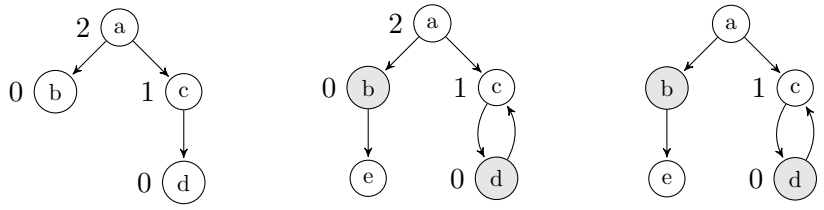
Based on the work of Cousot and Cousot [5]. Urban and Miné [1] extended the *termination semantics* to the more general notion of guarantee properties. A guarantee property states that some state satisfying a given property is guaranteed to be reached eventually. Termination is therefore just a guarantee property stating that some final state will be reached eventually. As for termination, the *guarantee semantics* is a ranking function that assigns each state an upper bound on the number of steps until a state satisfying the given property is reached.

Definition 4.2 *The guarantee semantics is a ranking function* $\tau^g_{[S]} \in \Sigma \to \mathbb{N}$ *where* $S \subseteq \Sigma$ *is a set of states satisfying a desired property. A program starting from some state* $\sigma \in \Sigma$ *will reach a state* $s \in S$ *if and only if* $\sigma \in dom(\tau^g_{[S]})$.

In addition to guarantee properties, Urban and Miné [1] also introduced the *recurrence semantics*. A recurrence property guarantees that a program starting from some state $\sigma \in \Sigma$ will reach some state satisfying a given property infinitely often. The value assigned to a state by the *recurrence semantics* is an upper bound on the number of executions steps until a state satisfying the property is reached the next time.

Definition 4.3 *The recurrence semantics is a ranking function* $\tau^r_{[S]} \in \Sigma \to \mathbb{N}$ *where* $S \subseteq \Sigma$ *is a set of states satisfying a desired property. A program starting from some state* $\sigma \in \Sigma$ *will reach a state* $s \in S$ *infinitely often if and only if* $\sigma \in dom(\tau^r_{[S]})$.

Figure 2 shows an example for the semantics discussed in this section. We illustrate the ranking functions by labeling the states in the transition systems with the corresponding value assigned to them by the ranking functions. The first example (a) shows the *termination semantics* for a state



(a) τ^t (b) $\tau^g_{[\{b,d\}]}$ (c) $\tau^r_{[\{b,d\}]}$

Figure 2: Example *termination semantics* (a), *guarantee semantics* (b) and *recurrence semantics* (c)

transition system that always terminates. Therefore the initial state has the value 2 assigned to it stating that this program terminates in at most two steps.

The second example (b) shows the *guarantee semantics* for the guarantee property that states that a gray state will be reached eventually. This holds for example (b) therefore the initial state has the value 2 assigned to it. The program reaches a gray state in at most two steps.

The last example (c) shows the *recurrence semantics* for the recurrence property that states that a gray state will be reached infinitely often. As one can see from the transition system, this is not true when starting from the initial state. Therefore the *recurrence semantics* is undefined for the initial state. However the property would hold when starting from state c or d. Accordingly these two states have the values 1 and 0 assigned to them.

We refer to [6] for a detailed discussion of the various semantics presented in this session.

5 Concrete Semantics for CTL

In section XY (TODO ref) we introduced the concept of ranking functions and explained how they express the semantics of recurrence and guarantee properties. To be able to analyse CTL properties, we extend the notion of ranking functions to CTL. Urban et al. [1] define ranking functions for guarantee and recurrence properties. The CTL semantics presented here are an extension of this work. We will define the *CTL semantics* inductively for each CTL operator such that arbitrary combinations of CTL properties can be analyzed.

Definition 5.1 *The CTL semantics for a given CTL formula Φ is a ranking function $\tau_\Phi \in \Sigma \to \mathbb{N}$. It encodes the semantics of Φ for a given state transition system $\langle \Sigma, \tau \rangle$ such that $\sigma \models \Phi \iff \sigma \in dom(\tau_\Phi)$ holds.*

We start by defining the *CTL semantics* for atomic propositions and logic operators (see definition 5.2). These definitions follow directly from the satisfiability relation for CTL properties (see section 3).

Definition 5.2 *Equations for basic CTL operators*

$$\tau_a \overset{\text{def}}{=} \lambda\sigma. \begin{cases} 0 & \text{if } \sigma \models a \\ \text{undefined} & \text{otherwise} \end{cases} \tag{3}$$

$$\tau_{\neg\Phi} \overset{\text{def}}{=} \lambda\sigma. \begin{cases} 0 & \text{if } \sigma \notin dom(\tau_\Phi) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{4}$$

$$\tau_{\Phi_1 \wedge \Phi_2} \overset{\text{def}}{=} \lambda\sigma. \begin{cases} \sup\{\tau_{\Phi_1}(s), \tau_{\Phi_2}(\sigma)\} & \text{if } \sigma \in dom(\tau_{\Phi_1}) \cap dom(\tau_{\Phi_2}) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{5}$$

$$\tau_{\Phi_1 \vee \Phi_2} \overset{\text{def}}{=} \lambda\sigma. \begin{cases} \sup\{\tau_{\Phi_1}(\sigma), \tau_{\Phi_2}(\sigma)\} & \text{if } \sigma \in dom(\tau_{\Phi_1}) \cap dom(\tau_{\Phi_2}) \\ \tau_{\Phi_1}(\sigma) & \text{if } \sigma \in dom(\tau_{\Phi_1}) \setminus dom(\tau_{\Phi_2}) \\ \tau_{\Phi_2}(\sigma) & \text{if } \sigma \in dom(\tau_{\Phi_2}) \setminus dom(\tau_{\Phi_1}) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{6}$$

Definition 5.3 defines how ranking functions are computed for universal and existential 'until' properties. This definition is an generalization of the *guarantee semantics* presented in [1].

Recall that for the CTL property $\forall(\Phi_1 U \Phi_2)$ to hold for some state $\sigma \in \Sigma$, all paths starting from said state must be a chain of states satisfying Φ_1 ending in a state satisfying Phi_2. We compute $\tau_{\forall(\Phi_1 U \Phi_2)}$ through a least fixed point iteration starting from the totally undefined ranking function $\dot{\emptyset}$. The first iteration assigns the value 0 to all states that satisfy Φ_2. In subsequent iterations we consider all states that satisfy Φ_1 and from which one can only transition to states that already satisfy $\forall(\Phi_1 U \Phi_2)$. These states are then assigned the largest ranking value of all reachable states plus one. By performing iterations this way, we backtrack paths in the state transition systems that end in a state satisfying Φ_2 and which are preceded by an unbroken chain of states satisfying Φ_1. Every state on such a path is guaranteed to satisfy the CTL 'until' property. Furthermore, by starting from 0 at states that satisfy Φ_2 and increasing while backtracking, we are constructing a ranking function in such a way that the value assigned to each state is an upper bound on the number of steps until a state is reached that satisfies Φ_2.

The $\widetilde{pre}$ relation guarantees that during the backtracking, only those states are considered which exclusively transition to states satisfying $\forall(\Phi_1 U \Phi_2)$.

This condition can be relaxed for existential 'until' properties by using the *pre* relation instead. That way, states that have at least one reachable state satisfying $\exists(\Phi_1 U \Phi_2)$ are also considered during the backtracking (see definitions 2.2 and 2.3).

Figures 3 and 4 give an example on how the iterative computation for 'until' works for universal and existential properties. Note how figure 4 has one addition iteration because of the existential quantifier. The initial state is added to ranking function in the last iteration because there exists one edge that leads to a state satisfying the property. For the universal property, the iteration stops after three iterations because not all successor states of the initial state satisfy the property.

Definition 5.3 *Equations for CTL until operator*

$$\tau_{\forall(\Phi_1 U \Phi_2)} \stackrel{\text{def}}{=} lfp^{\sqsubseteq}_{\dot{\emptyset}}\ \phi_{\forall(\Phi_1 U \Phi_2)} \tag{7}$$

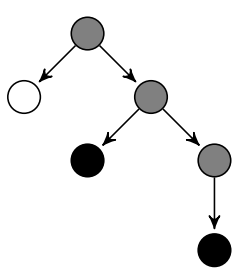
$$\phi_{\forall(\Phi_1 U \Phi_2)} f \stackrel{\text{def}}{=} \lambda\sigma. \begin{cases} 0 & \text{if } \sigma \in dom(\tau_{\Phi_2}) \\ \sup\{f(\sigma') + 1 \mid \langle \sigma, \sigma' \rangle \in \tau\} & \text{if } \sigma \notin dom(\tau_{\Phi_2}) \wedge \\ & \sigma \in dom(\tau_{\Phi_1}) \wedge \\ & \sigma \in \widetilde{pre}(dom(f)) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{8}$$

$$\tau_{\exists(\Phi_1 U \Phi_2)} \stackrel{\text{def}}{=} lfp^{\sqsubseteq}_{\dot{\emptyset}}\ \phi_{\exists(\Phi_1 U \Phi_2)} \tag{9}$$

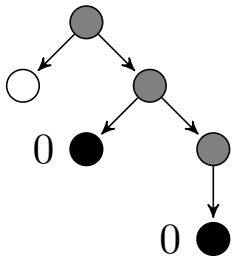
$$\phi_{\exists(\Phi_1 U \Phi_2)} f \stackrel{\text{def}}{=} \lambda\sigma. \begin{cases} 0 & \text{if } \sigma \in dom(\tau_{\Phi_2}) \\ \sup\{f(\sigma') + 1 \mid \langle \sigma, \sigma' \rangle \in \tau\} & \text{if } \sigma \notin dom(\tau_{\Phi_2}) \wedge \\ & \sigma \in dom(\tau_{\Phi_1}) \wedge \\ & \sigma \in pre(dom(f)) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{10}$$

$$\tag{11}$$

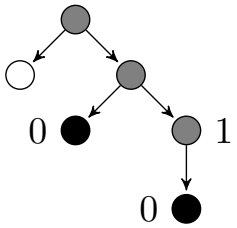
Definition 5.3 defines how ranking functions are computed for universal and existential 'global' properties ($\forall\Box\Phi$). This definition is generalization of the *recurrence semantics* presented in [1]. The CTL 'global' operator states that some property must hold globally for all paths starting from some state in the case of the universal quantifier ($\forall\Box\Phi$) or some path in case of the existential quantifier ($\exists\Box\Phi$). As for the 'until' operator, we distinguish between universal and existential properties by using either $\widetilde{pre}$ or *pre*. The ranking function for the 'global' operator is computed by greatest fixed point iteration. The iteration starts with the ranking function τ_Φ of the inner property.



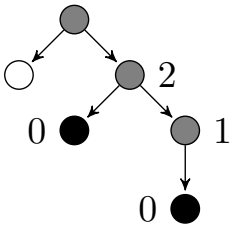
(a) Iteration 0



(b) Iteration 1

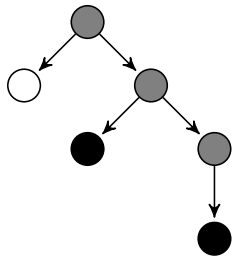


(c) Iteration 2

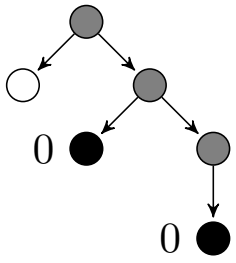


(d) Iteration 3

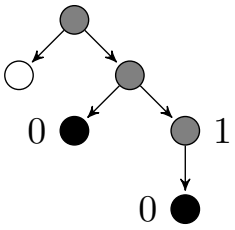
Figure 3: Iterative computation of $\tau_{\forall(gray\ U\ black)}$.



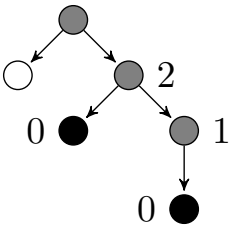
(a) Iteration 0



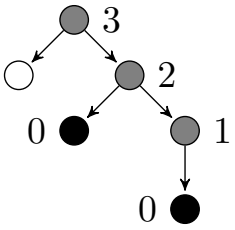
(b) Iteration 1



(c) Iteration 2

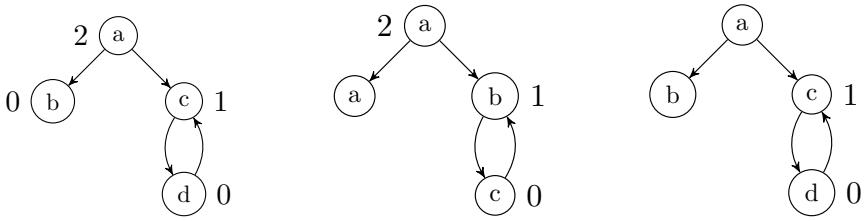


(d) Iteration 3



(e) Iteration 4

Figure 4: Iterative computation of $\tau_{\exists(gray\ U\ black)}$.



(a) Iteration 0 (b) Iteration 1 (c) Iteration 2

Figure 5: Iterative computation of $\tau_{\forall\Box\Phi}$.

For each iteration, every state that is still part of the domain of the ranking function is inspected. The inspected state is kept in the domain of the ranking function if all its successor states (or respectively some for the existential case) are also part of the domain of the ranking function, otherwise it is removed. The resulting final ranking function is a greatest fixed point that only contains states on paths where the property holds indefinitely.

Figures 5 and 6 give an example iteration for $\tau_{\exists\Box\Phi}$ and $\tau_{\forall\Box\Phi}$. Both iterations start with some initial ranking function τ_Φ. In the first iteration state 'b' is removed because it has no outgoing edges. For the existential case, the iteration stops here because all remaining states 'a', 'c' and ''d' have at least one edge to a node thats part of the ranking function. In the universal case we get an additional iteration that removes state 'a' because not all of its successor nodes (namely 'b') are part of the ranking function.
Note that the way $\tau_{\forall\Box\Phi}$ and $\tau_{\exists\Box\Phi}$ are defined, it is not possible to use the 'global' operator on finite paths. Only infinite paths are considered to hold globally. (TODO state why this choice was made)

Definition 5.4 *Equations for CTL global operator*

$$\tau_{\forall\Box\Phi} \stackrel{\text{def}}{=} gfp^{\sqsubseteq}_{\tau_\Phi}\ \phi_{\forall\Box\Phi} \tag{12}$$

$$\phi_{\forall\Box\Phi} f \stackrel{\text{def}}{=} \lambda\sigma. \begin{cases} f(x) & \text{if } \sigma \in \widetilde{pre}(dom(f)) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{13}$$

$$\tau_{\exists\Box\Phi} \stackrel{\text{def}}{=} gfp^{\sqsubseteq}_{\tau_\Phi}\ \phi_{\exists\Box\Phi} \tag{14}$$

$$\phi_{\exists\Box\Phi} f \stackrel{\text{def}}{=} \lambda\sigma. \begin{cases} f(x) & \text{if } \sigma \in pre(dom(f)) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{15}$$

Definition 5.5 defines the CTL 'next' operator. A state satisfies $\forall \bigcirc \Phi$ if all its successors satisfy the property, correspondingly $\exists \bigcirc \Phi$ is satisfied if at least one successor satisfies the property. This corresonds to the definition

(a) Iteration 0 (b) Iteration 1

Figure 6: Iterative computation of $\tau_{\exists\Box\Phi}$.

of the $\widetilde{pre}$ and pre relations. Zero is assigned to each state that satisfies the property.

Definition 5.5 *Equations for CTL next operator*

$$\tau_{\forall\bigcirc\Phi} \stackrel{\text{def}}{=} \lambda\sigma. \begin{cases} 0 & \text{if } \sigma \in \widetilde{pre}(dom(\tau_\Phi)) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{16}$$

$$\tau_{\exists\bigcirc\Phi} \stackrel{\text{def}}{=} \lambda\sigma. \begin{cases} 0 & \text{if } \sigma \in pre(dom(\tau_\Phi)) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{17}$$

6 Imperative Language

In this section we briefly introduce a minimal imperative programming language. It will be used in section 8 to define the abstract CTL semantics. The language has no procedures, pointers or recursion. It is non-deterministic. Variables are integer valued ($\mathbb{Z}$) and statically allocated.
First we define the syntax for arithmetic and boolean expressions. The syntax definitions are based on chapter 3 of [6].

Definition 6.1 *Syntax for arithmetic and boolean expressions. Arithmetic*

expressions are defined over a set of variables $\mathcal{X}$.

$$
\begin{array}{rcll}
aexp & ::= & X & X \in \mathcal{X} \\
 & | & [i_1, i_2] & i_1 \in \mathbb{Z} \cup \{-\infty\},\ i_1 \in \mathbb{Z} \cup \{\infty\},\ i_1 \leq i_2 \\
 & | & -\, aexp & \\
 & | & aexp \diamond aexp & \diamond \in \{+, -, *, /\}
\end{array}
$$

$$
\begin{array}{rcll}
bexp & ::= & ? & \textit{non-deterministic choice} \\
 & | & \neg bexp & \\
 & | & bexp \wedge bexp & \\
 & | & bexp \vee bexp & \\
 & | & aexp \diamond aexp & \diamond \in \{<, \leq, >, \geq\}
\end{array}
$$

The semantics for expressions are defined as expected. Please refer to [6] for a formal definition. Note that the symbol '?' stands for non-deterministic choice.

Programs are represented as control-flow-graphs (CFG) (see definition 6.2). A control-flow-graph $(V, E) \in cfg$ consists of a set of nodes V and edges E. Every control point of a program is assigned a label $l \in \mathcal{L}$. The nodes in the control-flow-graph correspond to those labels. An edge $(u, s, v) \in edge$ states that one can transition from node u to v by executing statement s. The `skip` statements transitions from one node to another without doing anything, the boolean expression *bexp* limits the set of states that are allowed to transition to the next node and the assignment $X := aexp$ assigns the value of the arithmetic expression *aexp* to the variable X. A program $(\text{cfg}, l_{entry}, l_{exit}) \in prog$ consists of a control-flow-graph and two special nodes that defined the entry and exit point of the program.

Definition 6.2 *Program representation as control-flow-graph. Let* $\mathcal{L}$ *be the set of program labels.*

$$
\begin{array}{rcll}
stmt & ::= & \texttt{skip} & \\
 & | & bexp & \\
 & | & X := aexp & X \in \mathcal{X}
\end{array}
$$

$$
\begin{array}{rcl}
edge & \stackrel{\text{def}}{=} & \mathcal{L} \times stmt \times \mathcal{L} \\
cfg & \stackrel{\text{def}}{=} & \mathcal{P}(\mathcal{L}) \times \mathcal{P}(edge) \\
prog & \stackrel{\text{def}}{=} & cfg \times \mathcal{L} \times \mathcal{L}
\end{array}
$$

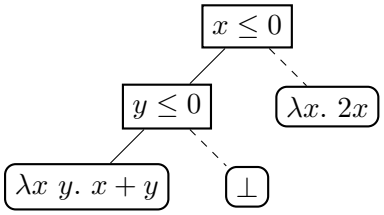


Figure 7: Example for decision tree

We also introduce the following auxiliary relations on nodes in the context of a control-flow-graph $(V, E) \in cfg$. The relation $in(l)$ denotes all direct predecessor nodes of l. The relation $out(l)$ denotes all direct successor nodes of l.

Definition 6.3 *Given a control-flow-graph* $(V, E) \in cfg$

$$in(l) \stackrel{\text{def}}{=} \{l' \in V \mid \exists\ (u, s, v) \in E \colon u = l' \wedge v = l\}$$
$$out(l) \stackrel{\text{def}}{=} \{l' \in V \mid \exists\ (u, s, v) \in E \colon u = l \wedge v = l'\}$$

7 Piecewise Defined Ranking Functions

This section briefly recaps the decision tree abstract domain. First we give a description of the domain. Then we introduce ordering relations between the elements of the domain and relevant operations on the elements of the domain. An in-depth description of the topics covered in this section can be found in [6].

7.1 Domain

The elements of abstract domain are *piecewise-defined* partial functions represented in terms of decision trees. The nodes of the trees are linear constrains and the leafs are linear functions. Decision trees partition a state space, given by a set of variable $\mathcal{X}$, into linear partitions. Each partition is defined through the conjunction of linear constraints on the path from root to leaf in the decision tree. The linear function at the leaf determines what value is associated to the partition.

Figure 7 gives an example for such a decision tree. It consists of two nodes with linear constraints $x \leq 0$ and $y \leq 0$. The left most leaf is the function $\lambda x\ y.\ x + y$. It is defined for all states satisfying $x \leq 0 \wedge y \leq 0$ according to the constraints from root to leaf. The right most leaf is the function $\lambda x.\ 2x$, it is defined for all states satisfying $\neg(x \leq 0)$ (taking a right turn negates the linear constraint of a node). The leaf in the middle is a bottom node, signifying that the function for the corresponding partition is undefined.

Combining all constraints and functions of the decision tree in figure 7 yields the following partial function:

$$f(x,y) = \begin{cases} x+y & \text{if } x \leq 0 \wedge y \leq 0 \\ 2x & \text{if } x > 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Now we formalize the domain using mathematical notations.

The constrains at the inner nodes of the decision tree are elements of the *linear constraints auxiliary abstract domain* $\mathcal{C}$.

$$\mathcal{C} \stackrel{\text{def}}{=} \left\{ c_1 X_1 + \cdots + c_n X_n + c_{n+1} \geq 0 \;\middle|\; \begin{array}{l} \mathcal{X} = \{X_1, \ldots, X_n\} \\ c_1, \ldots, c_n, c_{n+1} \in \mathbb{Z} \\ gcd(|c_1|, \ldots, |c_n|, |c_{n+1}|) = 1 \end{array} \right\}$$

The constraints of $\mathcal{C}$ can be instances of the *interval abstract domain*, the *octagon abstract domain* or the *polyhedra abstract domain* (TODO cite).

Leafs of the decision trees are elements of the *functions auxiliary abstract domain* $\mathcal{F}$. Elements of $\mathcal{F}$ are either natural valued functions or one of the two special elements $\top_{\mathcal{F}}$ or $\bot_{\mathcal{F}}$.

$$\mathcal{F} \stackrel{\text{def}}{=} \{\mathbb{Z}^{|\mathcal{X}|} \to \mathbb{N}\} \cup \{\top_{\mathcal{F}},\ \bot_{\mathcal{F}}\}$$

We now define the *computational order* $\sqsubseteq_{\mathcal{F}}$ and *approximation order* $\preceq_{\mathcal{F}}$ over the elements of $\mathcal{F}$. Intuitively, these partial orders approximate the *computational order* $\sqsubseteq$ and *approximation order* $\preceq$ presented in section XY (TODO ref).
In the *computational order* $\sqsubseteq_{\mathcal{F}}$ the special elements $\top_{\mathcal{F}}$ and $\bot_{\mathcal{F}}$ are comparable whereas in the *approximation order* $\preceq_{\mathcal{F}}$ they are incomparable as shown in the Hasse diagrams in figure 8. Regular function elements are compared point wise.

$$f_1 \sqsubseteq_{\mathcal{F}} f_2 \iff f_1 \preceq_{\mathcal{F}} f_2 \iff \forall x \in \mathbb{Z}^{|\mathcal{X}|} \colon f_1(x) \leq f_2(x)$$

We now define the decision tree abstract domain $\mathcal{T}$. An element $t \in \mathcal{T}$ is either a *leaf node* $LEAF\colon f$ consisting of a function $f \in \mathcal{F}$ (denoted $t.f$), or a *decision node* $NODE\{c\} : l; r$ consists of a linear constraint $c \in \mathcal{C}$ (denoted $t.c$) and a left and a right sub tree $l, r \in \mathcal{T}$ (denoted $t.l$ and $t.r$).

$$\mathcal{T} \stackrel{\text{def}}{=} \{LEAF\colon f \mid f \in \mathcal{F}\} \cup \{NODE\{c\} : l; r \mid c \in \mathcal{C}, l, r \in \mathcal{T}\}$$

For algorithmic purposes we also define $\mathcal{T}_{NIL}$. This adds an additional leaf element NIL to $\mathcal{T}$ to represent the absence of information about a partition.

$$\mathcal{T}_{NIL} \stackrel{\text{def}}{=} \{NIL\} \cup \{LEAF\colon f \mid f \in \mathcal{F}\} \cup \{NODE\{c\} : l; r \mid c \in \mathcal{C}, l, r \in \mathcal{T}_{NIL}\}$$

(a) *computational order* $\sqsubseteq_{\mathcal{F}}$ (b) *approximation order* $\preceq_{\mathcal{F}}$

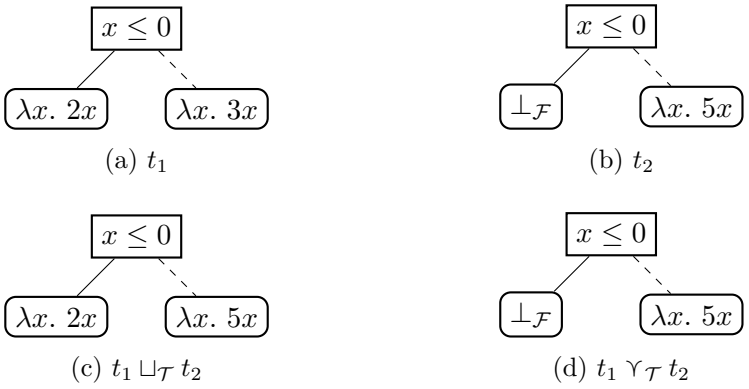
Figure 8: Hasse diagrams for $\sqsubseteq_{\mathcal{F}}$ and $\preceq_{\mathcal{F}}$

7.2 Unification

The function $tree\_unification\colon (\mathcal{T}_{NIL} \times \mathcal{T}_{NIL}) \to (\mathcal{T}_{NIL} \times \mathcal{T}_{NIL})$ takes two trees as input and reshapes them such that they have the same inner node structure i.e. the same linear constraints. The resulting trees are semantically equivalent to their original. This simplifies comparing the partitions of two trees to each other. The $tree\_unification$ algorithm is described in great detail in [6].

Algorithm 1 Tree Join Helper

```
▷ t_left, t_right ∈ T_NIL
▷ f_LEAF ∈ P(C) → F → F → T
▷ f_LeftNIL, f_RightNIL ∈ P(C) → F → T
function TREE_JOIN_HELPER(t_left, t_right, f_LEAF, f_LeftNIL, f_RightNIL)
    function AUX(t_l, t_r, C)
        if isNil(t_l) ∧ isNil(t_r) then
            return NIL
        else if isNil(t_l) ∧ isLeaf(t_r) then
            return f_LeftNIL(C, t_r.f)
        else if isLeaf(t_l) ∧ isNil(t_r) then
            return f_RightNIL(C, t_l.f)
        else if isLeaf(t_l) ∧ isLeaf(t_r) then
            return f_LEAF(C, t_l.f t_r.f)
        else                                        ▷ assert t_l.c = t_r.c
            l ← AUX(t_l.l, t_r.l, {t_l.c} ∪ C)
            r ← AUX(t_l.r, t_r.r, {¬t_l.c} ∪ C)
            return NODE{t_l.c}: l; r
        end if
    end function
    (t_l, t_r) ← TREE_UNIFICATION(t_left, t_right)
    return AUX(t_l, t_r)
end function
```



(a) t_1 (b) t_2 (c) $t_1 \sqcup_{\mathcal{T}} t_2$ (d) $t_1 \curlyvee_{\mathcal{T}} t_2$

Figure 9: Decision Tree Join Example

7.3 Join

Two trees can be joined to form the union of all partitions represented by the trees. There are two variations of the join operator. The *computational join* $\sqcup_{\mathcal{T}} \colon (\mathcal{T}_{NIL} \times \mathcal{T}_{NIL}) \to \mathcal{T}_{NIL}$ and the *approximation join* $\curlyvee_{\mathcal{T}} \colon (\mathcal{T}_{NIL} \times \mathcal{T}_{NIL}) \to \mathcal{T}_{NIL}$. The first one joins leaf nodes with $\sqsubseteq_{\mathcal{F}}$ the latter with $\preceq_{\mathcal{F}}$. Figure 9 demonstrates the difference between the two join types. When joining two trees where one defines a function for a partition and the other does not (see left leaf in t_1 and t_2), the *computational join* will preserve the defined function and the *approximation join* will set the partition to undefined. For detailed discussion about the two join types see [6].

7.4 Meet

7.5 Filter

TODO

7.6 Reset

TODO

7.7 Backward Assign

TODO

8 Abstract Semantics for CTL

In this section we present a sound, computable abstraction of the CTL semantics τ_Φ. Computing the ranking function τ_Φ is in general not computable as one can easily encode the halting problem in τ_Φ. Therefore, we settle for

a sound but computable approximation of τ_Φ by using the abstract domain of piecewise defined ranking functions presented in section 7.

Definition 8.1 *The abstract CTL semantics $\tau_\Phi^\sharp \in \mathcal{L} \to \mathcal{T}$ is a sound approximation of the CTL semantics τ_Φ with regards to the approximation order $\preceq$.*

Recall that the CTL semantics $\tau_\Phi \in \Sigma \to \mathbb{N}$ is a partial function that assigns numerical values to program states $\sigma \in \Sigma$. In the abstract version, program states are partitioned by decision trees $t \in \mathcal{T}$ and grouped by program labels $l \in \mathcal{L}$. A program satisfies a given CTL property Φ if the decision tree of the initial program label $\tau_\Phi^\sharp(t_{init})$ is defined over all partitions (TODO introduce notion of defined w.r.t decision trees).

The following sections present how to compute $\tau_\Phi^\sharp$ for each CTL operator. We start with the basic operators $\wedge, \vee, \neg$ and atomic propositions. These can be computed directly for each program label. Then we present how to compute the universal $\forall U$, $\forall\bigcirc$ and $\forall\Box$ operators through fixed-point iteration. Followed by a discussion on how to adapt the universal operators to their existential version. Note that the abstract CTL semantics are computed recursively. The recursion stops at atomic propositions.

8.1 Path Independent Operators

TODO

Atomic Propositions

Atomic propositions are path independent, therefore $\tau_a^\sharp$ assigns the same decision tree to each program label $l \in \mathcal{L}$. This decision tree assigns the constant value 0 to all partitions that satisfy the atomic proposition a. We compute this tree by using the RESET$[\![a]\!]$ operator on the totally undefined decision tree $\bot$ (see section XY TODO).

Definition 8.2 *Abstract CTL semantics for atomic propositions*

$$\tau_a^\sharp \stackrel{\text{def}}{=} \lambda l.\ RESET[\![a]\!]\bot \tag{18}$$

Fixing the RESET operator

The RESET$[\![a]\!]$ operator was originally introduced in [6] in the context of abstract guarantee semantics and would overapproximate the set of partitions that satisfy the atomic proposition a. During the work on this thesis, we discovered that this original definition is actually unsound and leads to incorrect analysis results.

The problem is best described using an example. Consider the abstract CTL semantics $\tau^\sharp_{x^2<y^3+1}$. The non linear constraint $x^2 < y^3 + 1$ can usually not be represented by any of the common numerical domains. An overapproximating implementation of RESET$[\![x^2 < y^3 + 1]\!]$ will therefore reset some pairs (x, y) for which $x^2 < y^3 + 1$ does not hold which is unsound as to the definition of the CTL semantics $\tau_{x^2<y^3+1}$. Note that this problem propagates to more complex temporal properties that depend on atomic propositions.

We resolve this problem by using an underapproximating implementation of RESET as presented in section XY (TODO ref).

Logical Operators
Now we define the abstract CTL semantics for the logical operators $\wedge$, $\vee$ and $\neg$.

Definition 8.3 *Abstract CTL semantics for logic operators*

$$\tau^\sharp_{\Phi_1 \wedge \Phi_2} \stackrel{\text{def}}{=} \lambda l.\ (\tau^\sharp_{\neg\Phi_1} l)\ \sqcup_{\mathcal{T}}\ (\tau^\sharp_{\neg\Phi_2} l) \tag{19}$$

$$\tau^\sharp_{\Phi_1 \vee \Phi_2} \stackrel{\text{def}}{=} \lambda l.\ (\tau^\sharp_{\neg\Phi_1} l)\ \sqcap_{\mathcal{T}}\ (\tau^\sharp_{\neg\Phi_2} l) \tag{20}$$

$$\tau^\sharp_{\neg\Phi} \stackrel{\text{def}}{=} \lambda l.\ \mathit{COMPLEMENT}\ (\tau^\sharp_{\neg\Phi} l) \tag{21}$$

The abstract CTL semantics for $\wedge$ and $\vee$ (equations 19 and 20) combine the decision trees of the nested properties piecewise for each program label.

The *computational join* $\sqcup_{\mathcal{T}}$ is used to combine the two trees (see section 7.3) in case of the logical $\vee$. This operator forms the union of the two decision trees. Note that we use the computational version of the join operator to include partitions that are defined in at least one of the two trees. If a partition is defined in both trees, the smallest upper bound of the assigned value is formed to approximate the concrete CTL semantics.

The corresponding definition for the logical $\wedge$ operator forms the intersection of the two trees by using the *computational meet* $\sqcap_{\mathcal{T}}$ (see section 7.4). As with the logical $\vee$, the smallest upper bound is formed if a partition is

defined in both trees. By using the computational version of the operator, we ensure that no NIL leafs are introduced when forming the intersection.

For the logical $\neg$ operator we introduce the COMPLEMENT operator. This operator replaces all defined leafs with a $\bot$-leaf and all $\bot$-leafs with the constant value 0. By doing so, all states that originally satisfied the property do not satisfy it any more and vice-versa. However one has to be careful when changing a partition from undefined to defined. Decision trees are an approximation of the concrete CTL semantics. Therefore not all states that are undefined in the abstract decision tree are actually undefined in the concrete ranking function. Partitions that are undefined because of this uncertainty are marked with a $\top$-leaf. To ensure soundness, these leafs have to be ignored when forming the complement of a decision tree. The COMPLEMENT operator is implemented in algorithm 2.

Algorithm 2 Tree Complement

```
function COMPLEMENT(t)                                   ▷ t ∈ T_NIL
    if (isNode(t) ∧ t.f = ⊤) ∨ isNil(t) then
        return t                                         ▷ ignore ⊤ and NIL
    else if isLeaf(t) ∧ t.f = ⊥ then
        return LEAF : 0                                  ▷ undefined becomes defined
    else if isLeaf(t) then
        return LEAF : ⊥                                  ▷ defined becomes undefined
    else
        l ← COMPLEMENT(t.l)
        r ← COMPLEMENT(t.r)
        return NODE{t.c} : l; r
    end if
end function
```

8.2 Path Dependent Operators

In this section we describe how the abstract CTL semantics for the path dependent operators ($\forall \bigcirc \Phi$, $\exists \bigcirc \Phi$, $\forall(\Phi_1 U \Phi_2)$, $\exists(\Phi_1 U \Phi_2)$, $\forall\Box\Phi$, $\exists\Box\Phi$) can be computed.

First, we define the two functions $[\![stmt]\!]_o \in \mathcal{T}_{NIL} \to \mathcal{T}_{NIL}$ and $[\![stmt]\!]_u \in \mathcal{T}_{NIL} \to \mathcal{T}_{NIL}$. The first one uses overapproximation on the underlying numerical domains and the second one underapproximation.

The purpose of these functions is to encode the effect of backward propagating an edge in the control-flow-graph i.e. the effect of executing a statement. Assume that we computed a decision tree for the target node of some

edge. This decision tree represents the value of the ranking function for the program label associated with this node. By applying $[\![\cdot]\!]$ to this tree, we compute the decision tree that holds before executing the statement i.e. the value of the ranking function associated to the source node of the edge.

Definition 8.4 *Abstract semantics for basic statements*

$$[\![\texttt{skip}]\!]_o \stackrel{\text{def}}{=} \lambda t.\ \mathit{STEP}(t)$$
$$[\![\mathit{bexp}]\!]_o \stackrel{\text{def}}{=} \lambda t.\ \mathit{B\text{-}ASSIGN}(t)$$
$$[\![X := \mathit{aexp}]\!]_o \stackrel{\text{def}}{=} \lambda t.\ \mathit{FILTER}(t)$$

$$[\![\texttt{skip}]\!]_u \stackrel{\text{def}}{=} \lambda t.\ \mathit{STEP}(t)$$
$$[\![\mathit{bexp}]\!]_u \stackrel{\text{def}}{=} \lambda t.\ \mathit{B\text{-}ASSIGN\text{-}UNDER}(t)$$
$$[\![X := \mathit{aexp}]\!]_u \stackrel{\text{def}}{=} \lambda t.\ \mathit{FILTER\text{-}UNDER}(t)$$

The `skip` statement is handled by the STEP operator. This operator increases the value of all defined partitions in the decision tree by one (see XY TODO). Recall that a defined partition in a decision tree represents a set of states that satisfies some CTL property. The associated value is an upper bound on the number of steps until some condition is reached. By executing `skip` this number is incremented by one. For assignments and boolean conditions we use the corresponding B-ASSIGN and FILTER operators.

What is left to do is define how to compute the abstract CTL semantics for the different path dependent CTL properties. All are computed through fixed-point iteration on the control flow graph. In general, the abstract CTL semantics $\tau^\sharp_\Phi$ are a fixed point of the abstract transformer $\phi^\sharp_\Phi$ w.r.t the computational order $\sqsubseteq_\mathcal{T}$.

$$\tau^\sharp_\Phi = lfp^{\sqsubseteq_\mathcal{T}} \phi^\sharp_\Phi$$
$$\phi^\sharp_\Phi \in (\mathcal{L} \to \mathcal{T}_{NIL}) \to (\mathcal{L} \to \mathcal{T}_{NIL})$$

The initial value for the fixed-point computation and the definition of $\phi^\sharp_\Phi$ depend on the semantic of the corresponding CTL property Φ. Note that $\phi^\sharp_\Phi$ must approximate the abstract transformer ϕ_Φ of the concrete CTL semantics.

Until
The abstract transformers for the universal and existential 'until' properties are given in definition XY

References

[1] Caterina Urban and Antoine Miné, "Proving Guarantee and Recurrence Temporal Properties by Abstract Interpretation," in *VMCAI*, 2015, pp. 190–208.

[2] Christel Baier, Joost-Pieter Katoen, and Kim Guldstrand Larsen, *Principles of model checking*, MIT press, 2008.

[3] Alan Touring, "Checking a large routing," *Report of a Conference on High Speed Automatic Calculating Machines*, pp. 67–69, 1949.

[4] Robert W. Floyd, "Assigning meanings to programs," *Proceedings of Symposium on Applied Mathematics*, pp. 19:19–32, 1967.

[5] P. Cousot and R. Cousot, "An abstract interpretation framework for termination," in *Conference Record of the 39$^{\text{th}}$ Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Philadelphia, PA, Jan. 25-27 2012, pp. 245–258, ACM Press, New York.

[6] Caterina Urban, *Static Analysis by Abstract Interpretation of Functional Temporal Properties of Programs.*, Ph.D. thesis, École Normale Supérieure, Paris, France, 2015.