Termination Analysis of Heap-Manipulating Programs by Abstract Interpretation

Master's Thesis Report

Lukas Neukom

Supervised by Dr. Caterina Urban
Prof. Dr. Peter Müller

ETH Zürich
September 28, 2016

Contents

1 Introduction

Computer software is an integral part of today's society. It is often the case that software does not work as expected, either by human mistake or because of invalid assumptions made during development. Such bugs may cause huge financial losses, as the bug in Knight's software that caused a $440 million loss in just 30 minutes[1], or even loss of life.

The Microsoft Zune Z2K bug[2], caused by non-termination, shows that one of the most important properties for software correctness is termination, which proven by Alan Turing in 1936, is undecidable in general [Tur36]. This means that there cannot exist a program that, given any other program, decides whether it terminates. Still, for some programs termination can be proven. The traditional method, proposed by Alan Turing in [Tur49], consists of inferring *ranking functions*. A ranking function is a function from program states to elements of a well-ordered set whose values decrease during program execution. Once such a ranking function is found for a program, we know that the program terminates, since given any state of the program the ranking function limits the number of steps possible during execution.

In [CC12] Patrick Cousot and Radhia Cousot prove the existence of a *most precise* ranking function that can be derived by abstract interpretation, a mathematical theory of sound approximation of the semantics of a computer program developed by Patrick Cousot and Radhia Cousot [CC77, Cou78]. Abstract interpretation allows defining various semantics of computer programs at different levels of detail and relating them. Soundness in the context of abstract interpretation means that a semantics always over-approximates the semantics it abstracts. The most precise ranking function defined in [CC12] is one such semantics, called the *termination semantics*. Still, the termination semantics is not computable in general.

FuncTion, a static analyzer developed by Caterina Urban at École Normale Supérieure [Urb15a], is able to infer piecewise-defined ranking functions for programs written in a C-like language by abstract interpretation of the termination semantics introduced by Cousot and Cousot. To do this, she introduces the decision trees abstract domain which is a sound and decidable abstraction of the termination semantics. The abstraction simplifies the problem by allowing the ranking function to be partially undefined. This means that in the worst case, the ranking function does not prove termination for any state of the program.

[1] `http://www.bloomberg.com/news/articles/2012-08-02/knight-shows-how-to-lose-440-million-in-30-minutes`

[2] `https://techcrunch.com/2008/12/31/zune-bug-explained-in-detail/`

FuncTion has only very limited support for heap manipulating programs. Since most programs use dynamic data structures which require a heap, FuncTion is not able to prove termination for a large set of programs. The goal of this project is the development of a new static analysis based on the termination semantics for proving termination of heap-manipulating programs and the implementation of this analysis in FuncTion. To motivate our work let us consider such a program:

Example 1.0.1 – List Traversal. Let us consider the following program:

$$\begin{array}{l}\texttt{while } {}^{1}l \neq \mathit{null} \texttt{ do}\\ \quad {}^{2}l := l.\mathit{next}\\ \texttt{od}^{3}\end{array}$$

which implements a traversal of a linked list. The label 1 represents the initial control point of the program, the label 2 represent the control point before the assignment $l := l.\mathit{next}$ and the label 3 represents the final control point of the program. The loop follows the *next* references until *null* is reached. Even though this program only consists of two lines, proving its termination is already quite sophisticated: using an implementation of a linked list which allows cycles, as is common in C, means that the program only terminates if the list pointed to by l is acyclic. A ranking function, partitioned over the program control points of the program, for Example 1.0.1 is:

$$\begin{aligned} f(1) &\stackrel{\text{def}}{=} \lambda s. \begin{cases} 2|l|+1 & l \text{ is acyclic in } s \\ \text{undefined} & \mathit{otherwise} \end{cases} \\ f(2) &\stackrel{\text{def}}{=} \lambda s. \begin{cases} 2|l| & l \text{ is acyclic in } s \\ \text{undefined} & \mathit{otherwise} \end{cases} \\ f(3) &\stackrel{\text{def}}{=} \lambda s.\, 0 \end{aligned}$$

where s is a program state and $|l|$ is the length of the list pointed to by l. At the final program control point 3 the program is terminated and thus no instructions are needed. At the program control point 2 the program needs $2|l|$ more steps to termination for acyclic lists and is undefined otherwise and at the initial control point 1 the program needs $2|l| + 1$ steps to termination for acyclic lists and is undefined otherwise. For example, a list of length 1 needs 3 steps to termination: the initial test at control point 1, the assignment at control point 2 and the final test at control point 1.

Another, much harder program, is an implementation of a counting-out game, also known as the Josephus Problem[3]:

Example 1.0.2 – Josephus Problem. Let us consider the following program:

```
while [1] l ≠ l.next do
    [2] l.next := l.next.next
    [3] l := l.next
od[4]
```

which implements a counting-out game. People (represented as list entries) are standing in a circle, waiting to be executed. Counting begins at a specified point (where l initially points to) and in a specific direction (following the *next* field). In each iteration one person is skipped and the following person is executed. This is repeated until only one person remains (when $l = l.next$). It is assumed that the input of the Josephus Problem is a cyclic list, in which case the algorithm terminates since the cycle gets smaller with each iteration. But the algorithm also terminates for any list that ends in a cycle.

Example 1.0.1 and Example 1.0.2 show that finding ranking functions for program manipulating heaps often involves reasoning about the shape of the heap, which can be quite tricky. Programs, such as Example 1.0.2, that manipulate the heap during execution make this even harder. Inferring ranking functions as shown in Example 1.0.1 gives the developer not only the very important termination guarantee for the right input, but also the valuable information that calling this program might not terminate for other inputs.

The main contribution of this thesis is the development of a heap abstraction which is then used to extend FuncTion's decision tree abstract domain to be able to prove termination of heap-manipulating programs. We focus on programs manipulating lists, where the termination argument depends on the size of the list such as Example 1.0.1 and 1.0.2. The resulting abstraction shows promising results for programs that have no heap manipulating statements, such as Example 1.0.1, or very little heap manipulating statements, such as Example 1.0.2, where we are able to infer ranking functions representing all terminating states.

The remainder of this thesis is structured as follows: in Chapter 2 we extend the syntax and semantics of FuncTion to allow heap manipulating statements. The novel heap abstraction is introduced in Chapter 3 and the extension of FuncTion's termination abstraction using the heap abstraction is presented in Chapter 4. In Chapter 5 we briefly discuss the implementation in FuncTion and then evaluate the implemented analysis.

[3] https://en.wikipedia.org/wiki/Josephus_problem

The final Chapter, Chapter 6, shows possible future work. We assume that the reader is familiar with abstract interpretation, otherwise we refer to [Urb15b, Chapter 2] for an overview or [CC92] for an exhaustive presentation.

2 Language & Semantics

We formalize our results with respect to a small imperative language. Specifically we extend the language defined in [Urb15b, Chapter 3] to support heap manipulating constructs such as allocation, field dereference and field updates. The variables are statically typed and either of type $\mathbb{Z}$, the set of mathematical integers, or of type reference. We do not explicitly type the variables but assume that a program is correctly typed.

2.1 Syntax

We first extend the syntax of the language with additional constructs for heap manipulation. The new syntax is inductively defined in Figure 2.1 with the new rules highlighted in gray. We distinguish between variables of type $\mathbb{Z}$ denoted by *Num* and variables of type reference denoted by *Ref*. Together they form the set of variables $Var = Num \cup Ref$ where $Num \cap Ref = \emptyset$. To simplify the presentation we have no notion of classes but instead define a set of fields *Field*, that can be used in field dereference expressions. We assume that each instance in the heap has all the fields in *Field*. In the discussion we focus on the newly introduced rules of the syntax. For a more thorough discussion of the other rules see the original definition in [Urb15b, Chapter 3].

The main addition is the introduction of reference expressions *rexp* which involve variables of type reference $x \in Ref$, the special reference *null* which represents a null reference and the new expression *new* which returns the reference of a new instance. We have also introduced the field dereference expression $x.f$ where we distinguish between field dereference expressions that return an arithmetic value, defined in *aexp*, and field dereference expressions that return a reference, defined in *rexp*.

Boolean expressions *bexp* can now also be built by comparing reference expressions using reference equality.

Statements now also support field assignments for arithmetic expressions $x.f := aexp$, assignments for reference expressions $x := rexp$ and field assignments for reference expressions $x.f := rexp$.

aexp	::=	x	$x \in \mathit{Num}$
	|	$x.f$	$x \in \mathit{Ref},\ f \in \mathit{Field}$
	|	$[i_1, i_2]$	$i_1 \in \mathbb{Z} \cup \{-\infty\},\ i_2 \in \mathbb{Z} \cup \{+\infty\},\ i_1 \leq i_2$
	|	$-$ *aexp*	
	|	*aexpop aexp*	$op \in \{+, -, *, /\}$
rexp	::=	x	$x \in \mathit{Ref}$
	|	$x.f$	$x \in \mathit{Ref},\ f \in \mathit{Field}$
	|	*null*	
	|	*new*	
bexp	::=	`?`	
	|	`not` *bexp*	
	|	*bexp* `and` *bexp*	
	|	*bexp* `or` *bexp*	
	|	*aexp op aexpr*	$op\ \in \{<, \leq, =, \neq\}$
	|	*rexp op rexp*	$op\ \in \{=, \neq\}$
stmt	::=	`skip`	
	|	x `:=` *aexp*	$x \in \mathit{Var}$
	|	$x.f$ `:=` *aexp*	$x \in \mathit{Var},\ f \in \mathit{Field}$
	|	x `:=` *rexp*	$x \in \mathit{Var}$
	|	$x.f$ `:=` *rexp*	$x \in \mathit{Var},\ f \in \mathit{Field}$
	|	`if` *bexp* `then` *stmt* `else` *stmt* `fi`	
	|	`while` *bexp* `do` *stmt* `od`	
	|	*stmt stmt*	
prog	::=	*stmt*	

Figure 2.1: The syntax of our programming language based on the programming language defined in [Urb15b, Chapter 3]. The new rules are highlighted in gray.

In the following the *initial* control point $i[\![stmt]\!]$ of an instruction *stmt* defines where the execution of the instruction starts and the *final* control point $f[\![stmt]\!]$ defines where the execution of the instruction *stmt* ends.

In the following sections we define various semantics for our small imperative language. These allow reasoning at various levels of detail about properties of programs written in our language.

2.2 Expression Semantics

We now define the semantics of expressions which is later used to define the invariance semantics (cf. Section 2.3) and the termination semantics (cf. Section 2.4). The original expression semantics was only defined for variables of type $\mathbb{Z}$ and therefore did not have a notion of heap. We first introduce a notion of *store* to formalize the heap and then define the semantics of arithmetic expressions *aexp*, reference expressions *rexp* and boolean expressions *bexp*. In the discussion we focus on the newly introduced rules for arithmetic and boolean expressions (highlighted in gray in Figure 2.2 and Figure 2.4) and the semantics for reference expressions.

Let *Loc* be the set of all possible locations in the heap. We use the special location *null* to denote an undefined reference. We define $Val = \mathbb{Z} \cup Loc$ as the set of values a variable can have.

Definition 2.2.1. An *environment* $e : Var \to Val$ is a function that maps each program variable $x \in Var$ to its value $e(x) \in Val$. We denote the set of all environments by *Env*.

A *store* $s : Loc \times Field \rightharpoonup Val$ maps each combination of location $l \in Loc$ and field $f \in Field$ to its value $s(l, f) \in Val$ or is undefined in case the field is not yet initialized. We denote the set of all stores by *Store*. We assume that, for all fields $f \in Field$ and all stores $s \in Store$, $s(null, f)$ is undefined. Intuitively this means that the location *null* has no fields.

A *state* $(e, s) \in Env \times Store$ is a pair of an environment and a store. We denote the set of all states by *State*.

The semantics of an arithmetic expression *aexp* is a function $[\![aexp]\!] : State \to \mathcal{P}(\mathbb{Z})$ which maps a state to the set of all possible values of *aexp*. The dereference expression $x.f$ returns the value of the field f of the reference x. Note that the result of the expression might be empty if x is *null* or the result of the expression is not an integer.

$$
\begin{array}{lcl}
[\![\, x \,]\!](e,s) & \stackrel{\text{def}}{=} & \{\, e(x) \mid x \in \mathit{Num} \,\} \\
[\![\, x.f \,]\!](e,s) & \stackrel{\text{def}}{=} & \{\, s(e(x),f) \mid e(x) \neq \mathit{null},\ s(e(x),f) \in \mathbb{Z} \,\} \\
[\![\, [a,b] \,]\!](e,s) & \stackrel{\text{def}}{=} & \{\, x \mid a \leq x \leq b \,\} \\
[\![\, -\, \mathit{aexp} \,]\!](e,s) & \stackrel{\text{def}}{=} & \{\, -x \mid x \in [\![\, \mathit{aexp} \,]\!](e,s) \,\} \\
[\![\, \mathit{aexp}_1 \,+\, \mathit{aexp}_2 \,]\!](e,s) & \stackrel{\text{def}}{=} & \{\, x + y \mid x \in [\![\, \mathit{aexp}_1 \,]\!](e,s),\ y \in [\![\, \mathit{aexp}_2 \,]\!](e,s) \,\} \\
[\![\, \mathit{aexp}_1 \,-\, \mathit{aexp}_2 \,]\!](e,s) & \stackrel{\text{def}}{=} & \{\, x - y \mid x \in [\![\, \mathit{aexp}_1 \,]\!](e,s),\ y \in [\![\, \mathit{aexp}_2 \,]\!](e,s) \,\} \\
[\![\, \mathit{aexp}_1 \,*\, \mathit{aexp}_2 \,]\!](e,s) & \stackrel{\text{def}}{=} & \{\, x * y \mid x \in [\![\, \mathit{aexp}_1 \,]\!](e,s),\ y \in [\![\, \mathit{aexp}_2 \,]\!](e,s) \,\} \\
[\![\, \mathit{aexp}_1 \,/\, \mathit{aexp}_2 \,]\!](e,s) & \stackrel{\text{def}}{=} & \{\, t(x \,/\, y) \mid x \in [\![\, \mathit{aexp}_1 \,]\!](e,s),\ y \in [\![\, \mathit{aexp}_2 \,]\!](e,s),\ y \neq 0 \,\}
\end{array}
$$

where $t\colon \mathbb{R} \to \mathbb{Z}$ rounds its argument towards an integer:

$$
t(x) \stackrel{\text{def}}{=} \begin{cases} \max\{y \in \mathbb{Z} \mid y \leq x\} & x \geq 0 \\ \min\{y \in \mathbb{Z} \mid y \geq x\} & x < 0 \end{cases}
$$

Figure 2.2: Semantics of arithmetic expressions *aexp*. The new rules are highlighted in gray.

The semantics of a reference expression *rexpr* is a function $[\![\, \mathit{rexpr} \,]\!]\colon \mathit{State} \to \mathcal{P}(\mathit{Loc})$ which maps a state to the set of all possible locations of *rexpr*. Note that, analogously to the field dereference expression in *aexp*, the result of the field dereference expression of *rexp* might be empty in the case where x is *null* or the result is not a location.

$$
\begin{array}{lcl}
[\![\, x \,]\!](e,s) & \stackrel{\text{def}}{=} & \{\, e(x) \mid x \in \mathit{Ref}, \,\} \\
[\![\, x.f \,]\!](e,s) & \stackrel{\text{def}}{=} & \{\, s(e(x),f) \mid e(x) \neq \mathit{null},\ s(e(x),f) \in \mathit{Loc} \,\} \\
[\![\, \mathit{null} \,]\!](e,s) & \stackrel{\text{def}}{=} & \{\, \mathit{null} \,\} \\
[\![\, \mathit{new} \,]\!](e,s) & \stackrel{\text{def}}{=} & \{\, r \mid s(r) \text{ is undefined } \,\}
\end{array}
$$

Figure 2.3: Semantics of reference expressions *rexp*.

The semantics of a boolean expression *bexpr* is a function $[\![\, \mathit{bexpr} \,]\!]\colon \mathit{State} \to \mathcal{P}(\{\mathit{true}, \mathit{false}\})$ which maps a state to the set of all possible truth values of *bexpr*.

$$
\begin{array}{lcl}
[\![\texttt{?}]\!](e,s) & \stackrel{\text{def}}{=} & \{ \mathit{true}, \mathit{false} \} \\
[\![\texttt{not}\ \mathit{bexp}]\!](e,s) & \stackrel{\text{def}}{=} & \{ \neg x \mid x \in [\![\mathit{bexp}]\!](e,s) \} \\
[\![\mathit{bexp}_1\ \texttt{and}\ \mathit{bexp}_2]\!](e,s) & \stackrel{\text{def}}{=} & \{ x \wedge y \mid x \in [\![\mathit{bexp}_1]\!](e,s),\ y \in [\![\mathit{bexp}_2]\!](e,s) \} \\
[\![\mathit{bexp}_1\ \texttt{or}\ \mathit{bexp}_2]\!](e,s) & \stackrel{\text{def}}{=} & \{ x \vee y \mid x \in [\![\mathit{bexp}_1]\!](e,s),\ y \in [\![\mathit{bexp}_2]\!](e,s) \} \\
[\![\mathit{aexp}_1 < \mathit{aexp}_2]\!](e,s) & \stackrel{\text{def}}{=} & \{ x < y \mid x \in [\![\mathit{aexp}_1]\!](e,s),\ y \in [\![\mathit{aexp}_2]\!](e,s) \} \\
[\![\mathit{aexp}_1 \leq \mathit{aexp}_2]\!](e,s) & \stackrel{\text{def}}{=} & \{ x \leq y \mid x \in [\![\mathit{aexp}_1]\!](e,s),\ y \in [\![\mathit{aexp}_2]\!](e,s) \} \\
[\![\mathit{aexp}_1 = \mathit{aexp}_2]\!](e,s) & \stackrel{\text{def}}{=} & \{ x = y \mid x \in [\![\mathit{aexp}_1]\!](e,s),\ y \in [\![\mathit{aexp}_2]\!](e,s) \} \\
[\![\mathit{aexp}_1 \neq \mathit{aexp}_2]\!](e,s) & \stackrel{\text{def}}{=} & \{ x \neq y \mid x \in [\![\mathit{aexp}_1]\!](e,s),\ y \in [\![\mathit{aexp}_2]\!](e,s) \} \\
[\![\mathit{rexp}_1 = \mathit{rexp}_2]\!](e,s) & \stackrel{\text{def}}{=} & \{ x = y \mid x \in [\![\mathit{rexp}_1]\!](e,s),\ y \in [\![\mathit{rexp}_2]\!](e,s) \} \\
[\![\mathit{rexp}_1 \neq \mathit{rexp}_2]\!](e,s) & \stackrel{\text{def}}{=} & \{ x \neq y \mid x \in [\![\mathit{rexp}_1]\!](e,s),\ y \in [\![\mathit{rexp}_2]\!](e,s) \}
\end{array}
$$

Figure 2.4: Semantics of boolean expressions *bexp*. The new rules are highlighted in gray.

2.3 Invariance Semantics

We now extend the invariance semantics defined in [Urb15b, Chapter 3] to include our new statements for heap manipulation (cf. Section 2.1) and use states instead of just environments (cf. Section 2.2). The invariance semantics allows reasoning about the states, i.e., the environments and the stores, before and after the execution of a statement. We only briefly repeat the relevant definitions and refer to [Urb15b, Chapter 3] for a more thorough discussion.

$$
\begin{array}{l}
\tau_{\mathcal{I}}[\![\texttt{skip}]\!]S \stackrel{\text{def}}{=} S \\
\tau_{\mathcal{I}}[\![x := \mathit{aexp}]\!]S \stackrel{\text{def}}{=} \{(e[x \leftarrow v], s) \in \mathit{State} \mid (e,s) \in S,\ v \in [\![\mathit{aexp}]\!](e,s)\} \\
\tau_{\mathcal{I}}[\![x.f := \mathit{aexp}]\!]S \stackrel{\text{def}}{=} \{(e, s[(e(x), f) \leftarrow v) \in \mathit{State} \mid (e,s) \in S,\ v \in [\![\mathit{aexp}]\!](e,s)\} \\
\tau_{\mathcal{I}}[\![x := \mathit{rexp}]\!]S \stackrel{\text{def}}{=} \{(e[x \leftarrow v], s) \in \mathit{State} \mid (e,s) \in S,\ v \in [\![\mathit{rexp}]\!](e,s)\} \\
\tau_{\mathcal{I}}[\![x.f := \mathit{rexp}]\!]S \stackrel{\text{def}}{=} \{(e, s[(e(x), f) \leftarrow v) \in \mathit{State} \mid (e,s) \in S,\ v \in [\![\mathit{rexp}]\!](e,s)\} \\
\tau_{\mathcal{I}}[\![\texttt{if}\ \mathit{bexp}\ \texttt{then}\ \mathit{stmt}_1\ \texttt{else}\ \mathit{stmt}_2\ \texttt{fi}]\!]S \stackrel{\text{def}}{=} \\
\qquad \tau_{\mathcal{I}}[\![\mathit{stmt}_1]\!]\{s \in S \mid \mathit{true} \in [\![\mathit{bexp}]\!](s)\} \cup \\
\qquad \tau_{\mathcal{I}}[\![\mathit{stmt}_2]\!]\{s \in S \mid \mathit{false} \in [\![\mathit{bexp}]\!](s)\} \\
\tau_{\mathcal{I}}[\![\texttt{while}\ \mathit{bexp}\ \texttt{do}\ \mathit{stmt}\ \texttt{od}]\!]S \stackrel{\text{def}}{=} \{s \in \text{lfp}\ \phi_{\text{post}} \mid \mathit{false} \in [\![\mathit{bexp}]\!](s)\} \\
\qquad \phi_{\text{post}}(X) \stackrel{\text{def}}{=} S \cup \tau_{\mathcal{I}}[\![\mathit{stmt}]\!]\{s \in X \mid \mathit{true} \in [\![\mathit{bexp}]\!](s)\} \\
\tau_{\mathcal{I}}[\![\mathit{stmt}_1\ \mathit{stmt}_2]\!]S \stackrel{\text{def}}{=} \tau_{\mathcal{I}}[\![\mathit{stmt}_2]\!](\tau_{\mathcal{I}}[\![\mathit{stmt}_1]\!]S)
\end{array}
$$

Figure 2.5: Invariance semantics of instructions *stmt*. The new rules are highlighted in gray.

Figure 2.5 shows the extended invariance semantics $\tau_{\mathcal{I}}[\![\mathit{stmt}]\!] \colon \mathcal{P}(\mathit{State}) \to \mathcal{P}(\mathit{State})$

of instructions. It takes as input a set of states and outputs the set of possible states after the execution of the statement *stmt*. As before the new rules are highlighted in gray. The invariance semantics $\tau_{\mathcal{I}}[\![prog]\!] \in \mathcal{P}(State)$ of a program *prog* returns the set of possible program states at the final program control point $f[\![prog]\!]$. It is defined from the set of all program states *State* as:

Definition 2.3.1 (Invariance Semantics)**.** Given a program *prog*, its *invariance semantics* $\tau_{\mathcal{I}}[\![prog]\!] \in \mathcal{P}(State)$ is:

$$\tau_{\mathcal{I}}[\![prog]\!] = \tau_{\mathcal{I}}[\![stmt]\!] \stackrel{\text{def}}{=} \tau_{\mathcal{I}}[\![stmt]\!](State) \tag{2.1}$$

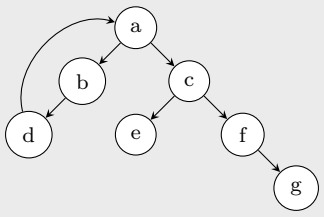
where $\tau_{\mathcal{I}}[\![stmt]\!] \in \mathcal{P}(State) \to \mathcal{P}(State)$ is the invariance semantics of each program instruction *stmt* defined in Figure 2.5.

2.4 Termination Semantics

We now extend the termination semantics defined in [Urb15b, Chapter 4] to include our new statements for heap manipulation (cf. Section 2.1) and use states instead of just environments (cf. Section 2.2). We focus on the new rules, highlighted in gray, for field and reference assignments in the discussion and refer to [Urb15b, Chapter 4] for a more thorough discussion of the other rules.

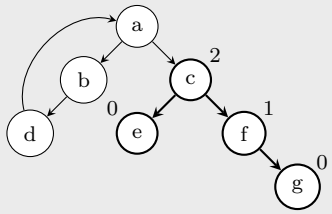
The termination semantics defines the most precise ranking function $\tau_{\mathrm{T}} : State \to \mathbb{O}$ for a program. It is defined starting from the final program states, where the value of the function is zero, and retracing the program *backwards* while mapping each program state in *State* definitely leading to a final state to an ordinal representing an upper bound on the number of program execution steps remaining to termination. This means that any state that might lead to a non-terminating state is not included in the termination semantics. The domain $\mathrm{dom}(\tau_{\mathrm{T}})$ of τ_{T} is the set of states from which the program execution terminates. To clarify, let us consider the following example:

Example 2.4.1. Let us consider the following transitions of a program:



The final states are the states (e) and (g). By going backwards from (e) we can reach (c) in one step, but since (c) can also be reached by going backwards from (g) in two steps the upper bound on the number of steps to termination for (c) is two. Going further

backwards is not possible since (a) might be a non-terminating state because of the left transition which leads to a loop. The resulting termination semantics is:



where thick states belong to the domain of the ranking function. Any state not belonging to the termination semantics might lead to a non-termination execution.

$$\tau_{\mathrm{T}}[\![\,\texttt{skip}\,]\!]r \stackrel{\text{def}}{=} \lambda s \in \mathrm{dom}(r).\ r(s) + 1$$

$$\tau_{\mathrm{T}}[\![\,x := aexp\,]\!]r \stackrel{\text{def}}{=} \lambda(e,s).\ \begin{cases} \sup\{r(e[x \leftarrow v], s) + 1 \mid v \in [\![aexp]\!](e,s)\} \\ \qquad\qquad \exists v \in [\![aexp]\!](e,s) \wedge \\ \qquad \forall v \in [\![aexp]\!](e,s) \colon (e[x \leftarrow v], s) \in \mathrm{dom}(r) \\ \text{undefined} \qquad\qquad \text{otherwise} \end{cases}$$

$$\tau_{\mathrm{T}}[\![\,x.f := aexp\,]\!]r \stackrel{\text{def}}{=} \lambda(e,s).\ \begin{cases} \sup\{r(e, s[(e(x), n) \leftarrow v]) + 1 \mid v \in [\![aexp]\!](e,s)\} \\ \qquad\qquad \exists v \in [\![aexp]\!](e,s) \wedge \\ \qquad \forall v \in [\![aexp]\!](e,s) \colon (e, s[(e(x), f) \leftarrow v]) \in \mathrm{dom}(r) \\ \text{undefined} \qquad\qquad \text{otherwise} \end{cases}$$

$$\tau_{\mathrm{T}}[\![\,x := rexp\,]\!]r \stackrel{\text{def}}{=} \lambda(e,s).\ \begin{cases} \sup\{r(e[x \leftarrow v], s) + 1 \mid v \in [\![rexp]\!](e,s)\} \\ \qquad\qquad \exists v \in [\![rexp]\!](e,s) \wedge \\ \qquad \forall v \in [\![rexp]\!](e,s) \colon (e[x \leftarrow v], s) \in \mathrm{dom}(r) \\ \text{undefined} \qquad\qquad \text{otherwise} \end{cases}$$

$$\tau_{\mathrm{T}}[\![\,x.f := rexp\,]\!]r \stackrel{\text{def}}{=} \lambda(e,s).\ \begin{cases} \sup\{r(e, s[(e(x), f) \leftarrow v]) + 1 \mid v \in [\![rexp]\!](e,s)\} \\ \qquad\qquad \exists v \in [\![rexp]\!](e,s) \wedge \\ \qquad \forall v \in [\![rexp]\!](e,s) \colon (e, s[(e(x), f) \leftarrow v]) \in \mathrm{dom}(r) \\ \text{undefined} \qquad\qquad \text{otherwise} \end{cases}$$

Figure 2.6: Termination semantics of instructions *stmt*. The new rules are highlighted in gray.

$$\tau_{\mathrm{T}}[\![\,\texttt{if}\ \mathit{bexp}\ \texttt{then}\ \mathit{stmt}_1\ \texttt{else}\ \mathit{stmt}_2\ \texttt{fi}\,]\!]r \stackrel{\text{def}}{=} F[r] \mathbin{\dot\cup} F_1[r] \mathbin{\dot\cup} F_2[r]$$

where

$S_1 \stackrel{\text{def}}{=} \tau_{\mathrm{T}}[\![\,\mathit{stmt}_1\,]\!]r$ and $S_2 \stackrel{\text{def}}{=} \tau_{\mathrm{T}}[\![\,\mathit{stmt}_2\,]\!]r$

$$F[r] \stackrel{\text{def}}{=} \lambda s \in \mathrm{dom}(S_1) \cap \mathrm{dom}(S_2).\ \begin{cases} \sup\{S_1(s)+1, S_2(s)+1\} & [\![\mathit{bexp}]\!]s = \{\mathrm{true}, \mathrm{false}\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$F_1[r] \stackrel{\text{def}}{=} \lambda s \in \mathrm{dom}(S_1).\ \begin{cases} S_1(s)+1 & [\![\mathit{bexp}]\!]s = \{\mathrm{true}\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$F_2[r] \stackrel{\text{def}}{=} \lambda s \in \mathrm{dom}(S_2).\ \begin{cases} S_2(s)+1 & [\![\mathit{bexp}]\!]s = \{\mathrm{false}\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\tau_{\mathrm{T}}[\![\,\texttt{while}\ \mathit{bexp}\ \texttt{do}\ \mathit{stmt}\ \texttt{od}\,]\!]r \stackrel{\text{def}}{=} \mathrm{lfp}^{\dot\sqsubseteq}_{\dot\emptyset}\ \phi_{\mathrm{T}}$$

where

$S \stackrel{\text{def}}{=} \tau_{\mathrm{T}}[\![\,\mathit{stmt}\,]\!]x$

$$\phi_{\mathrm{T}}(x) \stackrel{\text{def}}{=} F[x] \mathbin{\dot\cup} F_1[x] \mathbin{\dot\cup} F_2[r]$$

$$F[x] \stackrel{\text{def}}{=} \lambda s \in \mathrm{dom}(S) \cap \mathrm{dom}(r).\ \begin{cases} \sup\{S(s)+1, r(s)+1\} & [\![\mathit{bexp}]\!]s = \{\mathrm{true}, \mathrm{false}\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$F_1[x] \stackrel{\text{def}}{=} \lambda s \in \mathrm{dom}(S).\ \begin{cases} S(s)+1 & [\![\mathit{bexp}]\!]s = \{\mathrm{true}\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$F_2[r] \stackrel{\text{def}}{=} \lambda s \in \mathrm{dom}(r).\ \begin{cases} r(s)+1 & [\![\mathit{bexp}]\!]s = \{\mathrm{false}\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\tau_{\mathrm{T}}[\![\,\mathit{stmt}_1\ \mathit{stmt}_2\,]\!]r \stackrel{\text{def}}{=} \tau_{\mathrm{T}}[\![\,\mathit{stmt}_1\,]\!](\tau_{\mathrm{T}}[\![\,\mathit{stmt}_2\,]\!]r)$$

Figure 2.7: Termination semantics of instructions *stmt*.

Figure 2.6 and Figure 2.7 show the extended termination semantics $\tau_{\mathrm{T}}[\![\, stmt \,]\!]$ of instructions. It takes as input a ranking function and propagates it backwards within the program, taking into consideration the statement *stmt*.

The termination semantics of an assignment returns a ranking function that is defined over the states that, when subject to the assignment, always belong to the domain of the input ranking function (enforced by the check $\forall v \in [\![rexp]\!](e,s)\colon (e, s[(e(x), f) \in \mathrm{dom}(r))$. Note that the check $\exists v \in [\![rexp]\!](e,s)$ makes sure that the expression *rexp* returns a value. The value of the ranking function is increased by one, to take into account another execution step, and the value of the resulting ranking function is the least upper bound of these values.

The termination semantics of a loop instruction returns a ranking function that is defined as the least fixpoint of the function $\phi_{\mathrm{T}} : (\mathit{State} \to \mathbb{O}) \to (\mathit{State} \to \mathbb{O})$ within the partially ordered set $\langle \mathit{State} \to \mathbb{O}, \sqsubseteq \rangle$, where the *computational order* $\sqsubseteq$ is defined as:

$$r_1 \sqsubseteq r_2 \iff \mathrm{dom}(r_1) \subseteq \mathrm{dom}(r_2) \wedge \forall x \in \mathrm{dom}(r_1) : r_1(x) \leq r_2(x). \tag{2.2}$$

The function $\phi_{\mathrm{T}}\colon (\mathit{State} \to \mathbb{O}) \to (\mathit{State} \to \mathbb{O})$ takes as input a ranking function $x\colon \mathit{State} \to \mathbb{O}$ and adds to its domain the states for which one more loop iteration is needed before termination. For more details we refer to [Urb15b, Chapter 4].

The termination semantics $\tau_{\mathrm{T}}[\![\, prog \,]\!] \in \mathit{State} \rightharpoonup \mathbb{O}$ of a program *prog* is ranking function whose domain represents the terminating states, which is determined taking as input the zero function:

Definition 2.4.1 (Termination Semantics)**.** The *termination semantics* $\tau_{\mathrm{T}}[\![\, prog \,]\!] \in \mathit{State} \rightharpoonup \mathbb{O}$ of a program *prog* is:

$$\tau_{\mathrm{T}}[\![\, prog \,]\!] = \tau_{\mathrm{T}}[\![\, stmt \,]\!] \stackrel{\mathrm{def}}{=} \tau_{\mathrm{T}}[\![\, stmt \,]\!](\lambda s.\ 0) \tag{2.3}$$

where the function $\tau_{\mathrm{T}}[\![\, stmt \,]\!]\colon (\mathit{State} \rightharpoonup \mathbb{O}) \to (\mathit{State} \rightharpoonup \mathbb{O})$ is the termination semantics of each program instruction *stmt* defined in Figure 2.6 and Figure 2.7.

Remark. Note that possible run-time errors silently halting the program are ignored. More specifically, all states leading to run-time errors are discarded and do not belong to the domain of the termination semantics of a program *prog*.

The termination semantics $\tau_{\mathrm{T}}[\![\, prog \,]\!] \in \mathit{State} \rightharpoonup \mathbb{O}$ is usually not computable. In Chapter 4, we present sound decidable abstractions of $\tau_{\mathrm{T}}[\![\, prog \,]\!]$ by means of piecewise-defined functions. The soundness is related to the *approximation order* $\preccurlyeq$ defined as:

$$r_1 \preccurlyeq r_2 \iff \mathrm{dom}(r_1) \supseteq \mathrm{dom}(r_2) \wedge \forall x \in \mathrm{dom}(r_2) : r_1(x) \leq r_2(x) \tag{2.4}$$

which intuitively means that a ranking function r_1 is more precise than a ranking function r_2 when it is defined over a larger set of program states and when its value is always

smaller. This means that the ranking function r_1 is able to prove termination for more program states and in fewer steps than the ranking function r_2.

In the next chapter we introduce a heap abstract domain which is later used to represent the domain of a ranking function.

3 Heap Length Abstraction with Reachability

In this chapter, we present a heap abstraction of the invariance semantics presented in Section 2.3. The domain is later used to prove termination of heap manipulating programs. We do not define forward assignment or widening since they are not needed in the following chapters. To facilitate the definition of the transformers we assume that each statement or expression only contains a single field access[1].

3.1 Motivation & Intuition

We first give some motivation as well as intuition behind the abstract domain and then we formally define it. The main goal of the abstract domain is to be able to prove termination of programs manipulating lists where the termination arguments depends on the size of the list as seen in Example 1.0.1 and Example 1.0.2.

In Chapter 4 we will use the heap abstraction to represent the domain of a ranking function. For this the abstraction has to be precise enough to represent terminating states without also including non-terminating states.

We first analyze some example programs to see what kind of information we need to be able to partition the domain of the ranking function into terminating and non-terminating states. Figure 3.1 shows the terminating inputs for the traversal of Example 1.0.1.

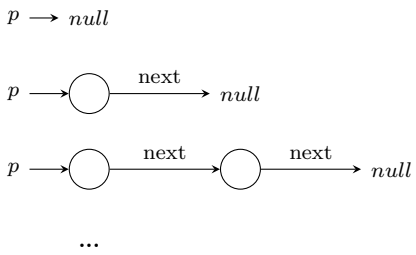


Figure 3.1: The terminating inputs for the list traversal of Example 1.0.1.

[1]This can always be achieved by rewriting the program.

To be able to represent the terminating states, we need the information that the input list is **acyclic** and has a specific **length**. Figure 3.2 shows part of the terminating states for the Josephus Problem shown in Example 1.0.2.

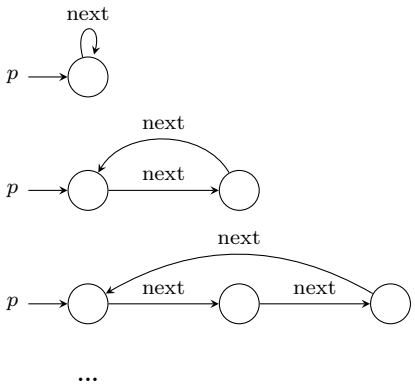


Figure 3.2: The terminating inputs for the Josephus Problem of Example 1.0.2.

Here we need the information that the input list is **cyclic** as well as the **length of the cycle**. Figure 3.3 shows that keeping only information about the length makes it impossible to differentiate between a heap configuration where two references p and q point to the same location and a heap configuration where two references point to different locations.

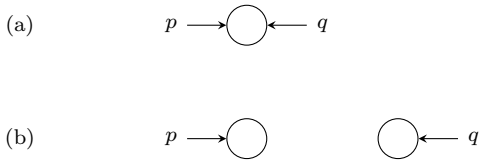


Figure 3.3: Two heap configurations (a) and (b) where the references p and q point to a list with a single element.

Both heap configurations (a) and (b) shown in Figure 3.3 can be abstracted using the information that p and q point to a list with a single element. But using only this information makes it impossible to distinguish between the two heap configurations (a) and (b). To be able to distinguish these cases, we will also keep track of reachability information between two references. In the heap configuration (a) of Figure 3.3 the references p and q can reach each other while in the heap configuration (b) they cannot reach each other.

3.2 Domain

We now define the abstract domain more formally. The abstract domain is parametric in the choice of the field $f \in \mathit{Field}$ for which information about the path is tracked. We first define the terms acyclic (cyclic) length and reachability and some auxiliary functions which allow us to access the according value given a concrete state.

Definition 3.2.1. Given a concrete state $s \in \mathit{State}$ and a reference $p \in \mathit{Ref}$ we define the *acyclic (cyclic) length* as the length of the acyclic (cyclic) part of the path starting from the node pointed to by p using only field f in s.

We will use the functions $a : \mathit{Ref} \times \mathit{Field} \times \mathit{State} \to \mathbb{N}$ and $c : \mathit{Ref} \times \mathit{Field} \times \mathit{State} \to \mathbb{N}$ to denote the acyclic length $a(p, f, s)$ and the cyclic length $c(p, f, s)$ respectively.

Definition 3.2.2. Given a concrete state $s \in \mathit{State}$ and two references $p, q \in \mathit{Ref}$ where $p \neq q$ we say that p *can reach* q if there exists a path using only the field f starting from p to the node pointed to by q in s.

We will use the function $r : \mathit{Ref} \times \mathit{Ref} \times \mathit{Field} \times \mathit{State} \to \{\mathit{true}, \mathit{false}\}$ to denote whether p can reach q.

To clarify, let us consider the following example:

Example 3.2.1. We consider a program with $\mathit{Ref} \stackrel{\text{def}}{=} \{p, q\}$ and $\mathit{Field} \stackrel{\text{def}}{=} \{f\}$. Let $s \in \mathit{State}$ be a state with the heap configuration shown in Figure 3.4.

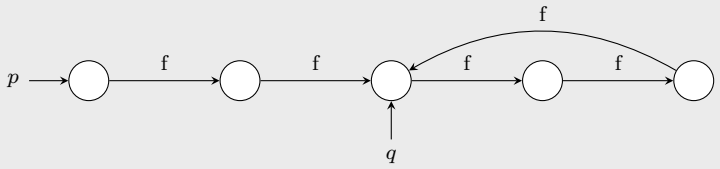


Figure 3.4: Heap configuration of the state s in Example 3.2.1.

By Definition 3.2.1 and 3.2.2 we have $a(p, f, s) = 2$, $c(p, f, s) = 3$, $a(q, f, s) = 0$, $c(q, f, s) = 3$, $r(p, q, f, s) = \mathit{true}$ and $r(q, p, f, s) = \mathit{false}$.

We now define two auxiliary structures which are then used to define the abstract domain.

Length Auxiliary Structure The length auxiliary structure is used to represent the length information per reference. For each reference $p \in \mathit{Ref}$ we store two intervals representing the lower and the upper bounds on the acyclic and cyclic length. More formally the elements of the length auxiliary structure belong to the following set:

$$\mathcal{L} \stackrel{\text{def}}{=} \{\bot_{\mathcal{L}}\} \cup \{([a_l, a_u], [c_l, c_u]) \mid a_l, c_l \in \mathbb{N}, a_u, c_u \in \mathbb{N} \cup \{\infty\}, a_l \leq a_u, c_l \leq c_u\} \quad (3.1)$$

which consists of the undefined length $\bot_{\mathcal{L}}$ and the defined lengths where a_l (c_l) represents the lower bound of the acyclic (cyclic) length and a_u (c_u) represents the upper bound of the acyclic (cyclic) length. We will use $\top_{\mathcal{L}}$ to denote the state $([0,\infty],[0,\infty]) \in \mathcal{L}$ which represents the length where the acyclic and cyclic length can be any non-negative number.

Reachability Auxiliary Structure For each pair of reference $p, q \in \mathit{Var}$ where $p \neq q$ we keep track of reachability information. The reachability auxiliary structure is used to represent this reachability information. The elements of the auxiliary structure belong to the following set:

$$\mathcal{R} \stackrel{\text{def}}{=} \{\bot_{\mathcal{R}}\} \cup \{\mathrm{R}, \mathrm{NR}\} \cup \{\top_{\mathcal{R}}\} \tag{3.2}$$

which consists of the undefined reachability $\bot_{\mathcal{R}}$ where p can neither reach nor not reach q, the reachable state `R` where p can reach q, the unreachable state NR where p cannot reach q and the top state $\top_{\mathcal{R}}$ where p can both reach or not reach q.

Heap Abstract Domain We now define the heap abstract domain based on the auxiliary structures. The elements of this domain belong to the following set:

$$\mathcal{D} \stackrel{\text{def}}{=} \{\bot_{\mathcal{D}}\} \cup ((\mathit{Var} \to \mathcal{L} \setminus \{\bot_{\mathcal{L}}\}) \times (\mathit{Var} \times \mathit{Var} \to \mathcal{R} \setminus \{\bot_{\mathcal{R}}\})) \tag{3.3}$$

which is a reduced product of the length auxiliary structure lifted to Var and the reachability auxiliary structure lifted to $\mathit{Var} \times \mathit{Var}$. The domain collapses all states having a $\bot_{\mathcal{L}}$ or $\bot_{\mathcal{R}}$ component into $\bot_{\mathcal{D}}$ which represents the undefined state.

3.2.1 Order

We now define a partial order for the heap abstract domain. To this end we first define partial orders for the auxiliary structures and then define a partial order for the heap abstraction based on the partial orders for the auxiliary structures.

Length Auxiliary Structure The partial order $\sqsubseteq_{\mathcal{L}}$ for the length auxiliary structure is defined as follows:

$$\begin{gathered} \bot_{\mathcal{L}} \sqsubseteq_{\mathcal{L}} l \stackrel{\text{def}}{=} \mathit{true} \\ l \sqsubseteq_{\mathcal{L}} \bot_{\mathcal{L}} \stackrel{\text{def}}{=} \mathit{false} \\ ([a_l, a_u], [c_l, c_u]) \sqsubseteq_{\mathcal{L}} ([a'_l, a'_u], [c'_l, c'_u]) \stackrel{\text{def}}{=} [a_l, a_u] \sqsubseteq_{\mathcal{I}} [a'_l, a'_u] \wedge [c_l, c_u] \sqsubseteq_{\mathcal{I}} [c'_l, c'_u] \end{gathered} \tag{3.4}$$

where $\sqsubseteq_{\mathcal{I}}$ is the usual partial order for intervals. Intuitively a length $l \in \mathcal{L}$ is smaller than another length $l' \in \mathcal{L}$ if the acyclic (cyclic) bound of l is contained in the acyclic (cyclic) bound of l'.

Reachability Auxiliary Structure The partial order $\sqsubseteq_{\mathcal{R}}$ for the reachability auxiliary structure is defined by the Hasse diagram shown in figure 3.5.

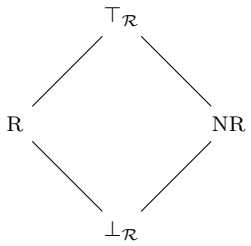


Figure 3.5: Hasse diagram defining the order $\sqsubseteq_{\mathcal{R}}$ of the reachability auxiliary structure.

Note that R and NR are not comparable since they are used to represent opposite states. Otherwise the order represents the fact that $\top_{\mathcal{R}}$ is either reachable or not reachable and that $\bot_{\mathcal{R}}$ is undefined.

Heap Abstract Domain Based on the order $\sqsubseteq_{\mathcal{L}}$ for the length information and the order $\sqsubseteq_{\mathcal{R}}$ for the reachability information, the order $\sqsubseteq_{\mathcal{D}}$ for the heap abstract domain is defined as follows:

$$\begin{aligned} \bot_{\mathcal{D}} &\sqsubseteq_{\mathcal{D}} d \stackrel{\text{def}}{=} \mathit{true} \\ d &\sqsubseteq_{\mathcal{L}} \bot_{\mathcal{D}} \stackrel{\text{def}}{=} \mathit{false} \\ (l_1, r_1) &\sqsubseteq_{\mathcal{D}} (l_2, r_2) \stackrel{\text{def}}{=} \forall p \in \texttt{Var}.\, l_1(p) \sqsubseteq_{\mathcal{L}} l_2(p) \\ &\qquad \wedge \forall p, q \in \texttt{Var}.\, p \neq q \implies r_1(p, q) \sqsubseteq_{\mathcal{R}} r_2(p, q) \end{aligned} \tag{3.5}$$

which means that an element $d_1 \in \mathcal{D}$ is smaller than another element $d_2 \in \mathcal{D}$ if and only if for each reference its length is smaller and for each pair of references its reachability is smaller.

To clarify, let us consider the following example:

Example 3.2.2. We consider a program with $\mathit{Ref} \stackrel{\text{def}}{=} \{p\}$ and $\mathit{Field} \stackrel{\text{def}}{=} \{f\}$. Given the following abstract states

$$\begin{aligned} d_1 &\stackrel{\text{def}}{=} (\{p \mapsto ([1,1],[2,3])\}, \emptyset)\} \\ d_2 &\stackrel{\text{def}}{=} (\{p \mapsto ([0,3],[3,5])\}, \emptyset)\} \\ d_3 &\stackrel{\text{def}}{=} (\{p \mapsto ([0,3],[1,4])\}, \emptyset)\} \end{aligned}$$

we have $d_1 \not\sqsubseteq_{\mathcal{D}} d_2$, $d_1 \sqsubseteq_{\mathcal{D}} d_3$ and $d_2 \not\sqsubseteq_{\mathcal{D}} d_3$. $d_1 \not\sqsubseteq_{\mathcal{D}} d_2$ does not hold since d_1 allows a cyclic length of 2 which is not possible in d_2 and $d_2 \not\sqsubseteq_{\mathcal{D}} d_3$ does not hold since d_2 allows

a cyclic length of 5 which is not possible in d_3.

3.2.2 Concretization

We now define the concretization function. To this end, we first define the utility function $h_{\mathcal{L}} \colon \mathit{Ref} \to \mathcal{D} \to \mathcal{P}(\mathbb{N} \times \mathbb{N})$:

$$h_{\mathcal{L}}(p, (l, r)) \stackrel{\text{def}}{=} \{(a, c) \mid ([a, a][c, c]) \sqsubseteq_{\mathcal{L}} l(p)\} \tag{3.6}$$

which, given a reference $p \in \mathit{Ref}$ and an element $d \in \mathcal{D}$, returns a set of all possible pairs of acyclic and cyclic lengths $h_{\mathcal{L}}(p, d)$ represented by the length information of p in d.

We also define the utility function $h_{\mathcal{R}} \colon \mathit{Ref} \to \mathit{Ref} \to \mathcal{D} \to \mathcal{P}(\{\mathit{true}, \mathit{false}\})$:

$$h_{\mathcal{R}}(p, q, (l, r)) \stackrel{\text{def}}{=} \begin{cases} \{\mathit{true}, \mathit{false}\} & \text{if } r(p, q) = \top_{\mathcal{R}} \\ \{\mathit{true}\} & \text{if } r(p, q) = \mathrm{R} \\ \{\mathit{false}\} & \text{if } r(p, q) = \mathrm{NR} \\ \emptyset & \text{if } r(p, q) = \bot_{\mathcal{R}} \end{cases} \tag{3.7}$$

which, given two references $p, q \in \mathit{Ref}$ where $p \neq q$ and an element $d \in \mathcal{D}$, returns a set $h_{\mathcal{L}}(p, d)$ containing *true* if it is possible for p to reach q in d and *false* if it is possible for p not to reach q in d.

Based on this the concretization function $\gamma_{\mathcal{D}} : \mathcal{D} \to \mathcal{P}(\mathit{State})$ is defined as follows:

$$\begin{aligned} \gamma_{\mathcal{D}}(\bot_{\mathcal{D}}) &\stackrel{\text{def}}{=} \emptyset \\ \gamma_{\mathcal{D}}(d) &\stackrel{\text{def}}{=} \{s \in \mathit{State} \mid \forall p \in \mathit{Var}.\ (a(p, f, s), c(p, f, s)) \in h_{\mathcal{L}}(p, d)\} \\ &\quad \cap \{s \in \mathit{State} \mid \forall p, q \in \mathit{Var}.\ p \neq q \implies r(p, q, f, s) \in h_{\mathcal{R}}(p, q, d)\} \end{aligned} \tag{3.8}$$

where a, c and r are defined as in Definition 3.2.1 and 3.2.2. Intuitively this means that the concretization of an element $d \in \mathcal{D}$ returns all states $s \in \mathit{State}$ such that the constraints described by the length and reachability information in d are satisfied in the heap configuration of s. For this the auxiliary function a, c and r are used to check that the acyclic length, the cyclic length and the reachability of all references in s are possible in d using the two utility functions $h_{\mathcal{L}}$ and $h_{\mathcal{R}}$.

To clarify, let us consider the following example:

Example 3.2.3. We consider a program with $\mathit{Ref} \stackrel{\text{def}}{=} \{p, q\}$ and $\mathit{Field} \stackrel{\text{def}}{=} \{f\}$. Given the abstract state $d \stackrel{\text{def}}{=} (\{p \mapsto ([1, 3], [1, 1]), q \mapsto ([0, 2], [1, 1])\}, \{(p, q) \mapsto \mathrm{R}, (q, p) \mapsto \mathrm{NR}\})$ we

have

$$\begin{aligned} h_{\mathcal{L}}(p,d) &= \{(1,1),(2,1),(3,1)\} \\ h_{\mathcal{L}}(q,d) &= \{(0,1),(1,1),(2,1)\} \\ h_{\mathcal{R}}(p,q,d) &= \{true\} \\ h_{\mathcal{R}}(q,q,d) &= \{false\} \end{aligned}$$

which means that $\gamma_{\mathcal{D}}(d)$ contains concrete states where the acyclic length of p is either one, two or three, the cyclic length of p is always one, the acyclic length of q is either zero, one or two, the cyclic length of q is always one, p always reaches q and q never reaches p. Figure 3.6 shows an example of such a heap.

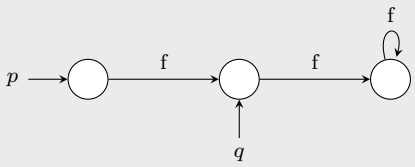


Figure 3.6: An example of a heap configuration satisfying the conditions of Example 3.2.3.

Lemma 3.2.1. The concretization function $\gamma_{\mathcal{D}}$ is monotonic, i.e., $\forall d_1, d_2 \in \mathcal{D}\colon d_1 \sqsubseteq_{\mathcal{D}} d_2 \implies \gamma_{\mathcal{D}}(d_1) \subseteq \gamma_{\mathcal{D}}(d_2)$.

Proof. By contradiction. We assume $\exists d_1, d_2 \in \mathcal{D}.\ d_1 \sqsubseteq_{\mathcal{D}} d_2 \wedge \gamma_{\mathcal{D}}(d_1) \supset \gamma_{\mathcal{D}}(d_2)$. This means that by definition of $\gamma_{\mathcal{D}}$ there is some element in $\gamma_{\mathcal{D}}(d_1)$ which has a length or reachability that is not possible in d_2 which is a contradiction by the definition of $\sqsubseteq_{\mathcal{D}}$ and the assumption that $d_1 \sqsubseteq_{\mathcal{D}} d_2$. □

3.3 Binary Operators

3.3.1 Join

We now define the join operator for the heap abstract domain. To this end we first define join operators for the auxiliary structures and then define the join operator for the heap abstraction based on the join operators for the auxiliary structures.

Length Auxiliary Structure The join operator for the length auxiliary structure $\sqcup_{\mathcal{L}}$ is defined as follows:

$$\begin{aligned} \bot_{\mathcal{L}} \sqcup_{\mathcal{L}} l &\stackrel{\text{def}}{=} l \\ l \sqcup_{\mathcal{L}} \bot_{\mathcal{L}} &\stackrel{\text{def}}{=} l \\ ([a_l,a_u],[c_l,c_u]) \sqcup_{\mathcal{L}} ([a'_l,a'_u],[c'_l,c'_u]) &\stackrel{\text{def}}{=} ([a_l,a_u] \sqcup_I [a'_l,a'_u],[c_l,c_u] \sqcup_I [c'_l,c'_u]) \end{aligned} \tag{3.9}$$

where $\sqcup_I$ is the join for intervals. The join of the undefined length $\bot_{\mathcal{L}}$ and a defined length $l \in \mathcal{L}$ is l. The join of two defined lengths $l, l' \in \mathcal{L}$ is a length where the bounds for the acyclic and cyclic lengths is the least upper bound of the acyclic and cyclic lengths of l and l'.

Reachability Auxiliary Structure The join operator for the reachability auxiliary structure $\sqcup_{\mathcal{R}}$ is defined by the Hasse diagram shown in figure 3.5.

The join of the undefined reachability $\bot_{\mathcal{R}}$ and any reachability $r \in \mathcal{R}$ is r. Since R and NR are not comparable their join is $\top_{\mathcal{R}}$. The join of $\top_{\mathcal{L}}$ and any reachability $r \in \mathcal{R}$ is $\top_{\mathcal{L}}$.

Heap Abstract Domain We now define the join operator $\sqcup_D$ by lifting $\sqcup_{\mathcal{L}}$ to *Ref* and $\sqcup_{\mathcal{R}}$ to $\mathit{Ref} \times \mathit{Ref}$:

$$\begin{aligned} \bot_{\mathcal{D}} \sqcup_D d &\stackrel{\text{def}}{=} d \\ d \sqcup_D \bot_{\mathcal{D}} &\stackrel{\text{def}}{=} d \\ (l, r) \sqcup_D (l', r') &\stackrel{\text{def}}{=} (l'', r'') \end{aligned} \tag{3.10}$$

where $l'' = \{p \mapsto (l(p) \sqcup_{\mathcal{L}} l'(p)) \mid p \in \mathit{Ref}\}$ and $r'' = \{(p, q) \mapsto (r(p, q) \sqcup_{\mathcal{R}} r'(p, q)) \mid p, q \in \mathit{Ref}, p \neq q\}$. The join of the undefined element $\bot_{\mathcal{D}}$ and any other element $d \in \mathcal{D}$ is d. The join of two defined elements $d, d' \in \mathcal{D}$ is the element d'' where for each reference $p \in \mathit{Ref}$ the length of p in d'' is the join of the length of p in d and the length of p in d', and for all references $p, q \in \mathit{Ref}$ where $p \neq q$ the reachability of p to q is the join of the reachability of p to q in d and the reachability of p to q in d'.

To clarify, let us consider the following example:

Example 3.3.1. We consider a program with $\mathit{Ref} \stackrel{\text{def}}{=} \{p, q\}$ and $\mathit{Field} \stackrel{\text{def}}{=} \{f\}$. Given the following abstract states

$$\begin{aligned} d_1 &\stackrel{\text{def}}{=} (\{p \mapsto ([0, 1], [1, 3]), q \mapsto ([1, 1], [0, 0])\}, \{(p, q) \mapsto \mathrm{R}, (q, p) \mapsto \mathrm{R}\}) \\ d_2 &\stackrel{\text{def}}{=} (\{p \mapsto ([4, 7], [2, 5]), q \mapsto ([3, 3], [2, 2])\}, \{(p, q) \mapsto \mathrm{R}, (q, p) \mapsto \mathrm{NR}\}) \end{aligned}$$

we have

$$d_1 \sqcup_D d_2 = (\{p \mapsto ([0, 7], [1, 5]), q \mapsto ([1, 3], [0, 2])\}, \{(p, q) \mapsto \mathtt{R}, (q, p) \mapsto \top_{\mathcal{R}}\})$$

The reachability for (q, p) becomes $\top_{\mathcal{R}}$ since p is reachable from q in d_1 and p is not reachable from q in d_2.

Lemma 3.3.1. The join operator $\sqcup_D$ is sound, i.e., $\forall d_1, d_2 \in \mathcal{D}.\ \gamma_{\mathcal{D}}(d_1) \cup \gamma_{\mathcal{D}}(d_2) \subseteq \gamma_{\mathcal{D}}(d_1 \sqcup_D d_2)$.

Proof. By contradiction. We assume $\exists d_1, d_2 \in \mathcal{D}.\ \gamma_{\mathcal{D}}(d_1) \cup \gamma_{\mathcal{D}}(d_2) \supset \gamma_{\mathcal{D}}(d_1 \sqcup_D d_2)$. Without loss of generality we assume that there exists an element $c \in \gamma_{\mathcal{D}}(d_1)$ which is not in $\gamma_{\mathcal{D}}(d_1 \sqcup_D d_2)$. By the definition of $\gamma_{\mathcal{D}}$ this means that there exists a reference $p \in \mathit{Var}$ such that its length is greater than possible in $\gamma_{\mathcal{D}}(d_1 \sqcup_D d_2)$ or its reachability is not valid in $\gamma_{\mathcal{D}}(d_1 \sqcup_D d_2)$. Both are not possible by the definitions of $\sqcup_{\mathcal{L}}$ and $\sqcup_{\mathcal{R}}$. □

3.3.2 Meet

We now define the meet operator for the heap abstract domain. To this end we first define meet operators for the auxiliary structures and then define the meet operator for the heap abstraction based on the meet operators for the auxiliary structures.

Length Auxiliary Structure The meet operator for the length auxiliary structure $\sqcap_{\mathcal{L}}$ is defined as follows:

$$\begin{aligned} \bot_{\mathcal{L}} \sqcap_{\mathcal{L}} l &\stackrel{\text{def}}{=} \bot_{\mathcal{L}} \\ l \sqcap_{\mathcal{L}} \bot_{\mathcal{L}} &\stackrel{\text{def}}{=} \bot_{\mathcal{L}} \\ ([a_l, a_u], [c_l, c_u]) \sqcap_{\mathcal{L}} ([a'_l, a'_u], [c'_l, c'_u]) &\stackrel{\text{def}}{=} ([a_l, a_u] \sqcap_I [a'_l, a'_u], [c_l, c_u] \sqcap_I [c'_l, c'_u]) \end{aligned} \tag{3.11}$$

where $\sqcap_I$ is the meet for intervals. The meet of the undefined length $\bot_{\mathcal{L}}$ and a defined length $l \in \mathcal{L}$ is $\bot_{\mathcal{L}}$. The meet of two defined lengths $l, l' \in \mathcal{L}$ is a length where the bounds for the acyclic and cyclic lengths is the greatest lower bound of the acyclic and cyclic lengths of l and l'.

Reachability Auxiliary Structure The meet operator for the reachability auxiliary structure $\sqcap_{\mathcal{R}}$ is defined by the Hasse diagram shown in figure 3.5.

The meet of the undefined reachability $\bot_{\mathcal{R}}$ and any reachability $r \in \mathcal{R}$ is $\bot_{\mathcal{R}}$. Since R and NR are not comparable their meet is $\bot_{\mathcal{R}}$. The meet of $\top_{\mathcal{R}}$ and any reachability $r \in \mathcal{R}$ is r.

Heap Abstract Domain We now define the meet operator $\sqcap_D$ by lifting $\sqcap_{\mathcal{L}}$ to *Ref* and $\sqcap_{\mathcal{R}}$ to *Ref* $\times$ *Ref*:

$$\begin{aligned} \bot_{\mathcal{D}} \sqcap_D d &\stackrel{\text{def}}{=} \bot_{\mathcal{D}} \\ d \sqcap_D \bot_{\mathcal{D}} &\stackrel{\text{def}}{=} \bot_{\mathcal{D}} \\ (l, r) \sqcap_D (l', r') &\stackrel{\text{def}}{=} (l'', r'') \end{aligned} \tag{3.12}$$

where $l'' = \{p \mapsto (l(p) \sqcap_{\mathcal{L}} l'(p)) \mid p \in \mathit{Ref}\}$ and $r'' = \{(p, q) \mapsto (r(p, q) \sqcap_{\mathcal{R}} r'(p, q)) \mid p, q \in \mathit{Ref}, p \neq q\}$. The meet of the undefined element $\bot_{\mathcal{D}}$ and any other element $d \in \mathcal{D}$ is $\bot_{\mathcal{D}}$. The meet of two defined elements $d, d' \in \mathcal{D}$ is the element d'' where for each reference

$p \in \mathit{Ref}$ the length of p in d'' is the meet of the length of p in d and the length of p in d', and for all references $p, q \in \mathit{Ref}$ where $p \neq q$ the reachability of p to q is the meet of the reachability of p to q in d and the reachability of p to q in d'.

To clarify, let us consider the following example:

Example 3.3.2. We consider a program with $\mathit{Ref} \stackrel{\text{def}}{=} \{p, q\}$ and $\mathit{Field} \stackrel{\text{def}}{=} \{f\}$. Given the following abstract states

$$d_1 \stackrel{\text{def}}{=} (\{p \mapsto ([0,1],[1,3]), q \mapsto ([1,1],[0,0])\}, \{(p,q) \mapsto \mathrm{R}, (q,p) \mapsto \mathrm{R}\})$$
$$d_2 \stackrel{\text{def}}{=} (\{p \mapsto ([4,7],[2,5]), q \mapsto ([3,3],[2,2])\}, \{(p,q) \mapsto \mathrm{R}, (q,p) \mapsto \mathrm{NR}\})$$

we have $d_1 \sqcap_D d_2 = \bot_{\mathcal{D}}$ since the acyclic length of p in d_1 and d_2 have no overlap. Note that the result is $\bot_{\mathcal{D}}$ since all elements where one component is bottom get collapsed to $\bot_{\mathcal{D}}$ (cf. Section 3.2).

Lemma 3.3.2. The meet operator $\sqcap_D$ is sound, i.e., $\forall d_1, d_2 \in \mathcal{D}.\ \gamma_{\mathcal{D}}(d_1) \cap \gamma_{\mathcal{D}}(d_2) \subseteq \gamma_{\mathcal{D}}(d_1 \sqcap_D d_2)$.

Proof. Analogous to the proof of Lemma 3.3.1. □

3.4 Unary Operators

We first show how partitioning can improve the precision of the transformers of backward assignment and filter and then formally define the transformers. To facilitate the definition of the transformer we assume in the following that $\mathit{Num} = \emptyset$, i.e., $\mathit{Var} = \mathit{Ref}$. This means that programs have no variables of type integer and we only have to define the transformers for statements over references.

3.4.1 Fixed partitions

Some transformers such as $p \neq \mathit{null}$ need to be able to represent the complement of a specific length, in this case for example the complement of $([0,0],[0,0])$ which is not possible, except by using $\top_{\mathcal{L}}$. This makes transformers for those statements very imprecise and unusable in practice.

To clarify, let us consider the following example:

Example 3.4.1. Let $l = ([0,1],[0,1]) \in \mathcal{L}$ be a length. Figure 3.7 (a) shows l represented as box and Figure 3.7 (b) shows its complement l^C.

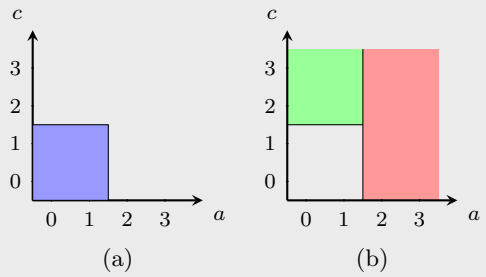


Figure 3.7: The length l and its complement l^C of Example 3.4.1 represented as boxes in (a) and (b) respectively.

To represent l^C we need at least two elements of the heap abstract domain. The most precise over-approximation of l^C using a single element in $\mathcal{L}$ is $\top_{\mathcal{L}}$.

To improve the precision, we will later partition the domain for each length l as shown in Figure 3.8.

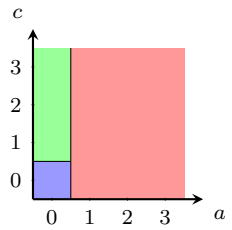


Figure 3.8: Partitions of a length l to improve precision of statements that require the complement of l.

For each length l we have fixed partitions for *null* (shown in blue, l is in $([0,0],[0,0])$), for only cycles (shown in green, l is in $([0,0],[1,\infty])$) and for the rest (shown in red, l is in $([1,\infty],[0,\infty])$). In the definition of the transformers we do not assume these partitions exist, but instead define the transformers for the fixed partitions as well as the general case. Because of this the transformers for backward assignment and filter can be used for any element, albeit being less precise.

3.4.2 Backward Assignment

We now define the backward assignment operator $\texttt{B-ASSIGN}_{\mathcal{D}} : \mathcal{D} \to \mathcal{D}$ to handle backward assignments by induction on the syntax of the language defined in Section 2.1.

Reference manipulating statements

We first define the transformers for statements that only manipulate references. We first prove a lemma which allows simpler reasoning for statements that only manipulate references.

Lemma 3.4.1. Statements that only manipulate references, i.e., statements of the form $p := rexp$, only change the length information on p as well as the reachability information concerning p.

Proof. Follows directly from the definition of the semantics of the statement $p := rexp$ (cf. Figure 2.5), the definition of the semantics of reference expressions (cf. Figure 2.3) and the definition of length and reachability (cf. Definition 3.2.1 and 3.2.2). □

Because of Lemma 3.4.1 we will only define and prove changes made to the length and reachabilities of references manipulated by the transformer. In the following we assume the input is not $\bot_{\mathcal{D}}$, otherwise we simply return $\bot_{\mathcal{D}}$. We also assume that field dereference expressions such as $p.f$ where $p \in \mathit{Ref}$ and $f \in \mathit{Field}$ only use the field given as the parameter, otherwise we simply return $\top_{\mathcal{D}}$. Note that this means that the domain is very imprecise when used for programs that use multiple fields.

$\texttt{B-ASSIGN}_{\mathcal{D}}[\![\, p := \textbf{null} \,]\!]$ The statement $p := \mathit{null}$ gives us the information that p is *null* in the post state which means that the acyclic and cyclic lengths of p are zero and that p neither reaches or is reachable by any other reference. If this is not the case, we are in an invalid trace and no pre state exists. The transformer is defined as follows:

$$\texttt{B-ASSIGN}_{\mathcal{D}}[\![\, p := \mathit{null}]\!](l, r) \stackrel{\text{def}}{=} \begin{cases} (l', r') & \text{if } ([0,0],[0,0]) \sqsubseteq_{\mathcal{L}} l(p) \text{ and} \\ & \forall q \in \mathit{Ref}.\ \mathrm{NR} \sqsubseteq_{\mathcal{R}} r(p,q) \text{ and} \\ & \forall q \in \mathit{Ref}.\ \mathrm{NR} \sqsubseteq_{\mathcal{R}} r(q,p) \\ \bot_{\mathcal{D}} & \text{otherwise} \end{cases} \tag{3.13}$$

where $l' = l[p \leftarrow \top_{\mathcal{L}}]$ and $r' = r[(p,q) \leftarrow \top_{\mathcal{R}}, (q,p) \leftarrow \top_{\mathcal{R}}\ \forall q \in \mathit{Var}]$.

Remark. The most precise result for the length and reachability of p is top since the statement does not give us any information about p in the pre state.

To clarify, let us consider the following examples:

Example 3.4.2. We consider a program with $\mathit{Ref} \stackrel{\text{def}}{=} \{p\}$ and $\mathit{Field} \stackrel{\text{def}}{=} \{f\}$. Let $d = (\{p \mapsto ([0,\infty],[0,0])\}, \emptyset)$ be a post state. The result of the backwards assignment $\texttt{B-ASSIGN}_{\mathcal{D}}[\![\, p := \mathit{null}]\!](d)$ is $(\{p \mapsto \top_{\mathcal{L}}\}, \emptyset)$. This example shows that even though d abstracts invalid states (every heap configuration where p does not point to *null*) the

result of the transformer has to include every concrete state that leads to one of the concrete states abstracted by d.

Example 3.4.3. We consider a program with $\mathit{Ref} \stackrel{\text{def}}{=} \{p, q\}$ and $\mathit{Field} \stackrel{\text{def}}{=} \{f\}$. Let $d = (\{p \mapsto ([1,1],[0,0]), q \mapsto ([1,3],[0,2])\}, \{(p,q) \mapsto \mathtt{R}, (q,p) \mapsto \top_{\mathcal{R}}\}$ be a post state. Since p does not point to *null* in d we are in an invalid state and we have $\mathtt{B\text{-}ASSIGN}_{\mathcal{D}}[\![p := \mathit{null}]\!](d) = \bot_D$.

Lemma 3.4.2. The backward assignment operator $\mathtt{B\text{-}ASSIGN}_{\mathcal{D}}[\![p := \mathit{null}]\!]$ is a sound over-approximation of $\tau_I[\![p := \mathit{null}]\!]$, i.e., given a post state $d \in \mathcal{D}$ of the assignment $p := \mathit{null}$ we have $\{s \in \mathit{State} \mid \tau_I[\![p := \mathit{null}]\!](s) \subseteq \gamma_{\mathcal{D}}(d))\} \subseteq \gamma_D(\mathtt{B\text{-}ASSIGN}_{\mathcal{D}}[\![p := \mathit{null}]\!](d))$.

Proof. Let $d \in \mathcal{D}$ be a post state. We argue by cases whether $p = \mathit{null}$ is possible in d.

In the first case we assume that $p = \mathit{null}$ is possible in $\gamma_{\mathcal{D}}(d)$. This means that the length and reachability of p are top in the pre state (cf. Equation 3.13). Together with Lemma 3.4.1 the inclusion trivially holds.

In the second case we assume that $p = \mathit{null}$ is not possible in $\gamma_{\mathcal{D}}(d)$. This means that the pre state is $\bot_D$ (cf. Equation 3.13). We assume, by absurd, that the inclusion does not hold. This means there is some state $s \in \mathit{State}$ such that $\tau_I[\![p := \mathit{null}]\!](s)$ leads to a state where the length of p is not $([0,0],[0,0])$ or p can reach some other variable or some other variable can reach p which is not possible according to the semantics (cf. Section 2.3). Thus the inclusion holds. □

$\mathtt{B\text{-}ASSIGN}_{\mathcal{D}}[\![p := \mathbf{new}]\!]$ The statement $p := \mathit{new}$ gives us the information that p is a new reference in the post state which means that the acyclic length of p is one, the cyclic length of p is zero and that p neither reaches or is reachable by any other reference. If this is not the case, we are in an invalid trace and no pre state exists. The transformer is defined as follows:

$$\mathtt{B\text{-}ASSIGN}_{\mathcal{D}}[\![p := \mathit{new}]\!](l,r) \stackrel{\text{def}}{=} \begin{cases} (l', r') & \text{if } ([1,1],[0,0]) \sqsubseteq_{\mathcal{L}} l(p) \text{ and} \\ & \forall q \in \mathit{Ref}.\ \mathrm{NR} \sqsubseteq_{\mathcal{R}} r(p,q) \text{ and} \\ & \forall q \in \mathit{Ref}.\ \mathrm{NR} \sqsubseteq_{\mathcal{R}} r(q,p) \\ \bot_{\mathcal{D}} & \text{otherwise} \end{cases} \tag{3.14}$$

where $l' = l[p \leftarrow \top_{\mathcal{L}}]$ and $r' = r[(p,q) \leftarrow \top_{\mathcal{R}}, (q,p) \leftarrow \top_{\mathcal{R}}\ \forall q \in \mathit{Var}]$.

Remark. This transformers is analogous to the transformer for $p = \mathit{null}$, the only difference is the length of p in the post state.

Lemma 3.4.3. The backward assignment operator $\mathtt{B\text{-}ASSIGN}_{\mathcal{D}}[\![p := \mathit{new}]\!]$ is a sound over-approximation of $\tau_I[\![p := \mathit{new}]\!]$, i.e., given a post state $d \in \mathcal{D}$ of the assignment $p := \mathit{null}$ we have $\{s \in \mathit{State} \mid \tau_I[\![p := \mathit{new}]\!](s) \subseteq \gamma_{\mathcal{D}}(d))\} \subseteq \gamma_D(\mathtt{B\text{-}ASSIGN}_{\mathcal{D}}[\![p := \mathit{new}]\!](d))$.

Proof. Analogous to the proof of Lemma 3.4.2. □

$\texttt{B-ASSIGN}_{\mathcal{D}}[\![p := q]\!]$ The assignment $p := q$ gives us the information that p and q point to the same location in the heap. Since q is not changed by the statement $p := q$ we can use the information on p to refine the information on q in the pre state. The transformer is defined as follows:

$$\texttt{B-ASSIGN}_{\mathcal{D}}[\![p := q]\!](l, r) \stackrel{\text{def}}{=} (l', r') \tag{3.15}$$

where $l' = l[p \leftarrow \top_{\mathcal{L}}, q \leftarrow l(p) \sqcap_{\mathcal{L}} l(q)]$ and $r' = r[(p,t) \leftarrow \top_{\mathcal{R}}, (t,p) \leftarrow \top_{\mathcal{R}} \; \forall t \in \mathit{Ref} \backslash \{p\}]$.

To clarify, let us consider the following example:

Example 3.4.4. We consider a program with $\mathit{Ref} \stackrel{\text{def}}{=} \{p, q\}$ and $\mathit{Field} \stackrel{\text{def}}{=} \{f\}$. Let $d = (\{p \mapsto ([0,5],[1,2]), q \mapsto ([1,3],[1,6])\}, \{(p,q) \mapsto \top_{\mathcal{R}}, (q,p) \mapsto \top_{\mathcal{R}}\})$ be a post state. Because of the statement $p := q$ we know that p and q point to the same location in the heap. Since both $([0,5],[1,2])$ and $([1,3],[1,6])$ are valid over-approximations of the length of the path pointed to by either p or q we can refine the information on q in the pre state as shown in the result $\texttt{B-ASSIGN}_{\mathcal{D}}[\![p := q]\!](d) = (\{p \mapsto \top_{\mathcal{L}}, q \mapsto ([1,3],[1,2])\}, \{(p,q) \mapsto \top_{\mathcal{R}}, (q,p) \mapsto \top_{\mathcal{R}}\})$.

Lemma 3.4.4. The backward assignment operator $\texttt{B-ASSIGN}_{\mathcal{D}}[\![p := q]\!]$ is a sound over-approximation of $\tau_I[\![p := q]\!]$, i.e., given a post state $d \in \mathcal{D}$ of the assignment $p := q$ we have $\{s \in \mathit{State} \mid \tau_I[\![p := q]\!](s) \subseteq \gamma_{\mathcal{D}}(d))\} \subseteq \gamma_D(\texttt{B-ASSIGN}_{\mathcal{D}}[\![p := q]\!](d))$.

Proof. Let $(l, r) \in \mathcal{D}$ be a post state. Since p and q point to the same location in the post state both $l(p)$ and $l(q)$ are valid over-approximation of the same length which means that $l(p) \sqcap_{\mathcal{L}} l(q)$ is also a valid over-approximation. Based on this and the fact that the length and reachabilities concerning p become $\top_{\mathcal{L}}$ and $\top_{\mathcal{R}}$ respectively we can conclude the proof using Lemma 3.4.1. □

For the remaining transformers we first define a utility function f^{-1} which, given a length $l \in \mathcal{L}$, returns the possible lengths which may lead to l when following the field f once. To facilitate the definition of f^{-1} we make use of the fixed partitions (cf. Section 3.4.1).

Definition 3.4.1. The function $f^{-1} : \mathcal{L} \to \mathcal{L}$ is defined as follows

$$f^{-1}([a_l, a_u][c_l, c_u]) \stackrel{\text{def}}{=} \begin{cases} ([1,1][0,0]) & \text{if } a_u = 0 \wedge c_u = 0 \\ ([0,1][c_l, c_u]) & \text{if } a_u = 0 \wedge c_l > 0 \\ ([a_l + 1, a_u + 1][c_l, c_u]) & \text{if } a_l \geq 1 \\ \top_L & \text{otherwise} \end{cases} \tag{3.16}$$

$\texttt{B-ASSIGN}_{\mathcal{D}}[\![p := p.f]\!]$ The statement $p := p.f$ gives us the information that the reference p points to the same location that $p.f$ pointed to in the pre state, i.e., p moved one step following the field f. The transformer is defined as follows:

$$\texttt{B-ASSIGN}_{\mathcal{D}}[\![p := q]\!](l, r) \stackrel{\text{def}}{=} (l', r') \tag{3.17}$$

where $l' = l[p \leftarrow f^{-1}(l(p))]$ and $r' = r[(t, p) \leftarrow \top_{\mathcal{R}} \; \forall t \in \mathit{Var}]$.

To clarify, let us consider the following example which illustrates the third case of f^{-1}:

Example 3.4.5. We consider a program with $\mathit{Ref} \stackrel{\text{def}}{=} \{p\}$ and $\mathit{Field} \stackrel{\text{def}}{=} \{f\}$. Let $d = (\{p \mapsto ([1,1],[3,3])\}, \emptyset)$ be a post state. Figure 3.9 (a) shows a possible concrete heap configuration in $\gamma_{\mathcal{D}}(d)$ and the only heap configuration which leads to this heap configuration when applying the statement $p = p.f$ (b).

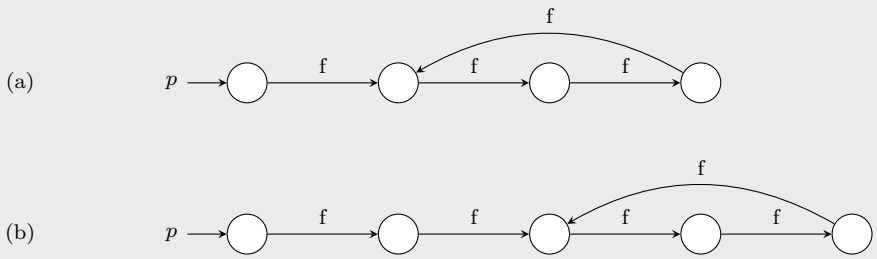


Figure 3.9: A heap configuration in $\gamma_{\mathcal{D}}(d)$ (a) and the only heap configuration that leads to (a) when applying the statement $p := p.f$ (b).

Since the paths pointed to by p in d always have some acyclic part (the lower bound of the acyclic part is one) the length of the path always increases when applying the backwards transformer for $p := p.f$. This situation is handled by the third case in the function f^{-1} (cf. Definition 3.4.1). The result of the backward assignment $\texttt{B-ASSIGN}_{\mathcal{D}}[\![p := p.f]\!](d)$ is the state $(\{p \mapsto ([2,2],[3,3])\}, \emptyset)$ which correctly over-approximates the heap configuration shown in Figure 3.9 (b).

The following example illustrates the second case of f^{-1}:

Example 3.4.6. We consider a program with $\mathit{Ref} \stackrel{\text{def}}{=} \{p\}$ and $\mathit{Field} \stackrel{\text{def}}{=} \{f\}$. Let $d = (\{p \mapsto ([0,0],[3,3])\}, \emptyset)$ be a post state. Figure 3.10 (a) shows a possible heap configuration in $\gamma_{\mathcal{D}}(d)$ and the heap configurations which lead to this heap configuration when applying the statement $p = p.f$ (b) and (c).

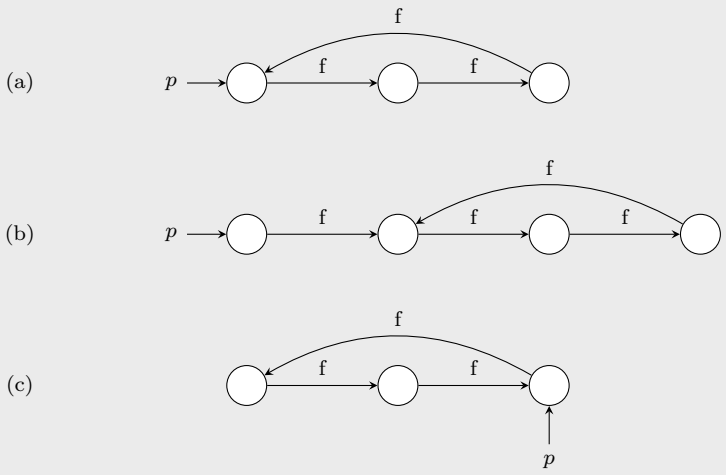


Figure 3.10: A heap configuration in $\gamma_{\mathcal{D}}(a)$ (a) and heap configurations that lead to (a) when applying the statement $p := p.f$ (b) and (c).

Since the paths pointed to by p in d have no acyclic part (the upper bound of the acyclic length is zero) there are two possible heap configurations (b) and (c) that lead to the heap configuration (a). This situation is handled by the second case in the function f^{-1} (cf. Definition 3.4.1). The result of the backward assignment $\texttt{B-ASSIGN}_{\mathcal{D}}[\![\, p := p.f \,]\!](d)$ is the state $(\{p \mapsto ([0,1],[3,3])\}, \emptyset)$ which correctly over-approximates the heap configurations shown in Figure 3.10 (b) and (c).

Lemma 3.4.5. The backward assignment operator $\texttt{B-ASSIGN}_{\mathcal{D}}[\![\, p := p.f \,]\!]$ is a sound over-approximation of $\tau_I[\![p := p.f]\!]$, i.e., given a post state $d \in \mathcal{D}$ of the assignment $p := p.f$ we have $\{s \in \mathit{State} \mid \tau_I[\![^l p := p.f,]\!](s) \subseteq \gamma_{\mathcal{D}}(d))\} \subseteq \gamma_D(\texttt{B-ASSIGN}_{\mathcal{D}}[\![\, p := p.f \,]\!](d))$.

Proof. Based on Lemma 3.4.1 we only have to show soundness regarding the length and reachability of p. Let d be a post state. We argue by case distinction on the length of p in the post state.

We first consider the case where the length of p is $l_{\text{post}} = [0,0],[0,0]$. The length of p in the pre state $\texttt{B-ASSIGN}_{\mathcal{D}}[\![\, p := p.f \,]\!](d)$ is $l_{\text{pre}} = f^{-1}(l_{\text{post}}) = ([1,1],[0,0])$ (cf. Equation 3.17). From the definition of the concretization function we know that the reference p points to *null* in all heap configurations in $\gamma_{\mathcal{D}}(d)$. The only heap configurations that lead to a heap configuration where p points to *null* after executing $p := p.f$ are heap configurations where $p.f$ points to *null*, i.e., the acyclic length of p is one and the cyclic length of p is zero (cf. Definition 3.2.1). These heap configurations are over-approximated by l_{pre}.

We now consider the case where the length of p is $l_{\text{post}} = ([a_l, a_u],[c_l, c_u])$ for some $a_l \geq 1$. The length of p in the pre state $\texttt{B-ASSIGN}_{\mathcal{D}}[\![\, p := p.f \,]\!](d)$ is $l_{\text{pre}} = f^{-1}(l_{\text{post}}) = ([a_l + 1, a_u + 1],[c_l, c_u])$ (cf. Equation 3.17). From the definition of the concretization

function we know that the acyclic length of p is some positive number $a \in [a_l, a_u]$ for all heap configurations in $\gamma_{\mathcal{D}}(d)$. The only heap configurations that lead to a heap configuration where the acyclic length of p is the positive number a after executing $p := p.f$ are heap configurations where the acyclic length of p is $a + 1$. All those heap configurations are over-approximated by l_{pre}.

We now consider the case where the length of p is $l_{\text{post}} = ([0,0],[c_l,c_u])$ for some $c_l \geq 1$. The length of p in the pre state $\texttt{B-ASSIGN}_{\mathcal{D}}[\![\, p := p.f \,]\!](d)$ is $l_{\text{pre}} = f^{-1}(l_{\text{post}}) = ([0,1],[c_l,c_u])$ (cf. Equation 3.17). From the definition of the concretization function we know that the cyclic length of p is some positive number $c \in [c_l, c_u]$ for all heap configurations in $\gamma_{\mathcal{D}}(d)$. This means that p points to a cycle in all heap configurations in $\gamma_{\mathcal{D}}(d)$. As illustrated in Example 3.4.6 there are two heap configurations that can lead to a heap configuration where p points to a cycle: either by staying in the cycle or by entering the cycle from "outside". Both cases are over-approximated by l_{pre}.

The final case returns $\top_L$ which is trivially sound. □

$\texttt{B-ASSIGN}_{\mathcal{D}}[\![\, p := q.f \,]\!]$ The statement $p := q.f$ gives us the information that the reference p points to the same location as $q.f$. This means that q can reach p. If this is not the case, we are in an invalid trace and no pre state exists. Similarly to the transformer for $p = q$ (cf. Equation 3.15) we can use the information on p to refine the information on q. The transformer is defined as follows:

$$\texttt{B-ASSIGN}_{\mathcal{D}}[\![\, p := q.f \,]\!](l,r) \overset{\text{def}}{=} \begin{cases} (l', r') & \text{if R} \sqsubseteq_{\mathcal{R}} r(q,p) \\ \bot_{\mathcal{D}} & \text{otherwise} \end{cases} \tag{3.18}$$

where $l' = l[p \leftarrow \top_{\mathcal{L}}, q \leftarrow f^{-1}(l(p) \sqcap_{\mathcal{L}} l(q))]$ and $r' = r[(p,t) \leftarrow \top_{\mathcal{R}}, (t,p) \leftarrow \top_{\mathcal{R}}\ \forall t \in \mathit{Ref} \setminus \{p\}]$.

Lemma 3.4.6. The backward assignment operator $\texttt{B-ASSIGN}_{\mathcal{D}}[\![\, p := q.f \,]\!]$ is a sound over-approximation of $\tau_I[\![p := q.f]\!]$, i.e., given a post state $d \in \mathcal{D}$ of the assignment $p := q.f$ we have $\{s \in \mathit{State} \mid \tau_I[\![^l p := p.f,]\!](s) \subseteq \gamma_{\mathcal{D}}(d)\} \subseteq \gamma_D(\texttt{B-ASSIGN}_{\mathcal{D}}[\![\, p := q.f \,]\!](d))$.

Proof. Analogous to the proof of Lemma 3.4.4 using a case distinction over $\text{R} \sqsubseteq_{\mathcal{R}} r(q,p)$ in the post state. □

Heap manipulating statements

We now define the transformers for statements that manipulate the heap. The transformers for these statements are much less precise since a statement such as $p.f := \ldots$ might affect another reference q due to aliasing.

To clarify, let us consider the following example:

Example 3.4.7. We consider a program with $Ref \stackrel{\text{def}}{=} \{p, q\}$ and $Field \stackrel{\text{def}}{=} \{f\}$. Let s be a state with the heap configuration shown in Figure 3.11 (a). Executing the statement $p.f := null$ on s gives us the state s' with the heap configuration shown in Figure 3.11 (b). Because p and q alias in the state s, the assignment to p also changes the length of q.

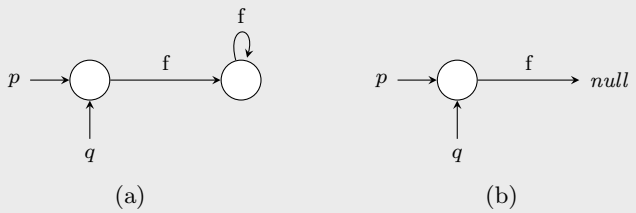


Figure 3.11: A heap configuration (a) and the resulting heap configurations (b) after the assignment $p.f := null$.

$\texttt{B-ASSIGN}_{\mathcal{D}}[\![\, p.f := \textbf{null}\,]\!]$ The statement $p.f := null$ gives us the information that $p.f$ points to *null* in the post state which means that the acyclic length of p is one and the cyclic length of p is zero. If this is not the case, we are in an invalid trace and no pre state exists. The transformer is defined as follows:

$$\texttt{B-ASSIGN}_{\mathcal{D}}[\![\, p.f := null]\!](l, r) \stackrel{\text{def}}{=} \begin{cases} \top_{\mathcal{D}} & \text{if } ([1,1],[0,0]) \sqsubseteq_{\mathcal{L}} l(p) \\ \bot_{\mathcal{D}} & \text{otherwise} \end{cases} \tag{3.19}$$

Lemma 3.4.7. The backward assignment operator $\texttt{B-ASSIGN}_{\mathcal{D}}[\![\, p.f := null]\!]$ is a sound over-approximation of $\tau_I[\![p.f := null]\!]$, i.e., given a post state $d \in \mathcal{D}$ of the assignment $p.f := null$ we have $\{s \in State \mid \tau_I[\![{}^l p.f := null]\!](s) \subseteq \gamma_{\mathcal{D}}(d))\} \subseteq \gamma_D(\texttt{B-ASSIGN}_{\mathcal{D}}[\![\, p.f := null]\!](d))$.

Proof. Let $(l, r) \in \mathcal{D}$ be a post state. We argue by case distinction whether $([1,1],[0,0]) \sqsubseteq_{\mathcal{L}} l(p)$ holds.

In the first case we assume $([1,1],[0,0]) \not\sqsubseteq_{\mathcal{L}} l(p)$. This means that the result of the transformer is $\bot_{\mathcal{D}}$. We assume, by absurd, that the inclusion does not hold. This means there is some state $s \in State$ such that $\tau_I[\![\, p := null]\!](s)$ leads to a state where the length of p is not in $([1,1],[0,0])$. This is not possible according to the semantics (cf. Section 2.3). Thus the inclusion holds.

The second case is trivially sound since $\top_{\mathcal{D}}$ is returned. □

$\texttt{B-ASSIGN}_{\mathcal{D}}[\![\, p.f := \textbf{new}\,]\!]$ The statement $p.f := new$ gives us the information that $p.f$ points to a new instance in the post state which means that the acyclic length of p is

two and the cyclic length of p is zero. If this is not the case, we are in an invalid trace and no pre state exists. The transformer is defined as follows:

$$\texttt{B-ASSIGN}_{\mathcal{D}}[\![\, p.f := new \,]\!](l, r) \stackrel{\text{def}}{=} \begin{cases} \top_{\mathcal{D}} & \text{if } ([2,2],[0,0]) \sqsubseteq_{\mathcal{L}} l(p) \\ \bot_{\mathcal{D}} & \text{otherwise} \end{cases} \tag{3.20}$$

Lemma 3.4.8. The backward assignment operator $\texttt{B-ASSIGN}_{\mathcal{D}}[\![\, p.f := new \,]\!]$ is a sound over-approximation of $\tau_I[\![p.f := new]\!]$, i.e., given a post state $d \in \mathcal{D}$ of the assignment $p.f := new$ we have $\{s \in State \mid \tau_I[\![^l p.f := new]\!](s) \subseteq \gamma_{\mathcal{D}}(d))\} \subseteq \gamma_D(\texttt{B-ASSIGN}_{\mathcal{D}}[\![\, p.f := new \,]\!](d))$.

Proof. Analogous to the proof of Lemma 3.4.7 using a case distinction over $([2,2],[0,0]) \sqsubseteq_{\mathcal{L}} l(p)$. □

$\texttt{B-ASSIGN}_{\mathcal{D}}[\![\, p.f := p \,]\!]$ The statement $p.f := p$ gives us the information that p points to a cycle with a single element in the post state which means that the acyclic length of p is zero and the cyclic length of p is one. If this is not the case, we are in an invalid trace and no pre state exists. The transformer is defined as follows:

$$\texttt{B-ASSIGN}_{\mathcal{D}}[\![\, p.f := p \,]\!](l, r) \stackrel{\text{def}}{=} \begin{cases} \top_{\mathcal{D}} & \text{if } ([0,0],[1,1]) \sqsubseteq_{\mathcal{L}} l(p) \\ \bot_{\mathcal{D}} & \text{otherwise} \end{cases} \tag{3.21}$$

Lemma 3.4.9. The backward assignment operator $\texttt{B-ASSIGN}_{\mathcal{D}}[\![\, p.f := p \,]\!]$ is a sound over-approximation of $\tau_I[\![p.f := p]\!]$, i.e., given a post state $d \in \mathcal{D}$ of the assignment $p.f := p$ we have $\{s \in State \mid \tau_I[\![\, p.f := p \,]\!](s) \subseteq \gamma_{\mathcal{D}}(d))\} \subseteq \gamma_D(\texttt{B-ASSIGN}_{\mathcal{D}}[\![\, p.f := p \,]\!](d))$.

Proof. Analogous to the proof of Lemma 3.4.7 using a case distinction over $([0,0],[1,1]) \sqsubseteq_{\mathcal{L}} l(p)$. □

$\texttt{B-ASSIGN}_{\mathcal{D}}[\![\, p.f := q \,]\!]$ The statement $p.f = q$ gives us the information that $p.f$ and q point to the same location in the post state. To facilitate the definition of the transformer we consider cases based on the fixed partitions for the length of p (cf. Section 3.4.1). The transformer is defined as follows:

$$\texttt{B-ASSIGN}_{\mathcal{D}}[\![\, p.f := q \,]\!](l, r) \stackrel{\text{def}}{=} \begin{cases} \bot_{\mathcal{D}} & \text{if } l(p) \sqsubseteq_{\mathcal{L}} ([0,0],[0,0]) \\ \bot_{\mathcal{D}} & \text{if } l(p) \sqsubseteq_{\mathcal{L}} ([1,\infty],[0,\infty]) \text{ and} \\ & (\text{R} \not\sqsubseteq_{\mathcal{R}} (p,q) \text{ or NR} \not\sqsubseteq_{\mathcal{R}} (q,p)) \\ (l', r') & \text{if } l(p) \sqsubseteq_{\mathcal{L}} ([1,\infty],[0,\infty]) \text{ and} \\ & \text{R} \sqsubseteq_{\mathcal{R}} (p,q) \text{ and NR} \sqsubseteq_{\mathcal{R}} (q,p)) \\ \top_{\mathcal{D}} & \text{if } l(p) \sqsubseteq_{\mathcal{L}} ([0,0],[1,\infty]) \\ \top_{\mathcal{D}} & \text{otherwise} \end{cases} \tag{3.22}$$

where $l' = \top_{\mathcal{L}}[p \leftarrow \top_{\mathcal{L}}, q \leftarrow ([a_l - 1, a_u - 1], [c_l, c_u])$ and $([a_l, a_u], [c_l, c_u]) \stackrel{\text{def}}{=} l(p)$ and $r' = r[(q, p) \leftarrow \text{NR}]$. The first case checks for an invalid state where p is *null* after the assignment, which is not possible. The second and third case are both used to handle the case in which the acyclic length of p is at least one. Based on the statement $p.f := q$ and the fact that the acyclic length of p is at least one we know that q points to a separate node reachable by the field f from p and that q cannot reach p, shown in Figure 3.12 (a). If this was not the case p would be a cycle after the statement $p.f := q$.

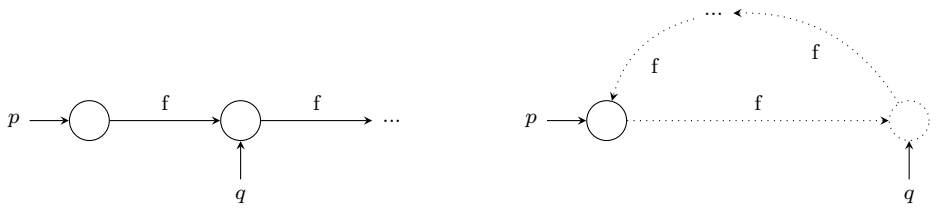


Figure 3.12: Illustration of the second and third case (a) and the fourth case (b) of the transformer $\texttt{B-ASSIGN}_{\mathcal{D}}[\![\, p.f := q \,]\!]$

The second case of $\texttt{B-ASSIGN}_{\mathcal{D}}[\![\, p.f := q \,]\!]$ checks for situations where p cannot reach q or where q can reach p, which are both invalid states. In the third case we use the information on p to refine the length of q. The fourth case represents the fixed partition where p is a cycle, shown in Figure 3.12 (b). From the statement and the fact that p is a cycle we know that q can reach p in the pre state (assuming q could not reach p in the pre state means that q could still not reach p after the statement $p.f = q$, which is absurd since p is a cycle and p can reach q). We choose to be very imprecise, i.e., returning $\top_{\mathcal{D}}$ in this case, since the abstraction is not precise enough to give useful results for the termination analysis. We will discuss this in more detail in the evaluation (cf. Example 5.2.7 and Example 5.2.8) and show how the abstraction could be improved to be able to handle this case (cf. Section 6.1).

To clarify, let us consider the following example:

Example 3.4.8. We consider a program with $\mathit{Ref} \stackrel{\text{def}}{=} \{p, q\}$ and $\mathit{Field} \stackrel{\text{def}}{=} \{f\}$. Let $d = (\{p \mapsto ([1, 5], [3, 9]), q \mapsto ([2, 5], [3, 9])\}, \{(p, q) \mapsto \top_{\mathcal{R}}, (q, p) \mapsto \top_{\mathcal{R}}\})$ be a post state. We have $[1, 5], [3, 9] \sqsubseteq_{\mathcal{L}} ([1, \infty], [0, \infty])$ and both $\text{R} \sqsubseteq_{\mathcal{R}} (p, q)$ and $\text{NR} \sqsubseteq_{\mathcal{R}} (q, p))$, which means that we are in the third case of $\texttt{B-ASSIGN}_{\mathcal{D}}[\![\, p.f := q \,]\!]$. Applying the transformer on d gives us $d' = \texttt{B-ASSIGN}_{\mathcal{D}}[\![\, p.f := q \,]\!](d) = (\{p \mapsto \top_{\mathcal{L}}, q \mapsto ([2, 4], [3, 9])\}, \{(p, q) \mapsto \top_{\mathcal{R}}, (q, p) \mapsto \text{NR}\})$.

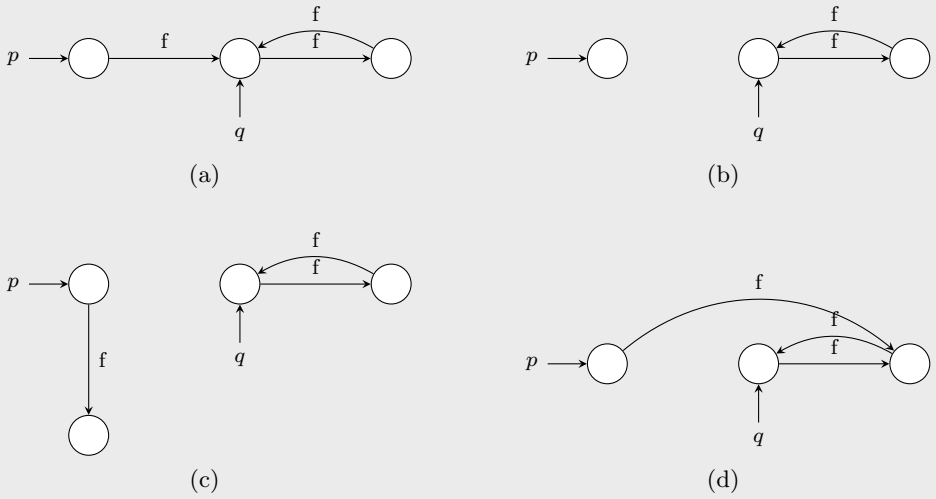


Figure 3.13: The heap configuration of a state $s \in \gamma_{\mathcal{D}}(a)$ (a) and heap configurations that lead to (a) by executing the statement $p.f = q$ (b), (c) and (d).

Let us consider a state $s \in \gamma_{\mathcal{D}}(d)$ with the heap configuration defined by Figure 3.13 (a). The reference p reaches q in one step using the field f and q is a cycle of length two. Figure 3.13 (b), (c) and (d) show possible heap configurations that lead to (a) when executing the statement $p.f = q$. All of which are over-approximated by d'. Note that all three heap configurations have two things in common: q cannot reach p and the length of q is defined by the length of p in (a), which is exactly what the third case in the transformer of $\texttt{B-ASSIGN}_{\mathcal{D}}[\![\, p.f := q \,]\!]$ does.

Lemma 3.4.10. The backward assignment operator $\texttt{B-ASSIGN}_{\mathcal{D}}[\![\, p.f := q \,]\!]$ is a sound over-approximation of $\tau_I[\![p.f := q]\!]$, i.e., given a post state $d \in \mathcal{D}$ of the assignment $p.f := q$ we have $\{s \in \mathit{State} \mid \tau_I[\![^l p.f := q \,]\!](s) \subseteq \gamma_{\mathcal{D}}(d))\} \subseteq \gamma_D(\texttt{B-ASSIGN}_{\mathcal{D}}[\![\, p.f := q \,]\!](d))$.

Proof. Let $(l, r) \in \mathcal{D}$ be a post state. We use the same case distinctions as in the transformer for $p.f = q$.

In the first case we assume $l(p) \sqsubseteq_{\mathcal{L}} ([0,0],[0,0])$. Together with the assumption that the post state is not $\bot_{\mathcal{D}}$ we have $l(p) = ([0,0],[0,0])$, i.e., p is *null* which is not possible according to the semantics of $p.f := q$ (cf. Section 2.3). This means that we are in an invalid state and there exists no pre state and therefore $\bot_{\mathcal{D}}$ is a valid over-approximation of the pre state.

In the second case we assume $l(p) \sqsubseteq_{\mathcal{L}} ([1,\infty],[0,\infty])$ and either R $\not\sqsubseteq_{\mathcal{R}} (p,q)$ or NR $\not\sqsubseteq_{\mathcal{R}} (q,p)$. The result of the transformer is $\bot_{\mathcal{D}}$ (cf. Equation 3.22). We assume, by absurd, that there is some state $s \in \mathit{State}$ such that after executing the statement $p.f := q$ is in

(l, r), the length of p is in $([1, \infty], [0, \infty]$ and either p cannot reach q or q can reach p. It is impossible that p cannot reach q after the statement $p.f = q$ by the definition of the semantics. Let us now assume that p is in $([1, \infty], [0, \infty])$ and that q can reach p. Since p can reach q and q can reach p, p and q either have to be the same element or have to belong to a cycle, both are not possible since the acyclic length of p is at least one.

In the third case we assume that $l(p) \sqsubseteq_{\mathcal{L}} ([1, \infty], [0, \infty])$, R $\sqsubseteq_{\mathcal{R}}$ (p, q) and NR $\sqsubseteq_{\mathcal{R}}$ (q, p). Let $([a_l, a_u], [c_l, c_u])$ be the length of p in (l, r) and let (l', r') be the result of the transformer (cf. Equation 3.22). We only have to show the soundness of the length and reachability information of q since everything else is $\top$, which is trivially sound. We assume, by absurd, that there exists some state $s \in \mathit{State}$ such that after executing the statement $p.f = q$ is in (l, r), the length of p is in $([1, \infty], [0, \infty])$ and p can reach q or q cannot reach p and s is not over-approximated by (l', r'). This means that either q can reach p in s or the length of q is not $([a_l - 1, a_u - 1], [c_l, c_u])$. We first assume that q can reach p in s. This means that p is a cycle after the statement $p.f = q$ which is not possible according to the assumption that p is in $([1, \infty], [0, \infty]$, since the acyclic length of p is always at least one. We now assume that the length of q is not $([a_l - 1, a_u - 1], [c_l, c_u])$. This means that after the statement $p.f = q$ the length of p cannot be $([a_l, a_u], [c_l, c_u])$, which is a contradiction.

The fourth and fifth cases are trivial since the result is $\top_{\mathcal{D}}$. □

Theorem 3.4.11. The backward assignment operator $\texttt{B-ASSIGN}_{\mathcal{D}}$ is a sound over-approximation of τ_I.

Proof (Sketch). The proof follows from the soundness of the transformer for the specific statements (cf. Lemma 3.4.2, Lemma 3.4.3, Lemma 3.4.4, Lemma 3.4.5, Lemma 3.4.6, Lemma 3.4.7, Lemma 3.4.8, Lemma 3.4.9 and Lemma 3.4.10) and the assumption that $\mathit{Num} = \emptyset$, i.e., $\mathit{Var} = \mathit{Ref}$. □

3.4.3 Filter

We now define the filter operator $\texttt{FILTER}_{\mathcal{D}} \colon \mathcal{D} \to \mathcal{D}$ to handle filter instructions by induction on the syntax of the language defined in Section 2.1. Note that in our language filter instructions do not change any state and therefore the filter instruction $\mathit{rexp}_1 = \mathit{rexp}_2$ is equivalent to $\mathit{rexp}_2 = \mathit{rexp}_1$. We will only define the transformer for one case.

$\texttt{FILTER}_{\mathcal{D}}[\![\, p = \textbf{null} \,]\!]$ The filter instruction $p = \mathit{null}$ returns only heap configurations where p is *null* which means that the acyclic and cyclic length of p are zero. This gives us the following transformer:

$$\texttt{FILTER}_{\mathcal{D}}[\![\, p = \mathit{null} \,]\!](l, r) \stackrel{\text{def}}{=} (l', r) \tag{3.23}$$

where $l' = l[p \leftarrow (l(p) \sqcap_{\mathcal{L}} ([0, 0], [0, 0])]$.

Lemma 3.4.12. The operator $\texttt{FILTER}_{\mathcal{D}}[\![\, p = \mathit{null} \,]\!]$ is sound, i.e., given an element $d \in \mathcal{D}$ we have $\{s \in \gamma_{\mathcal{D}}(d) \mid \texttt{true} \in [\![\, p = \mathit{null} \,]\!]s\} \subseteq \gamma_{\mathcal{D}}(\texttt{FILTER}_{\mathcal{D}}[\![\, p = \mathit{null} \,]\!]d)$.

Proof. Let $(l, r) \in \mathcal{D}$ be a post state. We assume, by absurd, that there exists some state $s \in \gamma_{\mathcal{D}}(d)$ such that $\texttt{true} \in [\![\, p = \mathit{null} \,]\!]s$ and $s \notin \gamma_{\mathcal{D}}(\texttt{FILTER}_{\mathcal{D}}[\![\, p = \mathit{null} \,]\!](l, r))$. By the semantics of references expressions (cf. Figure 2.3) we know that s is a state in $\gamma_{\mathcal{D}}(d)$ and that the acyclic and cyclic length of p are zero in s. This is absurd, since $\gamma_{\mathcal{D}}(\texttt{FILTER}_{\mathcal{D}}[\![\, p = \mathit{null} \,]\!]d)$ contains all states of $\gamma_{\mathcal{D}}(d)$ where the acyclic and cyclic length of p are zero (cf. Equation 3.23). □

$\texttt{FILTER}_{\mathcal{D}}[\![\, p \neq \textbf{null} \,]\!]$ The filter $p \neq \mathit{null}$ returns only heap configurations where p is not *null* which means that the acyclic and cyclic length of p are not zero. To improve the precision of the transformer we make use of the fixed partitions (cf. Section 3.4:

$$\texttt{FILTER}_{\mathcal{D}}[\![\, p \neq \mathit{null} \,]\!](l, r) \stackrel{\text{def}}{=} \begin{cases} \bot_{\mathcal{D}} & \text{if } l(p) \sqsubseteq_{\mathcal{L}} ([0,0],[0,0]) \\ (l, r) & \text{otherwise} \end{cases} \tag{3.24}$$

Lemma 3.4.13. The operator $\texttt{FILTER}_{\mathcal{D}}[\![\, p \neq \mathit{null} \,]\!]$ is sound, i.e., given an element $d \in \mathcal{D}$ we have $\{s \in \gamma_{\mathcal{D}}(d) \mid \mathit{true} \in [\![\, p \neq \mathit{null} \,]\!]s\} \subseteq \gamma_{\mathcal{D}}(\texttt{FILTER}_{\mathcal{D}}[\![\, p \neq \mathit{null} \,]\!]d)$.

Proof. Let $(l, r) \in \mathcal{D}$ be a post state. We argue by case distinction whether $l(p) \sqsubseteq_{\mathcal{L}} ([0,0],[0,0])$. In the first case we assume that $l(p) \sqsubseteq_{\mathcal{L}} ([0,0],[0,0])$. This means that p is *null* in all states $s \in \gamma_{\mathcal{D}}(l, r)$ and therefore $\texttt{FILTER}_{\mathcal{D}}[\![\, p \neq \mathit{null} \,]\!]$ is the empty set. Thus $\bot_{\mathcal{D}}$ is a valid over-approximation.

The other case is sound since we return (l, r) and $\{s \in \gamma_{\mathcal{D}}(l, r) \mid \mathit{true} \in [\![\, p \neq \mathit{null} \,]\!]s\} \subseteq \gamma_{\mathcal{D}}(l, r)$ trivially holds.

□

$\texttt{FILTER}_{\mathcal{D}}[\![\, p = p.f \,]\!]$ The filter instruction $p = p.f$ returns only heap configurations where p points to itself with the field f which means that the acyclic length of p is zero and the cyclic length of p is one. This gives us the following transformer:

$$\texttt{FILTER}_{\mathcal{D}}[\![\, p = p.f \,]\!](l, r) \stackrel{\text{def}}{=} (l', r) \tag{3.25}$$

where $l' = l[p \leftarrow (l(p) \sqcap_{\mathcal{L}} ([0,0],[1,1]))]$.

Lemma 3.4.14. The operator $\texttt{FILTER}_{\mathcal{D}}[\![\, p = p.f \,]\!]$ is sound, i.e., given an element $d \in \mathcal{D}$ we have $\{s \in \gamma_{\mathcal{D}}(d) \mid \mathit{true} \in [\![\, p = p.f \,]\!]s\} \subseteq \gamma_{\mathcal{D}}(\texttt{FILTER}_{\mathcal{D}}[\![\, p = p.f \,]\!]d)$.

Proof. Analogous to the proof of Lemma 3.4.12. □

$\texttt{FILTER}_{\mathcal{D}}[\![\, p \neq p.f \,]\!]$ The filter instruction $p = p.f$ returns only heap configurations where p points to itself with the field f which means that p cannot have an acyclic and cyclic length of zero and one. As in the transformer for $p \neq \mathit{null}$ we make use of the fixed partitions (cf. Section 3.4) to improve the precision of the transformer:

$$\texttt{FILTER}_{\mathcal{D}}[\![\, p \neq p.f \,]\!](l, r) \stackrel{\text{def}}{=} \begin{cases} \bot_{\mathcal{D}} & \text{if } l(p) \sqsubseteq_{\mathcal{L}} ([0,0],[1,1]) \\ (l, r) & \text{otherwise} \end{cases} \tag{3.26}$$

Lemma 3.4.15. The operator $\texttt{FILTER}_{\mathcal{D}}[\![\, p \neq p.f \,]\!]$ is sound, i.e., given an element $d \in \mathcal{D}$ we have $\{s \in \gamma_{\mathcal{D}}(d) \mid \mathit{true} \in [\![\, p \neq p.f \,]\!]s\} \subseteq \gamma_{\mathcal{D}}(\texttt{FILTER}_{\mathcal{D}}[\![\, p \neq p.f \,]\!]d)$.

Proof. Analogous to the proof of Lemma 3.4.13. □

$\texttt{FILTER}_{\mathcal{D}}[\![\, \textbf{rexp}_1 \texttt{ and } \textbf{rexp}_2 \,]\!]$ The filter instruction rexp_1 `and` rexp_2 only returns states where both rexp_1 and rexp_2 are satisfied. This gives us the following transformer:

$$\texttt{FILTER}_{\mathcal{D}}[\![\, \mathit{rexp}_1 \texttt{ and } \mathit{rexp}_2 \,]\!]d \stackrel{\text{def}}{=} d_1 \sqcap_{\mathcal{D}} d_2 \tag{3.27}$$

where $d_1 = \texttt{FILTER}_{\mathcal{D}}[\![\, \mathit{rexp}_1 \,]\!]d$, and $d_2 = \texttt{FILTER}_{\mathcal{D}}[\![\, \mathit{rexp}_2 \,]\!]d$.

Lemma 3.4.16. The operator $\texttt{FILTER}_{\mathcal{D}}[\![\, \mathit{rexp}_1 \texttt{ and } \mathit{rexp}_2 \,]\!]$ is sound, i.e., given an element $d \in \mathcal{D}$ we have $\{s \in \gamma_{\mathcal{D}}(d) \mid \mathit{true} \in [\![\, \mathit{rexp}_1 \texttt{ and } \mathit{rexp}_2 \,]\!]s\} \subseteq \gamma_{\mathcal{D}}(\texttt{FILTER}_{\mathcal{D}}[\![\, \mathit{rexp}_1 \texttt{ and } \mathit{rexp}_2 \,]\!]d)$.

Proof (Sketch). Follows from the soundness of the meet operator (cf. Lemma 3.3.2). □

$\texttt{FILTER}_{\mathcal{D}}[\![\, \textbf{rexp}_1 \texttt{ or } \textbf{rexp}_2 \,]\!]$ The filter instruction rexp_1 `or` rexp_2 only returns states where either rexp_1 or rexp_2 is satisfied. This gives us the following transformer:

$$\texttt{FILTER}_{\mathcal{D}}[\![\, \mathit{rexp}_1 \texttt{ or } \mathit{rexp}_2 \,]\!]d \stackrel{\text{def}}{=} d_1 \sqcup_{\mathcal{D}} d_2 \tag{3.28}$$

where $d_1 = \texttt{FILTER}_{\mathcal{D}}[\![\, \mathit{rexp}_1 \,]\!]d$, and $d_2 = \texttt{FILTER}_{\mathcal{D}}[\![\, \mathit{rexp}_2 \,]\!]d$.

Lemma 3.4.17. The operator $\texttt{FILTER}_{\mathcal{D}}[\![\, \mathit{rexp}_1 \texttt{ or } \mathit{rexp}_2 \,]\!]$ is sound, i.e., given an element $d \in \mathcal{D}$ we have $\{s \in \gamma_{\mathcal{D}}(d) \mid \mathit{true} \in [\![\, \mathit{rexp}_1 \texttt{ or } \mathit{rexp}_2 \,]\!]s\} \subseteq \gamma_{\mathcal{D}}(\texttt{FILTER}_{\mathcal{D}}[\![\, \mathit{rexp}_1 \texttt{ or } \mathit{rexp}_2 \,]\!]d)$.

Proof (Sketch). The proof follows from the soundness of the join operator (cf. Lemma 3.3.1). □

All other filter instructions like $p = q$, $p \neq q$, $p.f = q$, etc. simply return the passed abstract element which is trivially sound. It is not possible to be more precise since we cannot represent the fact that two references *must* point to the same location in the heap using the heap abstract domain.

Theorem 3.4.18. The operator $\texttt{FILTER}_{\mathcal{D}}$ is sound, i.e., $\{s \in \gamma_{\mathcal{D}}(d) \mid \mathit{true} \in [\![\mathit{bexp}]\!] s\} \subseteq \gamma_{\mathcal{D}}(\texttt{FILTER}_{D}[\![\mathit{bexp}]\!](d))\}$.

Proof (Sketch). The proof follows from the soundness of the specific transformers (cf. Lemma 3.4.12, Lemma 3.4.13, Lemma 3.4.14, Lemma 3.4.15, Lemma 3.4.16 and Lemma 3.4.17) and the assumption that $\mathit{Num} = \emptyset$, i.e., $\mathit{Var} = \mathit{Ref}$. □

4 Piecewise Defined Ranking Functions

In this chapter, we present an abstract domain for effectively proving program termination by abstract interpretation of the termination semantics presented in Section 2.4. The domain is based on the decision tree abstract domain introduced in [Urb15b, Chapter 5] and is used to automatically synthesize piecewise-defined ranking functions and infer sufficient preconditions for program termination.

In Section 4.1, we briefly repeat the relevant definitions of the original decision trees abstract domain. Based on this, in Section 4.2, we define a new decision trees abstract domain which uses the heap abstraction introduced in Chapter 3. To avoid confusion from now on we will call the original decision trees abstract domain *value decision trees* and the new decision trees abstract domain *heap decision trees*. In Section 4.3 we informally present the *combined decision trees* domain, which is a combination of the value and heap decision trees.

4.1 Value Decision Trees

In this section, we briefly repeat the definition of the value decision trees, for a more thorough discussion we refer to [Urb15b, Chapter 5]. In the following $\mathcal{X}$ denotes the set of program variables. Note that in the definition of the value decision trees there was no notion of heap or references.

4.1.1 Domain

The elements of the value decision trees are *piecewise-defined* partial functions. These piecewise-defined partial functions are represented by decision trees, where the decision nodes are labeled with linear constraints and the leaf nodes are labeled with linear functions of the program variables.

To clarify, let us consider the following example:

Example 4.1.1. Figure 4.1 shows an example decision tree.

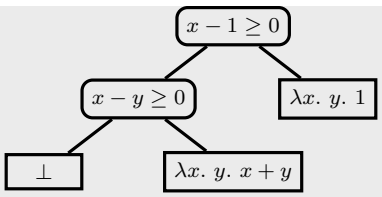


Figure 4.1: An example decision tree.

The linear constraints in the decision nodes (represented by nodes with rounded corners) are satisfied by their left subtree, while their right subtree satisfies their negation. For example in the left subtree of the root node the constraint $x - 1 \geq 0$ holds, while in the right subtree of the root node the constraint $x - 1 < 0$ holds. The leaf nodes represent partial functions whose domain is determined by the constraints satisfied along the path to the leaf node. For example the leaf node $\lambda x.\ y.\ x + y$ represents a partial function which is $x + y$ if $x \geq 1$ and $x < y$, and undefined otherwise. The leaf with value $\bot$ represents an undefined piece of the function.

The domain is parametrized by a numerical abstract domain underlying the linear constraints and by an auxiliary abstract domain for the leaf nodes.

Linear Constraints Auxiliary Abstract Domain The linear constraints auxiliary abstract domain is used to represent elements of a numerical domain as linear constraints. Elements of the *interval abstract domain* [CC76], the *octagons abstract domain* [Min06] and the *polyhedra abstract domain* [CH78] can be represented by the following canonical representation of linear constraints:

$$\mathcal{C} \stackrel{\text{def}}{=} \left\{ c_1 X_1 + \ldots + c_k X_k + c_{k+1} \geq 0 \;\middle|\; \begin{array}{c} X_1, \ldots, X_k \in \mathcal{X}, \\ c_1, \ldots, c_k, c_{k+1} \in \mathbb{Z} \\ \gcd(|c_1|, \ldots, |c_k|, |c_{k+1}|) = 1 \end{array} \right\} \tag{4.1}$$

A *total order* $<_c$ on $\mathcal{C}$ as well as the *negation* $\neg c$ of a linear constraint $c \in \mathcal{C}$ is defined. The order is used to define a canonical representation of the decision trees where a linear constraint c and its negation $\neg c$ are forbidden to appear simultaneously in a decision tree. Between the constraints c and $\neg c$ only the largest constraint with respect to $<_{\mathcal{C}}$ is kept.

Functions Auxiliary Abstract Domain The functions auxiliary abstract domain is dedicated to the manipulation of the leaf nodes of the value decision trees with respect to the set of linear constraints satisfied along the paths from the root of the decision tree. The elements of the domain are natural-valued functions of the program variables.

$$\mathcal{F} \stackrel{\text{def}}{=} \{\bot_{\mathcal{F}}\} \cup \left(\mathbb{Z}^{|\mathcal{X}|} \to \mathbb{N}\right) \cup \{\top_{\mathcal{F}}\}. \tag{4.2}$$

The functions auxiliary abstract domain defines the *computational order* $\sqsubseteq_{\mathcal{F}}$ and the *approximation order* $\preccurlyeq_{\mathcal{F}}$. Intuitively, the computational order $\sqsubseteq_{\mathcal{F}}$ abstracts the computational order $\sqsubseteq$ defined in Equation 2.2 and is used to the define the fixpoint of loops (cf. Figure 2.7) and the approximation order $\preccurlyeq_{\mathcal{F}}$ abstracts the approximation order $\preccurlyeq$ defined in Equation 2.4 and is used to relate elements of the value decision trees. In the approximation order $\bot_{\mathcal{F}}$ and $\top_{\mathcal{F}}$ are incomparable since both are used to represent undefined ranking functions.

Decision Trees Abstract Domain The elements of the value decision trees belong to the following set:

$$\mathcal{T} \stackrel{\text{def}}{=} \{\text{LEAF} : r \mid r \in \mathcal{F}\} \cup \{\text{NODE}\{c\} : t_1; t_2 \mid c \in \mathcal{C}, t_1, t_2 \in \mathcal{T}\} \tag{4.3}$$

where $\mathcal{F}$ is defined in Equation 4.2 and $\mathcal{C}$ is defined in Equation 4.1. A *decision tree* $t \in \mathcal{T}$ is either a *leaf node* $\text{LEAF} : r$, with r an element of $\mathcal{F}$ (in the following denoted by $t.r$), or a *decision node* $\text{NODE}\{c\} : t_1; t_2$, such that c is a linear constraint in $\mathcal{C}$ (in the following denoted by $t.c$) and the left subtree t_1 and the right subtree t_2 (in the following denoted by $t.l$ and $t.r$, respectively) belong to $\mathcal{T}$. In addition, given a decision tree $\text{NODE}\{c\} : t_1; t_2$, we impose that the linear constraint $c \in \mathcal{C}$ is always the largest constraint, with respect to $< \mathcal{C}$, appearing in the tree.

For the definition of the operators the following set is used:

$$\mathcal{T}_{\text{NIL}} \stackrel{\text{def}}{=} \{\text{NIL}\} \cup \{\text{LEAF} : r \mid r \in \mathcal{F}\} \cup \{\text{NODE}\{c\} : t_1; t_2 \mid c \in \mathcal{C}, t_1, t_2 \in \mathcal{T}_{\text{NIL}}\} \tag{4.4}$$

where the special element NIL represents the absence of information regarding some partition of the domain of the ranking function.

4.1.2 Order

In [Urb15b, Section 5.2.2] an order is defined for the value decision trees. The approximation order $\preccurlyeq_{\mathcal{T}}$ is an ordering based on the function approximation order $\preccurlyeq_{\mathcal{F}}$ and the computational order $\sqsubseteq_{\mathcal{T}}$ is an ordering based on the function computational order $\sqsubseteq_{\mathcal{F}}$. Given two decision trees $t_1, t_2 \in \mathcal{T}_{\text{NIL}}$ the approximation order (computational order) first unifies both trees and then compares the leaf nodes using the approximation order (computational order) using the linear constraints accumulated on the path to the leaf node. The tree t_1 is smaller than the tree t_2 if all of its leaf nodes are smaller. We refer to [Urb15b, Section 5.2.2] for a more thorough discussion.

4.1.3 Join

In [Urb15b, Section 5.2.2] the join operator is defined for the value decision trees. The join of two decision trees returns a decision tree representing a ranking function defined over the union of their partitions. There are two ways to join decision trees: the

approximation join $\curlyvee_{\mathcal{T}}$ is the least upper bound for the approximation order and the *computational join* $\sqcup_{\mathcal{T}}$ is the least upper bound for the computational order. We refer to [Urb15b, Section 5.2.2] for a more thorough discussion.

4.1.4 Step

In [Urb15b, Section 5.2.3] the step operator, in the following denoted by STEP, is defined for the value decision trees. Given a decision tree, it increases the constant of each ranking function to take into account that one more execution step is needed before termination. We refer to [Urb15b, Section 5.2.3] for a more thorough discussion.

4.1.5 Widening

In [Urb15b, Section 5.2.4] a widening is defined for the value decision trees. The widening is allowed more freedom than the other operators, in the sense that it is temporarily allowed to *under-approximate* the value of the termination semantics τ_{T} (cf. Section 2.4) or *over-approximate* its domain of definition, or both. This is necessary to extrapolate the ranking function for over the states on which it is not yet defined. The only requirement is that, when the iteration sequence with widening is stable for the computational order, its limit is a sound abstraction of the termination semantics with respect to the approximation order.

Intuitively the widening extrapolates the ranking function and then checks in the following iterations for cases where the function is not valid. If such a case is found, the respective partitions are marked so that they do not get extrapolated again. The analysis continues iterating until all these discrepancies are solved. We refer to [Urb15b, Section 5.2.4] for a more thorough discussion.

4.2 Heap Decision Trees

In this section, we show how we can extend the value decision trees to use our heap abstract domain of Chapter 3 to prove termination of heap manipulating programs. To facilitate the definition of the transformer, we assume in the following that $\mathit{Num} = \emptyset$, i.e., $\mathit{Var} = \mathit{Ref}$. This means that programs have no variables of type integer and we only have to define the transformers for statements over references. In the following $\mathit{Ref} = \{p_1, \ldots, p_k\}$ denotes the set of references.

4.2.1 Domain

Our heap decision trees are built on the heap abstract domain of Chapter 3 and a custom auxiliary functions domain which will be defined in the following section.

Linear Constraints Auxiliary Abstract Domain We first show how we can use the heap abstract domain as the underlying abstraction for the linear constraints auxiliary abstract domain of the value decision trees. To this end, we represent the elements of the heap abstract domain of Chapter 3 as sets (i.e., conjunctions) of linear constraints of the form:

$$\begin{aligned} \pm a_p \geq c \quad & \text{(bound on acyclic length)} \\ \pm c_p \geq c \quad & \text{(bound on cyclic length)} \\ \pm r_{p,q} \geq 1 \quad & \text{(bound on reachabilty from p to q)} \end{aligned}$$

where $p, q \in \mathit{Ref}$ and $c \in \mathbb{Z}$. We use symbolic values to represent the acyclic length a_p, the cyclic length c_p and the reachability $r_{p,q}$ of references $p, q \in \mathit{Ref}$. Since the reachability can only have two values (reachable and not reachable) we use the constraint $r_{p,q} \geq 1$ to denote that p can reach q and $-r_{p,q} \geq -1$ to denote that p can not reach q. In the following $\mathcal{C}_L \stackrel{\text{def}}{=} \{\pm a_p \geq c \mid p \in \mathit{Ref}\} \cup \{\pm c_p \geq c \mid p \in \mathit{Ref}\}$ denotes the set of all length constraints, $\mathcal{C}_R \stackrel{\text{def}}{=} \{\pm r_{p,q} \geq c \mid p, q \in \mathit{Ref}\}$ denotes the set of all reachability constraints and $\mathcal{C}_H \stackrel{\text{def}}{=} \mathcal{C}_L \cup \mathcal{C}_R$ denotes the set of all heap constraints. Note that we use the same canonical representation $\mathcal{C}$ (cf. Equation 4.1) for the heap constraints.

We can easily formalize a correspondence between the elements of the heap abstract domain of Chapter 3 and the set $\mathcal{C}$ of linear constraints as a Galois connection $\langle \mathcal{P}(\mathcal{C}), \sqsubseteq_{\mathcal{D}} \rangle \underset{\alpha_{\mathcal{C}}}{\overset{\gamma_{\mathcal{C}}}{\leftrightarrows}} \langle \mathcal{D}, \sqsubseteq_{\mathcal{D}} \rangle$, where the abstraction function $\alpha_{\mathcal{C}} : \mathcal{P}(\mathcal{C}) \to \mathcal{D}$ maps a set of heap constraints to heap abstract element and $\gamma_{\mathcal{C}} : \mathcal{D} \to \mathcal{P}(\mathcal{C})$ maps a heap abstract element to a set of heap constraints. In particular, we define $\gamma_{\mathcal{C}}(\top_{\mathcal{D}}) \stackrel{\text{def}}{=} \emptyset$ and $\gamma_{\mathcal{C}}(\bot_{\mathcal{D}}) \stackrel{\text{def}}{=} \{\bot_{\mathcal{C}}\}$, where $\bot_{\mathcal{C}}$ represents the unsatisfiable linear constraint $-1 \geq 0$. Note that the acyclic and cyclic length are natural numbers (cf. Section 3.2) and therefore having no constraint for their lower bound means that they are zero.

To clarify, let us consider the following example:

Example 4.2.1. We consider a program with $\mathit{Ref} \stackrel{\text{def}}{=} \{p, q\}$ and $\mathit{Field} \stackrel{\text{def}}{=} \{f\}$. Let $d = (\{p \mapsto ([1,5],[3,9]), q \mapsto ([0,1],[2,\infty])\}, \{(p,q) \mapsto \top_{\mathcal{R}}, (q,p) \mapsto \mathrm{R}\}) \in \mathcal{D}$ be an element of the abstract heap domain.

We have $\gamma_{\mathcal{C}}(d) = \{a_p \geq 1, -a_p \geq -5, c_p \geq 3, -c_p \geq -9\} \cup \{a_q \geq 0, -a_q \geq -1, c_q \geq 2\} \cup \{r_{p,q} \geq 1\}$ where $\{a_p \geq 1, -a_p \geq -5, c_p \geq 3, -c_p \geq -9\}$ are the constraints for the length of p, $\{a_q \geq 0, -a_q \geq -1, c_q \geq 2\}$ are the constraints for the length of q and $\{r_{p,q} \geq 1\}$ are the reachability constraints. Note that we do not have constraints for the upper bound of the cyclic length of q and for the reachability of p to q since the upper bound of the cyclic length of q is ∞ and the reachability of p to q is $\top_{\mathcal{R}}$ in d.

Functions Auxiliary Abstract Domain We now define the functions auxiliary abstract domain. In contrast to the functions auxiliary abstract domain of the value decision

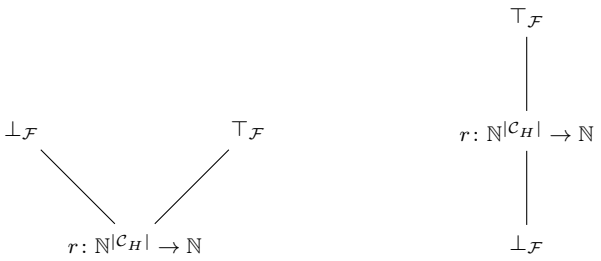


Figure 4.2: Hasse diagrams defining the approximation order $\preccurlyeq_{\mathcal{F}}[D]$ (a) and the computational order $\sqsubseteq_{\mathcal{F}}[D]$ (b) of the functions abstract domain.

tree it is not parametrized by a numerical domain but instead builds on the heap abstract domain $\mathcal{D}$. The elements of the functions auxiliary abstract domain belong to the following set:

$$\begin{aligned}\mathcal{F} \stackrel{\text{def}}{=} \{\bot_{\mathcal{F}}, \top_{\mathcal{F}}\} \cup \{r : \mathbb{N}^{|\mathcal{C}_L|} \to \mathbb{N} \mid r(a_{p_1}, c_{p_1}, \ldots, a_{p_k}, c_{p_k}) \\ = c_1 a_{p_1} + c_2 c_{p_1}, \ldots, c_{2k} a_{p_k} + c_{2k+1} c_{p_k} c_{2k+2}\}\end{aligned} \tag{4.5}$$

which consists of the natural valued affine functions of the symbolic variables for the acyclic and the cyclic lengths of the references.

We now define the *computational order* $\sqsubseteq_{\mathcal{F}}$ and the *approximation order* $\preccurlyeq_{\mathcal{F}}$ for the heap decision trees analogously to the original definition. For defined leaf nodes the orders are identical and are defined as follows:

$$\begin{aligned}r_1 \preccurlyeq_{\mathcal{F}} [D]\ r_2 \iff \forall s \in \gamma_{\mathcal{D}}(D) : r_1(a(p_1, f, s), c(p_1, f, s) \ldots, a(p_k, f, s), c(p_k, f, s)) \\ \leq r_2(a(p_1, f, s), c(p_1, f, s) \ldots, a(p_k, f, s), c(p_k, f, s))\end{aligned} \tag{4.6}$$

$$\begin{aligned}r_1 \sqsubseteq_{\mathcal{F}} [D]\ r_2 \iff \forall s \in \gamma_{\mathcal{D}}(D) : r_1(a(p_1, f, s), c(p_1, f, s) \ldots, a(p_k, f, s), c(p_k, f, s)) \\ \leq r_2(a(p_1, f, s), c(p_1, f, s) \ldots, a(p_k, f, s), c(p_k, f, s))\end{aligned} \tag{4.7}$$

where $D \in \mathcal{D}$ is an element of the heap abstract domain representing the linear constraints satisfied along the path to the compared leaf nodes and the functions a and c return the acyclic and cyclic length of p using the field f in the state s (cf. Definition 3.2.1). When one or both leaf nodes are undefined, the approximation order and computational order are defined by the Hasse diagrams in Figure 4.2 (a) and 4.2 (b) respectively.

As in the original definition a leaf node, together with its path from the root of the decision tree, represents a (piece of a) partial natural-valued functions of program states. We define the following concretization-based abstraction:

$$\langle \mathit{State} \rightharpoonup \mathbb{O}, \preccurlyeq \rangle \xleftarrow{\gamma_{\mathcal{F}}[D]} \langle \mathcal{F}, \preccurlyeq_{\mathcal{F}} \rangle$$

where $D \in \mathcal{D}$ represents the path to the leaf node. The concretization function $\gamma_{\mathcal{F}} \colon \mathcal{D} \to \mathcal{F} \to (\mathit{State} \rightharpoonup \mathbb{O})$ is defined as follows:

$$\begin{aligned} \gamma_{\mathcal{F}}[D]\bot_{\mathcal{F}} &\stackrel{\text{def}}{=} \dot{\emptyset} \\ \gamma_{\mathcal{F}}[D]r &\stackrel{\text{def}}{=} \lambda s \in \gamma_{\mathcal{D}}(D) : r(a(p_1, f, s), c(p_1, f, s) \ldots, a(p_k, f, s), c(p_k, f, s))) \\ \gamma_{\mathcal{F}}[D]\top_{\mathcal{F}} &\stackrel{\text{def}}{=} \dot{\emptyset} \end{aligned} \tag{4.8}$$

which means that for a defined leaf node we evaluate the ranking function r by substituting the symbolic variables for the length using the functions a and c defined in Definition 3.2.1. Note that both $\bot_{\mathcal{F}}$ and $\top_{\mathcal{F}}$ represent the totally undefined function and are used to represent parts of the ranking function that might not terminate.

Decision Trees Abstract Domain We now define the heap decision trees abstract domain. The domain uses the heap abstract domain $\mathcal{D}$ (cf. Chapter 3) as the underlying abstraction for the constraints and the functions auxiliary abstract domain $\mathcal{F}$ for the leafs. Analogously to the value decision trees, the elements of the heap decision trees belong to the following set:

$$\mathcal{T} \stackrel{\text{def}}{=} \{\text{LEAF} : r \mid r \in \mathcal{F}\} \cup \{\text{NODE}\{c\} \colon t_1; t_2 \mid c \in \mathcal{C}, t_1, t_2 \in \mathcal{T}\} \tag{4.9}$$

where $\mathcal{F}$ is defined in Equation 4.5 and $\mathcal{C}$ is defined in Equation 4.1.

The concretization function $\gamma_{\mathcal{T}} \colon \mathcal{D} \to \mathcal{T} \to (\mathit{State} \rightharpoonup \mathbb{O})$ is defined analogously to the original definition:

$$\gamma_{\mathcal{T}} t \stackrel{\text{def}}{=} \bar{\gamma}_{\mathcal{T}}[\emptyset]t \tag{4.10}$$

where $\bar{\gamma}_{\mathcal{T}} \colon \mathcal{D} \to \mathcal{T} \to (\mathit{State} \rightharpoonup \mathbb{O})$ accumulates into a set $C \in \mathcal{P}(\mathcal{C})$ (initially empty) the linear constraints satisfied along the paths of the decision tree up to a leaf node, where the concretization function returns a partially defined function:

$$\begin{aligned} \bar{\gamma}_{\mathcal{T}}[C]\text{LEAF} : r &\stackrel{\text{def}}{=} \gamma_{\mathcal{F}}[\alpha_{\mathcal{C}}(C)]r \\ \bar{\gamma}_{\mathcal{T}}[C]\text{NODE}\{c\} : t_1; t_2 &\stackrel{\text{def}}{=} \bar{\gamma}_{\mathcal{T}}[C \cup \{c\}]t_1 \dot{\cup} \bar{\gamma}_{\mathcal{T}}[C \cup \{\neg c\}]t_2 \end{aligned}$$

For the definition of the operators we will use the same set as in Equation 4.4:

$$\mathcal{T}_{\text{NIL}} \stackrel{\text{def}}{=} \{\text{NIL}\} \cup \{\text{LEAF} : r \mid r \in \mathcal{F}\} \cup \{\text{NODE}\{c\} : t_1; t_2 \mid c \in \mathcal{C}, t_1, t_2 \in \mathcal{T}_{\text{NIL}}\} \tag{4.11}$$

We have intentionally defined the heap decision trees very similar to the value decision trees to be able to reuse the operators for the order (cf. Section 4.1.2), the join (cf. Section 4.1.3), step (cf. Section 4.1.4) and the widening (cf. Section 4.1.5).

4.2.2 Fixed Partitions

We make an important assumption for our heap decision trees: for each reference $p \in Ref$ we assume that we always have fixed partitions for *null* (i.e., $a_p = 0$, $c_p = 0$), for only cycles (i.e., $a_p = 0$, $c_p \geq 1$) and for the rest. To clarify, let us consider the following example:

Example 4.2.2. We consider a program with $Ref = \{p\}$ and a decision tree consisting only of a leaf node with the ranking function $r \in \mathcal{F}$ shown in 4.3 (a). Figure 4.3 (b) shows an equivalent tree which satisfies our assumptions.

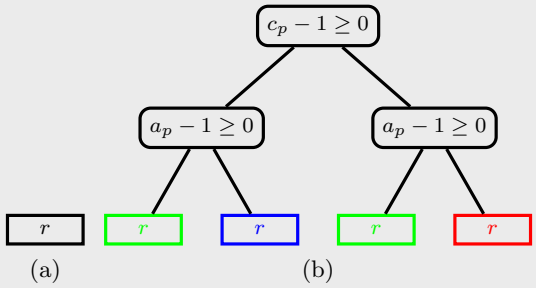


Figure 4.3: A decision tree consisting of a single leaf node with the ranking function r (a) and an equivalent decision tree satisfying the assumptions on partitions.

The partition for *null* is represented by the red leaf node (satisfying the constraints $c_p < 1$ and $a_p < 1$), the partition for only cycles is represented by the blue leaf node (satisfying the constraints $c_p \geq 1$ and $a_p < 1$) and the rest is represented by the green leaf nodes.

This gives us the fixed partitions discussed in Section 3.4.1 which allows us to use the more precise transformers of the heap abstract domain.

4.2.3 Collect

We now define the auxiliary algorithm COLLECT which, given a decision tree, collects all partitions together with their ranking functions. The algorithm is later used to define the backward assignment and filter operators.

The algorithm is defined in Algorithm 1. The main function COLLECT, given a heap decision tree $t \in \mathcal{T}_{\text{NIL}}$, calls the auxiliary function COLLECT-AUX. The function COLLECT-AUX accumulates into a set $C \in \mathcal{P}(\mathcal{C})$ (initially empty, cf. Line 10) the heap constraints encountered along the paths of the decision trees. Encountering a NIL node (cf. Line 2) the empty set is returned, for a defined leaf node a pair of its ranking function and the partition represented by the accumulated constraints in C is returned (cf. Line 3)

and for a decision node the union of the collected pairs of the left and right subtree is returned.

Algorithm 1 Collect

```
1: function COLLECT-AUX(t, C)                                  ▷ t ∈ T_NIL, C ∈ P(C)
2:     if ISNIL(t) then return ∅
3:     else if ISLEAF(t) then return (γ_C(C), t.r)
4:     else if ISNODE(t) then
5:         l ← COLLECT-AUX(t.l, C ∪ {t.c})
6:         r ← COLLECT-AUX(t.r, C ∪ {¬t.c})
7:         return l ∪ r
8:
9: function COLLECT(t)                                         ▷ t ∈ T
10:    return COLLECT-AUX(t, ∅)
```

To clarify, let us consider the following example:

Example 4.2.3. Let us consider the heap decision tree $t \in \mathcal{T}_{\text{NIL}}$ shown in Figure 4.4.

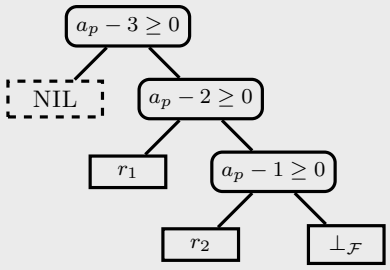


Figure 4.4: Decision tree used in Example 4.2.3.

Tree collect of t returns the set $\{(d_1, r_1), (d_2, r_2), (d_3, \bot_{\mathcal{F}})\}$ where

$$d_1 = \alpha_{\mathcal{C}}(\{a_p - 3 < 0, a_p - 2 \geq 0\}) = (\{p \mapsto ([2,2],[0,\infty])\}, \emptyset)$$
$$d_2 = \alpha_{\mathcal{C}}(\{a_p - 3 < 0, a_p - 2 < 0, a_p - 1 \geq 0)\} = (\{p \mapsto ([1,1],[0,\infty])\}, \emptyset)$$
$$d_3 = \alpha_{\mathcal{C}}(\{a_p - 3 < 0, a_p - 2 < 0, a_p - 1 < 0)\} = (\{p \mapsto ([0,0],[0,\infty])\}, \emptyset)$$

There is no pair for the NIL leaf since NIL leaf nodes are ignored in the algorithm (cf. Line 2 in Algorithm 1). Note that since the length is a natural number the constraint $a_p - 1 < 0$ implies that the acyclic length of a is equal to zero.

4.2.4 Make-Tree

We now define the auxiliary algorithm MAKE-TREE which is used to create a decision tree from a set of partition and ranking function pairs. The algorithm is later used to define the backward assignment and filter. Note that MAKE-TREE is the inverse of COLLECT, i.e., given a heap decision tree $t \in \mathcal{T}_{\text{NIL}}$ we have $t = \text{MAKE-TREE}(\text{COLLECT}(t))$.

The algorithm is defined in Algorithm 2. The main function MAKE-TREE, given a set of partition and ranking function pairs $D \in \mathcal{P}(\mathcal{D} \times \mathcal{F})$, iterates over all pairs and for each pair (d, r) first creates a tree t' only containing the partition d with the ranking function r using the function BUILD and then merges the current tree t (initialized to the empty tree NIL) and t' using the approximation join $\curlyvee_{\mathcal{T}}$. Note that the check at line 6 makes sure that the resulting decision tree is in the canonical representation for decision trees (cf. Section 4.1.1).

Algorithm 2 Make-Tree

```
 1: function BUILD(C, r)                                   ▷ C ∈ P(C), r ∈ F
 2:     if C = ∅ then
 3:         return LEAF{r}
 4:     else
 5:         c ← max C          ▷ c is the largest linear constraint appearing in C
 6:         if ¬c <_c c then                                  ▷ c is normalized
 7:             return NODE{c} : BUILD(C \ {c}, r), NIL
 8:         else                                          ▷ c is not normalized
 9:             return NODE{¬c} : NIL, BUILD(C \ {c}, r)
10:
11: function MAKE-TREE(D)                                  ▷ D ∈ P(D × F)
12:     t ← NIL
13:     for all (d, r) ∈ D do
14:         t' ← BUILD(α_C(d), r)
15:         t ← t ϒ_T t'
16:     return t
```

4.2.5 Backward Assignment

We now define the operator $\texttt{B-ASSIGN}_{\mathcal{T}}$ to handle backward reference assignments. For this we first define the operator $\texttt{B-ASSIGN}_{\mathcal{F}}$ to handle backward reference assignments of the ranking functions. The operator $\texttt{B-ASSIGN}_{\mathcal{F}}[\![p := rexp]\!] \colon \mathcal{D} \to \mathcal{F} \to \mathcal{F}$, given a heap abstraction $D \in \mathcal{D}$ of the linear constraints accumulated along the path to the leaf node and a function $r \in \mathcal{F}$ is defined in Figure 4.5.

$$
\begin{aligned}
\texttt{B-ASSIGN}_{\mathcal{F}}[\![\, p := \mathit{null} \,]\!][D]r &\stackrel{\text{def}}{=} r[a_p \leftarrow 0, c_p \leftarrow 0] + 1 \\
\texttt{B-ASSIGN}_{\mathcal{F}}[\![\, p := \mathit{new} \,]\!][D]r &\stackrel{\text{def}}{=} r[a_p \leftarrow 1, c_p \leftarrow 0] + 1 \\
\texttt{B-ASSIGN}_{\mathcal{F}}[\![\, p := q \,]\!][D]r &\stackrel{\text{def}}{=} r[a_p \leftarrow a_q, c_p \leftarrow c_q] + 1 \\
\texttt{B-ASSIGN}_{\mathcal{F}}[\![\, p := p.n \,]\!][D]r &\stackrel{\text{def}}{=} r[a_p \leftarrow a_q - 1] + 1 \\
\texttt{B-ASSIGN}_{\mathcal{F}}[\![\, p := q.n \,]\!][D]r &\stackrel{\text{def}}{=} r[a_p \leftarrow a_q - 1, c_p \leftarrow c_q] + 1 \\
\texttt{B-ASSIGN}_{\mathcal{F}}[\![\, p.f := \mathit{null} \,]\!][D]r &\stackrel{\text{def}}{=} r[a_p \leftarrow 1, c_p \leftarrow 0] + 1 \\
\texttt{B-ASSIGN}_{\mathcal{F}}[\![\, p.f := \mathit{new} \,]\!][D]r &\stackrel{\text{def}}{=} r[a_p \leftarrow 2, c_p \leftarrow 0] + 1 \\
\texttt{B-ASSIGN}_{\mathcal{F}}[\![\, p.f := p \,]\!][D]r &\stackrel{\text{def}}{=} r[a_p \leftarrow 0, c_p \leftarrow 1] + 1 \\
\texttt{B-ASSIGN}_{\mathcal{F}}[\![\, p.f := q \,]\!][D]r &\stackrel{\text{def}}{=} \begin{cases} r[a_p \leftarrow a_q + 1, & \text{if } l(p) \sqsubseteq_{\mathcal{L}} ([1,\infty],[0,\infty]) \\ c_p \leftarrow c_q] + 1 & \text{and R} \sqsubseteq_{\mathcal{R}} (p,q) \\ & \text{and NR} \sqsubseteq_{\mathcal{R}} (q,p)) \\ \top_{\mathcal{D}} & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 4.5: Backwards assignment operator $\texttt{B-ASSIGN}_{\mathcal{F}}$ for the functions auxiliary abstract domain.

We can use the information the statement gives us to substitute the symbolic variables in r. For example the statement $p := \mathit{null}$ gives us the information that p points to *null* in the post state, which means that the acyclic and cyclic lengths of p are zero, i.e., we can substitute a_p and c_p with zero in r. Then the result of $\texttt{B-ASSIGN}_{\mathcal{F}}[\![\, p := \mathit{null} \,]\!]$ is the substituted ranking function increment by one to take into account that one more program execution step is needed before termination. Note that we substitute all symbolic variables that might get affected by the changes to the partitions. The substitution for the assignment $p.f := q$ is only precise for the case in which the transformer for the heap abstraction is precise (cf. Equation 3.22).

To clarify, let us consider the following example:

Example 4.2.4. We consider a program with $\mathit{Ref} = \{p, q\}$ and $\mathit{Field} = \{f\}$. Let $r = \lambda a_p,\ c_p,\ a_q,\ c_q.\ a_p + c_p$ be a ranking function and $D \in \mathcal{D}$ be a heap abstraction representing the linear constraints representing the domain of r in the post state of the statement $p := q$. The result of the backward assignment is $r' = \texttt{B-ASSIGN}_{\mathcal{F}}[\![\, p := q \,]\!][D]r = r[a_p \leftarrow a_q, c_p \leftarrow c_q] + 1 = \lambda a_p,\ c_p,\ a_q,\ c_q.\ a_q + c_q + 1$. We have substituted the symbolic variables of p with the symbolic variables of q since p and q point to the same location in the post state.

We now define the operator $\texttt{B-ASSIGN}_{\mathcal{T}}$ to handle backward reference assignments of heap decision trees. An assignment might impact multiple constraints within the decision nodes as well as some functions within the leaf nodes. Since we cannot manipulate the linear constraints independently we collect all leaf nodes together with the path leading to them (which represents a partition of the domain of the ranking function), perform the backward assignment on both the partition and the ranking function and then create a new heap decision tree from all the resulting constraints and functions. This is implemented in Algorithm 3.

Algorithm 3 Backward Assignment

```
1:  function BWD-ASSIGN-AUX(D)                                   ▷ D ⊆ P(D × F)
2:      if D = ∅ then
3:          return ∅
4:      else                                                     ▷ D has at least one element
5:          (d, r) ← some element in D
6:          return (B-ASSIGN_D(d), B-ASSIGN_F[d](r)) ∪ BWD-ASSIGN-AUX(D \ (d, r))
7:
8:  function CLEAN(t)                                            ▷ t ∈ T_NIL
9:      if ISNIL(t) then return LEAF{⊥_F}
10:     else if ISLEAF(t) then return t
11:     else if ISNODE(t) then return NODE{t.c} : CLEAN(t.l), CLEAN(t.r)
12:
13: function BWD-ASSIGN(t_post)                                  ▷ t ∈ T
14:     d_post ← COLLECT(t_post)
15:     d_pre ← BWD-ASSIGN-AUX(d_post)
16:     t_pre ← MAKE-TREE(d_pre)
17:     return CLEAN(t_pre)
```

The main function BWD-ASSIGN, given a heap decision tree $t \in \mathcal{T}$, calls the auxiliary function COLLECT (cf. Algorithm 1) to collect all partitions together with their ranking functions (cf. Line 14 in Algorithm 3). The function then calls the auxiliary function BWD-ASSIGN-AUX which, given a set of partitions and ranking functions, applies the backward assignment to the partition and the ranking functions and returns the resulting set (cf. Line 15 in Algorithm 3). The transformed set is then merged back into a decision tree using the auxiliary function MAKE-TREE defined in Section 4.2.4 (cf. Line 16 in Algorithm 3). Finally all NIL nodes in the merged tree get substituted by leaf nodes with $\bot_{\mathcal{F}}$, using the auxiliary function CLEAN, and the resulting tree is returned (cf. Line 17 in Algorithm 3).

To clarify, let us consider the following example:

Example 4.2.5. We consider a program with $\mathit{Var} \stackrel{\text{def}}{=} \{p, q\}$ and $\mathit{Field} \stackrel{\text{def}}{=} \{f\}$. Figure 4.6 (a) shows a decision tree $t \in \mathcal{T}$ in the post state of the statement $p := p.f$.

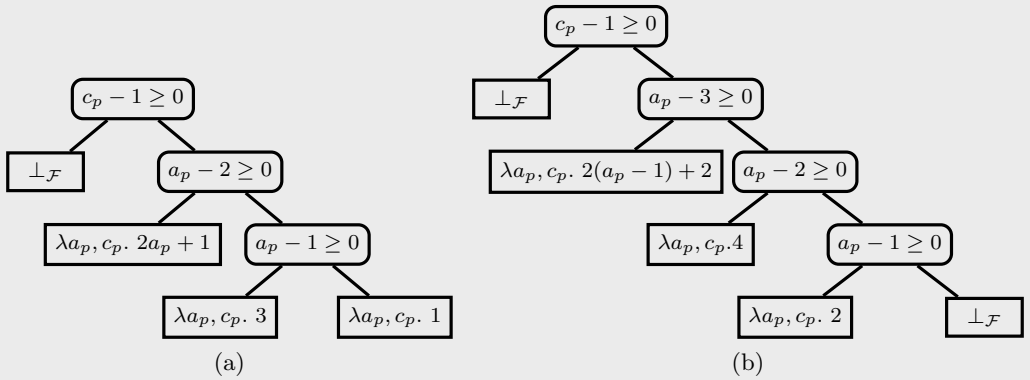


Figure 4.6: Decision tree used in Example 4.2.5.

The function BWD-ASSIGN first collects all partitions together with their ranking functions (cf. Line 14 in Algorithm 3):

$$
\begin{aligned}
(d_1, r_1) &= (\{p \mapsto ([0,\infty],[1,\infty])\}, \emptyset), \bot_{\mathcal{F}}) \\
(d_2, r_2) &= (\{p \mapsto ([2,\infty],[0,0])\}, \emptyset), \lambda a_p, c_p.\ 2a_p + 1) \\
(d_3, r_3) &= (\{p \mapsto ([1,1],[0,0])\}, \emptyset), \lambda a_p, c_p.\ 3) \\
(d_4, r_4) &= (\{p \mapsto ([0,0],[0,0])\}, \emptyset), \lambda a_p, c_p.\ 1)
\end{aligned}
$$

The second step then applies $\text{B-ASSIGN}_{\mathcal{D}}$ to the partitions and $\text{B-ASSIGN}_{\mathcal{F}}$ to the ranking functions (cf. Line 15 in Algorithm 3):

$$
\begin{aligned}
(d_1', r_1') &= (\text{B-ASSIGN}_{\mathcal{D}}[\![\, p := p.n \,]\!](d_1), \text{B-ASSIGN}_{\mathcal{F}}[\![\, p := p.n \,]\!][d_1](r_1)) \\
&= (\{p \mapsto ([0,\infty],[1,\infty])\}, \emptyset), \bot_{\mathcal{F}}) \\
(d_2', r_2') &= (\text{B-ASSIGN}_{\mathcal{D}}[\![\, p := p.n \,]\!](d_2), \text{B-ASSIGN}_{\mathcal{F}}[\![\, p := p.n \,]\!][d_1](r_2)) \\
&= (\{p \mapsto ([3,\infty],[0,0])\}, \emptyset), \lambda a_p, c_p.\ 2(a_p - 1) + 1) \\
(d_3', r_3') &= (\text{B-ASSIGN}_{\mathcal{D}}[\![\, p := p.n \,]\!](d_3), \text{B-ASSIGN}_{\mathcal{F}}[\![\, p := p.n \,]\!][d_1](r_3)) \\
&= (\{p \mapsto ([2,2],[0,0])\}, \emptyset), \lambda a_p, c_p.\ 4) \\
(d_4', r_4') &= (\text{B-ASSIGN}_{\mathcal{D}}[\![\, p := p.n \,]\!](d_4), \text{B-ASSIGN}_{\mathcal{F}}[\![\, p := p.n \,]\!][d_1](r_4)) \\
&= (\{p \mapsto ([1,1],[0,0])\}, \emptyset), \lambda a_p, c_p.\ 2)
\end{aligned}
$$

Note that the acyclic length of d'_1 is 0 since it is possible to have only a cycle in d_1 and stay in this cycle (cf. Equation 3.17 and Example 3.4.6). Since the partitions $d'_1, \ldots, d'_4$ do not overlap the result of MAKE-TREE is simply a tree representing all partitions with their ranking functions shown in Figure 4.6 (b). Note that without the final step in Algorithm 3, which removes NIL nodes in the merged decision tree, the resulting decision tree would contain a NIL node for the partition $(\{p \mapsto ([0,0],[0,0])\}, \emptyset)$ since it is not defined.

Lemma 4.2.1. In absence of run-time errors, for a reference assignment $p := rexp$ the backward assignment operator B-ASSIGN$_{\mathcal{T}}[\![p := rexp]\!]$ is a sound over-approximation of $\tau_{\mathrm{T}}[\![p := rexp]\!]$ defined in Figure 2.6, i.e., $\tau_{\mathrm{T}}[\![p := rexp]\!](\gamma_{\mathcal{D}}(t)) \preccurlyeq$ $\gamma_{\mathcal{T}}(\text{B-ASSIGN}_{\mathcal{T}}[\![p := rexp]\!](t))$.

Proof. Let $C \stackrel{\text{def}}{=} \tau_{\mathrm{T}}[\![p := rexp]\!](\gamma_{\mathcal{T}}(t))$ and let $A \stackrel{\text{def}}{=} \gamma_{\mathcal{T}}(\text{B-ASSIGN}_{\mathcal{T}}[\![p := rexp]\!](t))$. We prove that $\mathrm{dom}(C) \supseteq \mathrm{dom}(A)$ and $\forall s \in \mathrm{dom}(A)\colon C(s) \leq A(s)$.

Let us assume, by absurd, that $\mathrm{dom}(C) \subset \mathrm{dom}(A)$. Then, there exists a state $s \in \mathit{State}$ such that $s \in \mathrm{dom}(A)$ and $s \notin \mathrm{dom}(C)$. This means that there is some defined node in B-ASSIGN$_{\mathcal{T}}[\![p := rexp]\!](t)$ which is responsible for s. Since MAKE-TREE merges both over-approximations of the terminating states and over-approximations of the non-terminating states and favors undefined nodes, s cannot be a non-terminating state which means $s \in \mathrm{dom}(C)$ which is absurd. Therefore, we conclude that $\mathrm{dom}(C) \supseteq \mathrm{dom}(A)$.

We now show $\forall s \in \mathrm{dom}(A)\colon C(s) \leq A(s)$. We have, by the definition of $\tau_{\mathrm{T}}[\![p := rexp]\!]$ that $C(s) = \sup\{(\gamma_T[D]t)(e[p \leftarrow v], s) + 1 \mid v \in [\![rexp]\!]\}$. As noted in the definition of the semantics of reference expression (cf. Section 2.2), we either have one element in $[\![rexp]\!]$ or zero in case of a *null* dereference. Since we assume that we have no run-time errors we have $C(s) = (\gamma_T[D]t)(e[p \leftarrow v], s) + 1$ where $v \in [\![rexp]\!]$. By the definition of B-ASSIGN$_{\mathcal{T}}$ (cf. Algorithm 3), we apply B-ASSIGN$_{\mathcal{F}}[D]$ to each ranking function r in the leaf nodes of t, where $D \in \mathcal{D}$ is a heap abstraction of the partition over which r is defined. $C(s) \leq A(s)$ follows from the fact that the substitutions made by B-ASSIGN$_{\mathcal{F}}[D]$ do not change the value of the ranking function, while making sure that the assignment on the partition over which the function is defined does not impact the ranking function. □

Lemma 4.2.2. In absence of run-time errors, for a reference assignment $p.f := rexp$ the backward assignment operator B-ASSIGN$_{\mathcal{T}}[\![p.f := rexp]\!]$ is a sound over-approximation of $\tau_{\mathrm{T}}[\![p.f := rexp]\!]$ defined in Figure 2.6, i.e., $\tau_{\mathrm{T}}[\![p.f := rexp]\!](\gamma_{\mathcal{D}}(t)) \preccurlyeq$ $\gamma_{\mathcal{T}}(\text{B-ASSIGN}_{\mathcal{T}}[\![p.f := rexp]\!](t))$.

Proof. Analogous to the proof of Lemma 4.2.1. □

4.2.6 Filter

We now define the operator FILTER$_{\mathcal{T}}$ to handle filter statements. FILTER$_{\mathcal{D}}$ preserves all paths in the decision trees that are feasible and disregards all paths that can never be

followed according to the filter condition. This is implemented in Algorithm 4.

Algorithm 4 Filter

```
1: function FILTER-AUX(D)                                   ▷ D ⊆ P(D × F)
2:     if D = ∅ then
3:         return ∅
4:     else                                                 ▷ D has at least one element
5:         (d, r) ← some element in D
6:         return (FILTER_D(d), STEP(r)) ∪ FILTER-AUX(D \ (d, r))
7:
8: function FILTER(t_post)                                  ▷ t ∈ T
9:     d_post ← COLLECT(t_post)
10:    d_pre ← FILTER-AUX(d_post)
11:    t_pre ← MAKE-TREE(d_pre)
12:    return t_pre
```

The algorithm for filter is the same as the algorithm for backward assignment (cf. Algorithm 3), except that we use $\text{FILTER}_{\mathcal{D}}$ for the domain and STEP on the ranking function (cf. Line 6 in Algorithm 4) and that we do not remove NIL nodes since they are used to represent empty parts of the ranking function.

4.2.7 Abstract Definite Termination Semantics

The operators for join, meet, step and widening of the value decision trees and the operators for backward assignment and filter of the heap decision tree can now be used to define the abstract termination semantics analogous to [Urb15b, Section 5.3].

$$
\begin{array}{l}
\tau^\natural_{\mathrm{T}}[\![\, \texttt{skip} \,]\!]t \stackrel{\text{def}}{=} \text{STEP}_{\mathcal{T}}(t) \\
\tau^\natural_{\mathrm{T}}[\![\, x := aexp \,]\!]t \stackrel{\text{def}}{=} \text{B-ASSIGN}_{\mathcal{T}}[\![\, x := aexp \,]\!](t) \\
\tau^\natural_{\mathrm{T}}[\![\, x.f := aexp \,]\!]t \stackrel{\text{def}}{=} \text{B-ASSIGN}_{\mathcal{T}}[\![\, x.f := aexp \,]\!](t) \\
\tau^\natural_{\mathrm{T}}[\![\, x := rexp \,]\!]t \stackrel{\text{def}}{=} \text{B-ASSIGN}_{\mathcal{T}}[\![\, x := rexp \,]\!](t) \\
\tau^\natural_{\mathrm{T}}[\![\, x.f := rexp \,]\!]t \stackrel{\text{def}}{=} \text{B-ASSIGN}_{\mathcal{T}}[\![\, x.f := rexp \,]\!](t) \\
\tau^\natural_{\mathrm{T}}[\![\, \texttt{if } bexp \texttt{ then } stmt_1 \texttt{ else } stmt_2 \texttt{ fi} \,]\!]t \stackrel{\text{def}}{=} F_1^\natural[t] \curlyvee_{\mathcal{T}} F_2^\natural[f] \\
\quad F_1^\natural[t] \stackrel{\text{def}}{=} \text{FILTER}_{\mathcal{T}}[\![\, bexp \,]\!](\tau^\natural_{\mathrm{T}}[\![\, stmt_1 \,]\!]t) \\
\quad F_2^\natural[t] \stackrel{\text{def}}{=} \text{FILTER}_{\mathcal{T}}[\![\, \texttt{not } bexp \,]\!](\tau^\natural_{\mathrm{T}}[\![\, stmt_2 \,]\!]t) \\
\tau^\natural_{\mathrm{T}}[\![\, \texttt{while } {}^l bexp \texttt{ do } stmt \texttt{ od} \,]\!]t \stackrel{\text{def}}{=} \text{lfp}^\natural\, \phi^\natural_{\mathrm{T}} \\
\quad \phi^\natural_{\mathrm{T}}(x) \stackrel{\text{def}}{=} F^\natural[x] \curlyvee_{\mathcal{T}} \text{FILTER}_{\mathcal{T}}[\![\, \texttt{not } bexp \,]\!](t) \\
\quad F^\natural[x] \stackrel{\text{def}}{=} \text{FILTER}_{\mathcal{T}}[\![\, bexp \,]\!](\tau^\natural_{\mathrm{T}}[\![\, stmt \,]\!]x) \\
\tau^\natural_{\mathrm{T}}[\![\, stmt_1 \;\; stmt_2 \,]\!]t \stackrel{\text{def}}{=} \tau^\natural_{\mathrm{T}}[\![\, stmt_1 \,]\!](\tau^\natural_{\mathrm{T}}[\![\, stmt_2 \,]\!]t)
\end{array}
$$

Figure 4.7: Abstract termination semantics of instructions *stmt*. The new rules are highlighted in gray.

In Figure 4.7 we define the abstract termination semantics $\tau^\natural_{\mathrm{T}}[\![\, stmt \,]\!] \colon \mathcal{T} \to \mathcal{T}$, for each program instruction *stmt* analogously to the original definition in [Urb15b, Section 5.3]. The new rules, highlighted in gray, are analogous to the original rules. The function $\tau^\natural_{\mathrm{T}}[\![\, stmt \,]\!]$, given a heap decision tree over-approximating the ranking function in the post state of the instruction *stmt* returns a heap decision tree, which over-approximates the ranking function in the pre state of *stmt*. The termination semantics $\tau^\natural_{\mathrm{T}}[\![\, prog \,]\!] \in \mathcal{T}$ of program *prog* returns a heap decision tree over-approximating the ranking function corresponding to the initial program control point $i[\![prog]\!]$ based on the input LEAF : $\lambda a_{p_1}, c_{p_1}, \ldots, a_{p_k}, c_{p_k}.\, 0$:

Definition 4.2.1 (Abstract Termination Semantics)**.** The *abstract termination semantics* $\tau^\natural_{\mathrm{T}}[\![\, prog \,]\!] \in \mathcal{T}$ of a program *prog* is:

$$\tau^\natural_{\mathrm{T}}[\![\, prog \,]\!] = \tau^\natural_{\mathrm{T}}[\![\, stmt \,]\!] \stackrel{\text{def}}{=} \tau^\natural_{\mathrm{T}}[\![\, stmt \,]\!]\text{LEAF} : \lambda a_{p_1}, c_{p_1}, \ldots, a_{p_k}, c_{p_k}.\, 0 \tag{4.12}$$

where the abstract termination semantics $\tau^\natural_{\mathrm{T}}[\![\, stmt \,]\!] \in \mathcal{T} \to \mathcal{T}$ of each program instruction *stmt* is defined in Figure 4.7.

Theorem 4.2.3. The abstract termination semantics $\tau^\natural_{\mathrm{T}}[\![\, prog \,]\!] \in \mathcal{T}$ is a sound abstraction of the termination semantics $\tau_{\mathrm{T}}[\![\, prog \,]\!] \in \mathit{State} \to \mathbb{O}$, i.e., $\tau_{\mathrm{T}}[\![\, prog \,]\!] \preccurlyeq \gamma_{\mathcal{T}}(\tau^\natural_{\mathrm{T}}[\![\, prog \,]\!])$.

Proof (Sketch). The proof follows from the soundness of the operators of the value decision trees for step, test, loop instruction and the widening (cf. Lemma 5.2.8, Lemma 5.2.15, Lemma 5.2.16 and Lemma 5.2.24 in [Urb15b, Appendix A]), the soundness of the operators for backward assignment of the heap decision trees (cf. Lemma 4.2.1 and Lemma 4.2.2) and the assumption that $\mathit{Num} = \emptyset$. □

4.2.8 Example

We conclude this section with a detailed analysis of an example program using the heap decision trees. To make the presentation easier, instead of showing the decision tree, we list the partitions together with their ranking functions, as would be returned by the COLLECT algorithm defined in Section 4.2.3, separated by a question mark.

Example 4.2.6. Let us consider again the list traversal of Example 1.0.1:

$$\begin{array}{l}\texttt{while}\ {}^{1}(l \neq \mathit{null})\ \texttt{do}\\ \quad {}^{2}l := l.\mathit{next}\\ \texttt{od}^{3}\end{array}$$

We present the analysis of the program using the field *next* as parameter for the heap abstract domain. The starting point is the zero function at the program final control point:
3 :

$$\begin{array}{l}(\{l \mapsto ([0,0],[0,0])\},\emptyset)\ ?\ \lambda a_l, c_l.\ 0\\ (\{l \mapsto ([1,\infty],[0,0])\},\emptyset)\ ?\ \lambda a_l, c_l.\ 0\\ (\{l \mapsto ([0,0],[1,\infty])\},\emptyset)\ ?\ \lambda a_l, c_l.\ 0\\ (\{l \mapsto ([1,\infty],[1,\infty]\},\emptyset)\ ?\ \lambda a_l, c_l.\ 0\end{array}$$

Note that we have the fixed partitions as defined in Section 4.2.2. The ranking function is then propagated backwards towards the program initial control point taking the loop into account:
1 :

$$\begin{array}{l}(\{l \mapsto ([0,0],[0,0])\},\emptyset)\ ?\ \lambda a_l, c_l.\ 1\\ (\{l \mapsto ([1,\infty],[0,0])\},\emptyset)\ ?\ \bot_{\mathcal{F}}\\ (\{l \mapsto ([0,0],[1,\infty])\},\emptyset)\ ?\ \bot_{\mathcal{F}}\\ (\{l \mapsto ([1,\infty],[1,\infty]\},\emptyset)\ ?\ \bot_{\mathcal{F}}\end{array}$$

The first iteration of the loop is able to conclude that the program terminates in at most one program step if the loop condition $l \neq \mathit{null}$ is not satisfied. Then, at program control point 2, the operator B-ASSIGN$_T$ gives us the following decision tree:

2 :

$$
\begin{aligned}
(\{l \mapsto ([0,0],[0,0])\},\emptyset) \ ? \ \bot_{\mathcal{F}} \\
(\{l \mapsto ([1,1],[0,0])\},\emptyset) \ ? \ \lambda a_l, c_l.\ 2 \\
(\{l \mapsto ([2,\infty],[0,0])\},\emptyset) \ ? \ \bot_{\mathcal{F}} \\
(\{l \mapsto ([0,0],[1,\infty])\},\emptyset) \ ? \ \bot_{\mathcal{F}} \\
(\{l \mapsto ([1,1],[1,\infty])\},\emptyset) \ ? \ \bot_{\mathcal{F}} \\
(\{l \mapsto ([2,\infty],[1,\infty])\},\emptyset) \ ? \ \bot_{\mathcal{F}}
\end{aligned}
$$

The only terminating partition is the state where the acyclic length of l is one, i.e. l is a list of length one. The second iteration concludes that the program terminates in at most three steps when l has acyclic length 1 and cyclic length 0, and in at most one step if l is *null*:

1 :

$$
\begin{aligned}
(\{l \mapsto ([0,0],[0,0])\},\emptyset) \ ? \ \lambda a_l, c_l.\ 1 \\
(\{l \mapsto ([1,1],[0,0])\},\emptyset) \ ? \ \lambda a_l, c_l.\ 3 \\
(\{l \mapsto ([2,\infty],[0,0])\},\emptyset) \ ? \ \bot_{\mathcal{F}} \\
(\{l \mapsto ([0,0],[1,\infty])\},\emptyset) \ ? \ \bot_{\mathcal{F}} \\
(\{l \mapsto ([1,1],[1,\infty])\},\emptyset) \ ? \ \bot_{\mathcal{F}} \\
(\{l \mapsto ([2,\infty],[1,\infty])\},\emptyset) \ ? \ \bot_{\mathcal{F}}
\end{aligned}
$$

This pattern continues in the third iteration:

1 :

$$
\begin{aligned}
(\{l \mapsto ([0,0],[0,0])\},\emptyset) \ ? \ \lambda a_l, c_l.\ 1 \\
(\{l \mapsto ([1,1],[0,0])\},\emptyset) \ ? \ \lambda a_l, c_l.\ 3 \\
(\{l \mapsto ([2,2],[0,0])\},\emptyset) \ ? \ \lambda a_l, c_l.\ 5 \\
(\{l \mapsto ([3,\infty],[0,0])\},\emptyset) \ ? \ \bot_{\mathcal{F}} \\
(\{l \mapsto ([0,0],[1,\infty])\},\emptyset) \ ? \ \bot_{\mathcal{F}} \\
(\{l \mapsto ([1,1],[1,\infty])\},\emptyset) \ ? \ \bot_{\mathcal{F}} \\
(\{l \mapsto ([2,2],[1,\infty])\},\emptyset) \ ? \ \bot_{\mathcal{F}} \\
(\{l \mapsto ([3,\infty],[1,\infty])\},\emptyset) \ ? \ \bot_{\mathcal{F}}
\end{aligned}
$$

At which point the widening (cf. Section 4.1.5) extrapolates the ranking function on the partitions over which it is not yet defined:

1 :

$$
\begin{aligned}
(\{l \mapsto ([0,0],[0,0])\},\emptyset) \ ? \ \lambda a_l, c_l.\ 1 \\
(\{l \mapsto ([1,1],[0,0])\},\emptyset) \ ? \ \lambda a_l, c_l.\ 3 \\
(\{l \mapsto ([2,2],[0,0])\},\emptyset) \ ? \ \lambda a_l, c_l.\ 5 \\
(\{l \mapsto ([3,\infty],[0,0])\},\emptyset) \ ? \ \lambda a_l, c_l.\ 2a_l + 1 \\
(\{l \mapsto ([0,0],[1,\infty])\},\emptyset) \ ? \ \bot_{\mathcal{F}} \\
(\{l \mapsto ([1,1],[1,\infty])\},\emptyset) \ ? \ \bot_{\mathcal{F}} \\
(\{l \mapsto ([2,2],[1,\infty])\},\emptyset) \ ? \ \bot_{\mathcal{F}} \\
(\{l \mapsto ([3,\infty],[1,\infty])\},\emptyset) \ ? \ \bot_{\mathcal{F}}
\end{aligned}
$$

Note that the widening extrapolated the ranking function for the remaining acyclic partitions to be $\lambda a_l, c_l.\ 2a_l + 1$. The fourth iteration passes all checks (cf. Section 4.1.5) and is a fixpoint. The fixpoint represents the following piecewise-defined ranking function:

$$
\lambda(e,s).\ \begin{cases} 1 & a(l,next,s) = 0 \wedge c(l,next,s) = 0 \\ 3 & a(l,next,s) = 1 \wedge c(l,next,s) = 0 \\ 5 & a(l,next,s) = 2 \wedge c(l,next,s) = 0 \\ 2 \cdot a(l,next,s) + 1 & a(l,next,s) \geq 3 \wedge c(l,next,s) = 0 \\ \text{undefined} & \text{otherwise} \end{cases}
$$

where the functions a and c are defined in Definition 3.2.1. The ranking function proves that the program is terminating for all acyclic lists.

4.3 Combined Decision Trees

In this section we informally show how we can combine the value decision trees of Section 4.1 and the heap decision trees of Section 4.2. The combination is based on the fact that the numerical domains used in the value decision trees only abstract numerical variables and the heap abstract domain used in the heap decision trees only abstracts references. We can create a combined abstraction where the numerical abstraction is responsible for the numerical variables and the heap abstraction is responsible for the references.

The formalization of the combined decision trees is then analogous to the heap decision trees with the following differences: we use the combined abstraction as underlying abstraction for the linear constraints. Since the value decision trees have linear constraints over the numerical variables and the heap decision trees have linear constraints over the symbolic variables a correspondence between elements of the combined abstract domain and the linear constraints can easily be defined. For the functions auxiliary domain we

use a combination of the affine functions over numerical variables (cf. Section 4.1.1) and the affine functions over the acyclic and cyclic lengths of references (cf. Section 4.2.1). The backward assignment and filter are defined analogously to the definition in the heap decision trees but, based on the type of the statement, use the corresponding underlying abstraction.

4.3.1 Example

We conclude this section with a detailed analysis of an example program using the combined decision trees. As in Section 4.2.8, instead of showing the tree, we list the partitions together with their ranking functions, as would be returned by the COLLECT algorithm defined in Section 4.2.3, separated by a question mark. Note that an element of the combined domain consists of a numerical abstraction for the integer variables *Num*, and the heap abstraction for the references *Ref*. To facilitate the presentation, we will list the constraints for the numerical domain.

Example 4.3.1. Let us consider the following program:

$$
\begin{array}{l}
\texttt{while } {}^{1}(i < 10) \texttt{ do} \\
\quad {}^{2}i := i + 1 \\
\texttt{od}^{3} \\
\texttt{while } {}^{4}(l \neq \mathit{null}) \texttt{ do} \\
\quad {}^{5}l := l.\mathit{next} \\
\texttt{od}^{6}
\end{array}
$$

Note that we have $\mathit{Num} \stackrel{\text{def}}{=} \{i\}$ and $\mathit{Ref} \stackrel{\text{def}}{=} \{l\}$. We present the analysis of the program using the interval domain as the numerical domain and using the field *next* as parameter for the heap abstract domain. The starting point is the zero function at the program final control point:
6:

$$
\begin{array}{r}
\emptyset, (\{l \to ([0,0],[0,0])\}, \emptyset) \ ? \ \lambda a_l, c_l, i.\ 0 \\
\emptyset, (\{l \to ([1,\infty],[0,0])\}, \emptyset) \ ? \ \lambda a_l, c_l, i.\ 0 \\
\emptyset, (\{l \to ([0,0],[1,\infty])\}, \emptyset) \ ? \ \lambda a_l, c_l, i.\ 0 \\
\emptyset, (\{l \to ([1,\infty],[1,\infty])\}, \emptyset) \ ? \ \lambda a_l, c_l, i.\ 0
\end{array}
$$

Note that, as in Example 4.2.6, we have the fixed partitions as defined in Section 3.4.1 and that we have no constraints in the numerical domain, i.e. i can have any value in $\mathbb{Z}$. We will not show the analysis of the second loop since it is already discussed in Example 4.2.6. The resulting decision tree at the third program control point is:

3:

$$\begin{aligned}
\emptyset, (\{l \mapsto ([0,0],[0,0])\}, \emptyset) \ ? \ \lambda a_l, c_l, i.\ 1 \\
\emptyset, (\{l \mapsto ([1,1],[0,0])\}, \emptyset) \ ? \ \lambda a_l, c_l, i.\ 3 \\
\emptyset, (\{l \mapsto ([2,2],[0,0])\}, \emptyset) \ ? \ \lambda a_l, c_l, i.\ 5 \\
\emptyset, (\{l \mapsto ([3,\infty],[0,0])\}, \emptyset) \ ? \ \lambda a_l, c_l, i.\ 2a_l + 1 \\
\emptyset, (\{l \mapsto ([0,0],[1,\infty])\}, \emptyset) \ ? \ \bot_{\mathcal{F}} \\
\emptyset, (\{l \mapsto ([1,1],[1,\infty])\}, \emptyset) \ ? \ \bot_{\mathcal{F}} \\
\emptyset, (\{l \mapsto ([2,2],[1,\infty])\}, \emptyset) \ ? \ \bot_{\mathcal{F}} \\
\emptyset, (\{l \mapsto ([3,\infty],[1,\infty])\}, \emptyset) \ ? \ \bot_{\mathcal{F}}
\end{aligned}$$

The ranking function is then propagated backwards towards the program initial control point taking the first loop into account:
1:

$$\begin{aligned}
\{i \leq 9\}, (\{l \mapsto ([0,0],[0,0])\}, \emptyset) \ ? \ \bot_{\mathcal{F}} \\
\{i \geq 10\}, (\{l \mapsto ([0,0],[0,0])\}, \emptyset) \ ? \ \lambda a_l, c_l, i.\ 2 \\
\{i \geq 10\}, (\{l \mapsto ([1,1],[0,0])\}, \emptyset) \ ? \ \lambda a_l, c_l, i.\ 4 \\
\{i \geq 10\}, (\{l \mapsto ([2,2],[0,0])\}, \emptyset) \ ? \ \lambda a_l, c_l, i.\ 6 \\
\{i \geq 10\}, (\{l \mapsto ([3,\infty],[0,0])\}, \emptyset) \ ? \ \lambda a_l, c_l, i.\ 2a_l + 2 \\
\{i \geq 10\}, (\{l \mapsto ([0,0],[1,\infty])\}, \emptyset) \ ? \ \bot_{\mathcal{F}} \\
\{i \geq 10\}, (\{l \mapsto ([1,1],[1,\infty])\}, \emptyset) \ ? \ \bot_{\mathcal{F}} \\
\{i \geq 10\}, (\{l \mapsto ([2,2],[1,\infty])\}, \emptyset) \ ? \ \bot_{\mathcal{F}} \\
\{i \geq 10\}, (\{l \mapsto ([3,\infty],[1,\infty])\}, \emptyset) \ ? \ \bot_{\mathcal{F}}
\end{aligned}$$

The first iteration is able to conclude that if i is greater than or equal to 10, the program terminates for the acyclic lists using the ranking function from the previous loop. Note that the values are increased by one because of the test in the first loop. Then, at control point **2**, the operator B-ASSIGN$_{\mathcal{T}}$ gives us the following partitions:

2:

$$\begin{aligned}
\{i \leq 8\}, (\{l \mapsto ([0,0],[0,0])\}, \emptyset) \ ? \ \bot_{\mathcal{F}} \\
\{i \geq 9\}, (\{l \mapsto ([0,0],[0,0])\}, \emptyset) \ ? \ \lambda a_l, c_l, i.\ 3 \\
\{i \geq 9\}, (\{l \mapsto ([1,1],[0,0])\}, \emptyset) \ ? \ \lambda a_l, c_l, i.\ 5 \\
\{i \geq 9\}, (\{l \mapsto ([2,2],[0,0])\}, \emptyset) \ ? \ \lambda a_l, c_l, i.\ 7 \\
\{i \geq 9\}, (\{l \mapsto ([3,\infty],[0,0])\}, \emptyset) \ ? \ \lambda a_l, c_l, i.\ 2a_l + 3 \\
\{i \geq 9\}, (\{l \mapsto ([0,0],[1,\infty])\}, \emptyset) \ ? \ \bot_{\mathcal{F}} \\
\{i \geq 9\}, (\{l \mapsto ([1,1],[1,\infty])\}, \emptyset) \ ? \ \bot_{\mathcal{F}} \\
\{i \geq 9\}, (\{l \mapsto ([2,2],[1,\infty])\}, \emptyset) \ ? \ \bot_{\mathcal{F}} \\
\{i \geq 9\}, (\{l \mapsto ([3,\infty],[1,\infty])\}, \emptyset) \ ? \ \bot_{\mathcal{F}}
\end{aligned}$$

The assignment to i changes the numerical value the of the partitions over which the decision tree is defined and increases the value of the ranking function by one. The second iteration gives us the following decision tree:

1:

$$\begin{aligned}
\{i \leq 8\}, (\{l \mapsto ([0,\infty],[0,\infty])\}, \emptyset) \ ? \ \bot_{\mathcal{F}} \\
\{i = 9\}, (\{l \mapsto ([0,0],[0,0])\}, \emptyset) \ ? \ \lambda a_l, c_l, i.\ 4 \\
\{i = 9\}, (\{l \mapsto ([1,1],[0,0])\}, \emptyset) \ ? \ \lambda a_l, c_l, i.\ 6 \\
\{i = 9\}, (\{l \mapsto ([2,2],[0,0])\}, \emptyset) \ ? \ \lambda a_l, c_l, i.\ 8 \\
\{i = 9\}, (\{l \mapsto ([3,\infty],[0,0])\}, \emptyset) \ ? \ \lambda a_l, c_l, i.\ 2a_l + 4 \\
\{i = 9\}, (\{l \mapsto ([0,0],[1,\infty])\}, \emptyset) \ ? \ \bot_{\mathcal{F}} \\
\{i = 9\}, (\{l \mapsto ([1,1],[1,\infty])\}, \emptyset) \ ? \ \bot_{\mathcal{F}} \\
\{i = 9\}, (\{l \mapsto ([2,2],[1,\infty])\}, \emptyset) \ ? \ \bot_{\mathcal{F}} \\
\{i = 9\}, (\{l \mapsto ([3,\infty],[1,\infty])\}, \emptyset) \ ? \ \bot_{\mathcal{F}} \\
\{i \geq 10\}, (\{l \mapsto ([0,0],[0,0])\}, \emptyset) \ ? \ \lambda a_l, c_l, i.\ 2 \\
\{i \geq 10\}, (\{l \mapsto ([1,1],[0,0])\}, \emptyset) \ ? \ \lambda a_l, c_l, i.\ 4 \\
\{i \geq 10\}, (\{l \mapsto ([2,2],[0,0])\}, \emptyset) \ ? \ \lambda a_l, c_l, i.\ 6 \\
\{i \geq 10\}, (\{l \mapsto ([3,\infty],[0,0])\}, \emptyset) \ ? \ \lambda a_l, c_l, i.\ 2a_l + 2 \\
\{i \geq 10\}, (\{l \mapsto ([0,0],[1,\infty])\}, \emptyset) \ ? \ \bot_{\mathcal{F}} \\
\{i \geq 10\}, (\{l \mapsto ([1,1],[1,\infty])\}, \emptyset) \ ? \ \bot_{\mathcal{F}} \\
\{i \geq 10\}, (\{l \mapsto ([2,2],[1,\infty])\}, \emptyset) \ ? \ \bot_{\mathcal{F}} \\
\{i \geq 10\}, (\{l \mapsto ([3,\infty],[1,\infty])\}, \emptyset) \ ? \ \bot_{\mathcal{F}}
\end{aligned}$$

This pattern continues in the third iteration:
1:

$$
\begin{aligned}
\{i \leq 7\}, (\{l \mapsto ([0,\infty],[0,\infty])\}, \emptyset) \ ?\ & \bot_{\mathcal{F}} \\
\{i = 8\}, (\{l \mapsto ([0,0],[0,0])\}, \emptyset) \ ?\ & \lambda a_l, c_l, i.\ 6 \\
\{i = 8\}, (\{l \mapsto ([1,1],[0,0])\}, \emptyset) \ ?\ & \lambda a_l, c_l, i.\ 8 \\
\{i = 8\}, (\{l \mapsto ([2,2],[0,0])\}, \emptyset) \ ?\ & \lambda a_l, c_l, i.\ 10 \\
\{i = 8\}, (\{l \mapsto ([3,\infty],[0,0])\}, \emptyset) \ ?\ & \lambda a_l, c_l, i.\ 2a_l + 6 \\
\{i = 8\}, (\{l \mapsto ([0,0],[1,\infty])\}, \emptyset) \ ?\ & \bot_{\mathcal{F}} \\
\{i = 8\}, (\{l \mapsto ([1,1],[1,\infty])\}, \emptyset) \ ?\ & \bot_{\mathcal{F}} \\
\{i = 8\}, (\{l \mapsto ([2,2],[1,\infty])\}, \emptyset) \ ?\ & \bot_{\mathcal{F}} \\
\{i = 8\}, (\{l \mapsto ([3,\infty],[1,\infty])\}, \emptyset) \ ?\ & \bot_{\mathcal{F}} \\
\{i = 9\}, (\{l \mapsto ([0,0],[0,0])\}, \emptyset) \ ?\ & \lambda a_l, c_l, i.\ 4 \\
\{i = 9\}, (\{l \mapsto ([1,1],[0,0])\}, \emptyset) \ ?\ & \lambda a_l, c_l, i.\ 6 \\
\{i = 9\}, (\{l \mapsto ([2,2],[0,0])\}, \emptyset) \ ?\ & \lambda a_l, c_l, i.\ 8 \\
\{i = 9\}, (\{l \mapsto ([3,\infty],[0,0])\}, \emptyset) \ ?\ & \lambda a_l, c_l, i.\ 2a_l + 4 \\
\{i = 9\}, (\{l \mapsto ([0,0],[1,\infty])\}, \emptyset) \ ?\ & \bot_{\mathcal{F}} \\
\{i = 9\}, (\{l \mapsto ([1,1],[1,\infty])\}, \emptyset) \ ?\ & \bot_{\mathcal{F}} \\
\{i = 9\}, (\{l \mapsto ([2,2],[1,\infty])\}, \emptyset) \ ?\ & \bot_{\mathcal{F}} \\
\{i = 9\}, (\{l \mapsto ([3,\infty],[1,\infty])\}, \emptyset) \ ?\ & \bot_{\mathcal{F}} \\
\{i \geq 10\}, (\{l \mapsto ([0,0],[0,0])\}, \emptyset) \ ?\ & \lambda a_l, c_l, i.\ 2 \\
\{i \geq 10\}, (\{l \mapsto ([1,1],[0,0])\}, \emptyset) \ ?\ & \lambda a_l, c_l, i.\ 4 \\
\{i \geq 10\}, (\{l \mapsto ([2,2],[0,0])\}, \emptyset) \ ?\ & \lambda a_l, c_l, i.\ 6 \\
\{i \geq 10\}, (\{l \mapsto ([3,\infty],[0,0])\}, \emptyset) \ ?\ & \lambda a_l, c_l, i.\ 2a_l + 2 \\
\{i \geq 10\}, (\{l \mapsto ([0,0],[1,\infty])\}, \emptyset) \ ?\ & \bot_{\mathcal{F}} \\
\{i \geq 10\}, (\{l \mapsto ([1,1],[1,\infty])\}, \emptyset) \ ?\ & \bot_{\mathcal{F}} \\
\{i \geq 10\}, (\{l \mapsto ([2,2],[1,\infty])\}, \emptyset) \ ?\ & \bot_{\mathcal{F}} \\
\{i \geq 10\}, (\{l \mapsto ([3,\infty],[1,\infty])\}, \emptyset) \ ?\ & \bot_{\mathcal{F}}
\end{aligned}
$$

At which point the widening (cf. Section 4.1.5) extrapolates the ranking function on the partitions over which it is not yet defined:

1:

$$\begin{aligned}
\{i \leq 8\}, (\{l \mapsto ([0,0],[0,0])\}, \emptyset) \ ? \ \lambda a_l, c_l, i. \ -2i + 22 \\
\{i \leq 8\}, (\{l \mapsto ([1,1],[0,0])\}, \emptyset) \ ? \ \lambda a_l, c_l, i. \ -2i + 24 \\
\{i \leq 8\}, (\{l \mapsto ([2,2],[0,0])\}, \emptyset) \ ? \ \lambda a_l, c_l, i. \ -2i + 26 \\
\{i \leq 8\}, (\{l \mapsto ([3,\infty],[0,0])\}, \emptyset) \ ? \ \lambda a_l, c_l, i. \ 2a_l - 2i + 22 \\
\{i \leq 8\}, (\{l \mapsto ([0,0],[1,\infty])\}, \emptyset) \ ? \ \bot_{\mathcal{F}} \\
\{i \leq 8\}, (\{l \mapsto ([1,1],[1,\infty])\}, \emptyset) \ ? \ \bot_{\mathcal{F}} \\
\{i \leq 8\}, (\{l \mapsto ([2,2],[1,\infty])\}, \emptyset) \ ? \ \bot_{\mathcal{F}} \\
\{i \leq 8\}, (\{l \mapsto ([3,\infty],[1,\infty])\}, \emptyset) \ ? \ \bot_{\mathcal{F}} \\
\{i = 9\}, (\{l \mapsto ([0,0],[0,0])\}, \emptyset) \ ? \ \lambda a_l, c_l, i. \ 4 \\
\{i = 9\}, (\{l \mapsto ([1,1],[0,0])\}, \emptyset) \ ? \ \lambda a_l, c_l, i. \ 6 \\
\{i = 9\}, (\{l \mapsto ([2,2],[0,0])\}, \emptyset) \ ? \ \lambda a_l, c_l, i. \ 8 \\
\{i = 9\}, (\{l \mapsto ([3,\infty],[0,0])\}, \emptyset) \ ? \ \lambda a_l, c_l, i. \ 2a_l + 4 \\
\{i = 9\}, (\{l \mapsto ([0,0],[1,\infty])\}, \emptyset) \ ? \ \bot_{\mathcal{F}} \\
\{i = 9\}, (\{l \mapsto ([1,1],[1,\infty])\}, \emptyset) \ ? \ \bot_{\mathcal{F}} \\
\{i = 9\}, (\{l \mapsto ([2,2],[1,\infty])\}, \emptyset) \ ? \ \bot_{\mathcal{F}} \\
\{i = 9\}, (\{l \mapsto ([3,\infty],[1,\infty])\}, \emptyset) \ ? \ \bot_{\mathcal{F}} \\
\{i \geq 10\}, (\{l \mapsto ([0,0],[0,0])\}, \emptyset) \ ? \ \lambda a_l, c_l, i. \ 2 \\
\{i \geq 10\}, (\{l \mapsto ([1,1],[0,0])\}, \emptyset) \ ? \ \lambda a_l, c_l, i. \ 4 \\
\{i \geq 10\}, (\{l \mapsto ([2,2],[0,0])\}, \emptyset) \ ? \ \lambda a_l, c_l, i. \ 6 \\
\{i \geq 10\}, (\{l \mapsto ([3,\infty],[0,0])\}, \emptyset) \ ? \ \lambda a_l, c_l, i. \ 2a_l + 2 \\
\{i \geq 10\}, (\{l \mapsto ([0,0],[1,\infty])\}, \emptyset) \ ? \ \bot_{\mathcal{F}} \\
\{i \geq 10\}, (\{l \mapsto ([1,1],[1,\infty])\}, \emptyset) \ ? \ \bot_{\mathcal{F}} \\
\{i \geq 10\}, (\{l \mapsto ([2,2],[1,\infty])\}, \emptyset) \ ? \ \bot_{\mathcal{F}} \\
\{i \geq 10\}, (\{l \mapsto ([3,\infty],[1,\infty])\}, \emptyset) \ ? \ \bot_{\mathcal{F}}
\end{aligned}$$

Note that the widening extrapolated the ranking function for the remaining values of i. The fourth iteration passes all checks (cf. Section 4.1.5) and is a fixpoint.

5 Implementation

5.1 FuncTion

We have implemented the heap abstract domain (cf. Chapter 3), the heap decision trees (cf. Section 4.2) and the combined decision trees (cf. Section 4.3) in FuncTion. This includes the following changes to FuncTion:

Parser The parser was extended to support the new language constructs described in Section 2.1. We have implemented a more general grammar than described in this report which allows nested dereferences such as $p.f.f$. We also collect type information during the semantic analysis to be able to distinguish between integer variables and references.

Bounds In FuncTion abstractions of the invariance semantics (cf. Section 2.3) are implemented using the *BOUND* interface. BOUND implementations represent the *domain* of a piecewise-defined ranking function and are used as the underlying abstraction for its linear constraints (cf. Section 4.1.1). There are three existing BOUND implementations: one for the interval abstract domain [CC76], one for the polyhedra abstract domain [CH78] and one for the octagons abstract domain [Min06]. We have implemented two new bounds: the first, called *HeapBound*, is the heap abstraction introduced in Chapter 3. The second, called *CombinedBound*, is the combined bound briefly described in Section 4.3. The bounds also contain the correspondence between the constraints and the bound.

Function In FuncTion the function auxiliary domains are implemented using the *FUNCTION* interface. FUNCTION implementations represent the *value* of a piecewise-defined ranking function. There are two existing FUNCTION implementations: one for affine functions over numerical variables and ordinal valued functions, which we have not discussed in this thesis. We have implemented two new functions: the first, called *HeapAffines*, is the functions auxiliary domain described in Section 4.2.1. The second, called *CombinedAffines*, is a combination of the functions auxiliary domain for values and the functions auxiliary domain for heaps briefly described in Section 4.3.

Domain In FuncTion abstractions of the termination semantics (cf. Section 2.4) are implemented using the *DOMAIN* interface. DOMAIN implementations represent *piecewise-*

defined ranking functions. The value decision trees is the only existing domain implementation (cf. Section 4.1.1). We have implemented two new domains: the first, called *HeapDomain*, is the heap decision tree domain described in Section 4.2.1. The second, called *CombinedDomain*, is the combined decision tree domain briefly described in Section 4.3.

5.2 Experimental Evaluation

We have evaluated the implementation in FuncTion using artificial programs as well as small real world programs such as list traversal, sum of list values, tree traversal and the Josephus problem. In the following we will present the most interesting programs showing the benefits of our implementation as well as its limitations.

In the following we always use the combined decision trees of Section 4.3 parametrized by the interval abstraction. We will use the same representation of the decision trees used in the detailed analyses (cf. Section 4.2.8 and Section 4.3.1). To reduce clutter we will not show the value abstraction if a program has no integer variables. We only define *Var*, *Field* and which field $f \in Field$ was used for the analysis if it is not clear from the context.

A summary of the analysis results is shown in Table 5.1. In the column **Precision** we use *All* to denote that termination could be proven for all terminating states, *Few* to denote that termination could be proven only for a small finite subset of the terminating states and *None* to denote that termination could not be proven for any states. The column **Time** shows the average time the analysis took. The experiments were performed on a system with a 2.4 GHz 8-Core CPU (Intel i7-6700) with 16GB of RAM, and running Ubuntu 16.04.

Example	Description	Precision	Time
5.2.1	List traversal	All	0.036s
5.2.2	List traversal (two steps)	Few	0.160s
5.2.3	Unsatisfiable loop condition	All	0.004s
5.2.4	Random choice	All	0.028s
5.2.5	Sum of list values	All	4.376s
5.2.6	Creation of acyclic list & traversal	All	0.472s
5.2.7	Josephus Problem (extended)	Few	0.904s
5.2.8	Josephus Problem	All	0.944s
5.2.9	Heap value	None	0.004s
5.2.10	Tree traversal for field *left*	None	0.007s
	Tree traversal for field *right*	None	0.006s

Table 5.1: Summary of the analysis results.

Note that for Example 5.2.10 we have two entries since the analysis was run for two fields. In the following we will discuss each example in detail.

Example 5.2.1. Let us again consider the list traversal shown in Example 1.0.1 and in the detailed analysis in Example 4.2.6:

$$\begin{array}{l}\texttt{while}\ {}^{1}(l \neq \mathit{null})\ \texttt{do}\\ \quad {}^{2}l := l.\mathit{next}\\ \texttt{od}^{3}\end{array}$$

The result of the analysis gives us the following decision tree at the initial control point:

$$\begin{aligned}(\{l \mapsto ([0,0],[0,0])\},\emptyset)\ ?\ \lambda a_l, c_l.\ 1\\ (\{l \mapsto ([1,1],[0,0])\},\emptyset)\ ?\ \lambda a_l, c_l.\ 3\\ (\{l \mapsto ([2,2],[0,0])\},\emptyset)\ ?\ \lambda a_l, c_l.\ 5\\ (\{l \mapsto ([3,\infty],[0,0])\},\emptyset)\ ?\ \lambda a_l, c_l.\ 2a_l + 1\\ (\{l \mapsto ([0,0],[1,\infty])\},\emptyset)\ ?\ \bot_{\mathcal{F}}\\ (\{l \mapsto ([1,1],[1,\infty])\},\emptyset)\ ?\ \bot_{\mathcal{F}}\\ (\{l \mapsto ([2,2],[1,\infty])\},\emptyset)\ ?\ \bot_{\mathcal{F}}\\ (\{l \mapsto ([3,\infty],[1,\infty])\},\emptyset)\ ?\ \bot_{\mathcal{F}}\end{aligned}$$

The analysis returns a decision tree representing the ranking function described in the introduction in Example 1.0.1 and therefore is able to find all terminating states.

Example 5.2.2. Let us consider the following program:

```
while [1](l ≠ null) do
  [2]l := l.next
  [3]l := l.next
od[4]
```

The program loops over a list by following the field *next* twice each iteration. The result of the analysis gives us the following decision tree at the initial control point:

$$
\begin{aligned}
(\{l \mapsto ([0,0],[0,0])\},\emptyset) \ ?\ & \lambda a_l, c_l.\ 1\\
(\{l \mapsto ([1,1],[0,0])\},\emptyset) \ ?\ & \bot_{\mathcal{F}}\\
(\{l \mapsto ([2,2],[0,0])\},\emptyset) \ ?\ & \lambda a_l, c_l.\ 4\\
(\{l \mapsto ([3,3],[0,0])\},\emptyset) \ ?\ & \bot_{\mathcal{F}}\\
(\{l \mapsto ([4,4],[0,0])\},\emptyset) \ ?\ & \lambda a_l, c_l.\ 7\\
(\{l \mapsto ([5,\infty],[0,0])\},\emptyset) \ ?\ & \top_{\mathcal{F}}\\
(\{l \mapsto ([0,0],[1,\infty])\},\emptyset) \ ?\ & \bot_{\mathcal{F}}\\
(\{l \mapsto ([1,1],[1,\infty])\},\emptyset) \ ?\ & \bot_{\mathcal{F}}\\
(\{l \mapsto ([2,2],[1,\infty])\},\emptyset) \ ?\ & \bot_{\mathcal{F}}\\
(\{l \mapsto ([3,3],[1,\infty])\},\emptyset) \ ?\ & \bot_{\mathcal{F}}\\
(\{l \mapsto ([4,4],[1,\infty])\},\emptyset) \ ?\ & \bot_{\mathcal{F}}\\
(\{l \mapsto ([5,\infty],[1,\infty])\},\emptyset) \ ?\ & \bot_{\mathcal{F}}
\end{aligned}
$$

The analysis is able to prove termination if l is an acyclic list of length zero, two, or four. However it is not possible to represent all terminating states in a summarized form using the heap abstract domain since it cannot represent the fact that the acyclic length must be even. Note that $\top_{\mathcal{F}}$ for the partition $(\{l \mapsto ([5,\infty],[0,0])\},\emptyset)$ is caused by the widening trying to guess a ranking function and, after failing to do so, marking the partition with $\top_{\mathcal{F}}$ so it is ignored in further iterations.

Example 5.2.3. Let us consider the following program:

```
while [1](l = null and l = l.next) do
  [2]l := l.next
od[3]
```

The result of the analysis returns the following decision tree at the initial control point:

$$\top_{\mathcal{D}} \ ? \ \lambda a_l, c_l.\ 1$$

The program terminates in one step for any input which is correct since it is impossible for both $l = null$ and $l = l.next$ to be satisfied by any heap configuration, which means that the loop body never gets executed.

Example 5.2.4. Let us consider the following program:

```
while [1](l ≠ null) do
  if [2]? then
    [3]l := l.next
  else
    [4]skip
  fi
od[5]
```

Note that in the body of the loop either l is updated to $l.next$ or `skip` is executed, chosen at random. The result of the analysis gives us the following decision tree at the initial control point:

$$\begin{aligned}
(\{l \mapsto ([0,0],[0,0])\},\emptyset) \ ? \ \lambda a_l, c_l.\ 1 \\
(\{l \mapsto ([1,\infty],[0,0])\},\emptyset) \ ? \ \bot_{\mathcal{F}} \\
(\{l \mapsto ([0,0],[1,\infty])\},\emptyset) \ ? \ \bot_{\mathcal{F}} \\
(\{l \mapsto ([1,\infty],[1,\infty])\},\emptyset) \ ? \ \bot_{\mathcal{F}}
\end{aligned}$$

The program terminates in one step if l is *null* and it is unknown whether the program terminates for any other input. Indeed, l being *null*, is the only terminating state since it is possible that every iteration of the loop executes `skip`, which means that any state where l is not *null* might not terminate.

Example 5.2.5. Let us consider the following program:

$$
\begin{array}{l}
{\scriptstyle 1}s := 0 \\
\texttt{while } {\scriptstyle 2}(l \neq \mathit{null} \texttt{ and } r \neq \mathit{null}) \texttt{ do} \\
\quad {\scriptstyle 3}s := l.\mathit{value} + t.\mathit{value} \\
\quad {\scriptstyle 4}l := l.\mathit{next} \\
\quad {\scriptstyle 5}r := r.\mathit{next} \\
\texttt{od}{\scriptstyle 6}
\end{array}
$$

Note that we have $\mathit{Num} \stackrel{\text{def}}{=} \{s\}$, $\mathit{Ref} \stackrel{\text{def}}{=} \{l, r\}$ and $\mathit{Field} \stackrel{\text{def}}{=} \{\mathit{value}, \mathit{next}\}$. The program creates a sum of the values of the two lists pointed to by l and r as long as both l and r are not *null*. Similar to the list traversal (cf. Example 5.2.1) the program does not terminate if both l and r point to a cycle. The result of the analysis gives us more than 50 partitions. In all partitions s can be any value. To save space we will only show the heap abstraction. In the following we show some of the interesting partitions. For all partitions where both l or r point to a cyclic list the value of the ranking function is undefined:

$$
\begin{array}{r}
(\{l \to ([0,0],[1,\infty]), r \to ([0,0],[1,\infty])\}, \{(l,r) \to \top_{\mathcal{R}}, (r,l) \to \top_{\mathcal{R}}\}) \ ? \ \bot_{\mathcal{F}} \\
(\{l \to ([1,1],[1,\infty]), r \to ([0,0],[1,\infty])\}, \{(l,r) \to \top_{\mathcal{R}}, (r,l) \to \top_{\mathcal{R}}\}) \ ? \ \bot_{\mathcal{F}} \\
(\{l \to ([2,2],[1,\infty]), r \to ([0,0],[1,\infty])\}, \{(l,r) \to \top_{\mathcal{R}}, (r,l) \to \top_{\mathcal{R}}\}) \ ? \ \bot_{\mathcal{F}} \\
(\{l \to ([3,\infty],[1,\infty]), r \to ([0,0],[1,\infty])\}, \{(l,r) \to \top_{\mathcal{R}}, (r,l) \to \top_{\mathcal{R}}\}) \ ? \ \bot_{\mathcal{F}} \\
\ldots
\end{array}
$$

For all other partitions the analysis is able to prove termination:

$$
\begin{array}{lr}
(\{l \to ([0,\infty],[0,\infty]), r \to ([0,0],[0,0])\}, \{(l,r) \to \top_{\mathcal{R}}, (r,l) \to \top_{\mathcal{R}}\}) \ ? & \lambda a_l, c_l, a_r, c_r.\ 2 \\
(\{l \to ([3,\infty],[0,0]), r \to ([0,0],[1,\infty])\}, \{(l,r) \to \top_{\mathcal{R}}, (r,l) \to \top_{\mathcal{R}}\}) \ ? & \lambda a_l, c_l, a_r, c_r. \\
 & \quad 4a_l + 2 \\
(\{l \to ([3,\infty],[0,0]), r \to ([3,\infty],[0,0])\}, \{(l,r) \to \top_{\mathcal{R}}, (r,l) \to \top_{\mathcal{R}}\}) \ ? & \lambda a_l, c_l, a_r, c_r. \\
 & \quad 4a_l + 4a_r - 10 \\
 & \ldots
\end{array}
$$

Example 5.2.6. Let us consider the following program:

```
while [1](i < 10) do
  [2]i := i + 1
  [3]t := new
  [4]t.next := l
  [5]l := t
od[6]
while [7](l ≠ null) do
  [8]l := l.next
od[9]
```

Note that we have $Num \stackrel{\text{def}}{=} \{i\}$ and $Ref \stackrel{\text{def}}{=} \{l, t\}$. The first loop adds $i + 10$ elements to the list pointed to by l and the second loop traverses the list. The program terminates for all states where l initially is acyclic. For all partitions where l has a cycle the value of the ranking function is undefined. For all other partitions the analysis is able to prove termination:

$$\{i \geq 10\}, (\{l \to ([3, \infty], [0, 0]), t \to ([0, \infty], [0, \infty]\}, \{(l, t) \to \top_{\mathcal{R}}, (t, l) \to \top_{\mathcal{R}}\}) \; ?$$
$$\lambda a_l, c_l, a_t, c_t, i.\; 2a_l + 2$$
$$\{i < 8\}, (\{l \to ([3, \infty], [0, 0]), t \to ([0, \infty], [0, \infty]\}, \{(l, t) \to \top_{\mathcal{R}}, (t, l) \to \top_{\mathcal{R}}\}) \; ?$$
$$\lambda a_l, c_l, a_t, c_t, i.\; -7i + 2a_l + 72$$
$$\{i = 9\}, (\{l \to ([1, 1], [0, 0]), t \to ([0, \infty], [0, \infty]\}, \{(l, t) \to \top_{\mathcal{R}}, (t, l) \to \top_{\mathcal{R}}\}) \; ?$$
$$\lambda a_l, c_l, a_t, c_t, i.\; 11$$
$$\ldots$$

By adding assignments $i := 9$ and $l := new$ at the beginning of the program we select the last partition and the analysis returns a decision tree consisting of a single node at the initial control point:

$$\{i \to \top_{\mathcal{I}}\}, (\{l \to ([0, \infty], [0, 0]), t \to ([0, \infty], [0, \infty]\}, \{(l, t) \to \top_{\mathcal{R}}, (t, l) \to \top_{\mathcal{R}}\}) \; ?$$
$$\lambda a_l, c_l, a_t, c_t, i.\; 13$$

which proves that the program terminates for any input. Note that the value is increased by two because of the two additional statements.

Example 5.2.7. Let us consider the following extended version of Josephus Problem which satisfies our simple grammar (cf. Section 2.1) and the assumptions we made for the transformers of the heap analysis (cf. Chapter 3).

$$
\begin{array}{l}
\texttt{while}\ {}^{1}(l \neq \mathit{null})\ \texttt{do} \\
\quad {}^{2}p := l.\mathit{next} \\
\quad {}^{3}q := p.\mathit{next} \\
\quad {}^{4}l.\mathit{next} := q \\
\quad {}^{5}l := l.\mathit{next} \\
\texttt{od}^{6}
\end{array}
$$

Note that we have $\mathit{Ref} = \{l, p, q\}$ where p and q are used as temporaries during the execution of the loop. As discussed in Example 1.0.2 the program terminates for lists that end in a cycle. The result of the analysis gives us the following decision tree at the initial control point:
1 :

$$
\begin{array}{l}
(\{l \mapsto ([0,0],[1,1]), p \mapsto ([0,\infty],[0,\infty]), q \mapsto ([0,\infty],[0,\infty])\}, \\
\qquad\qquad \{(l,p) \to \top_{\mathcal{R}}, \ldots, (q,p) \to \top_{\mathcal{R}}\})\ ?\ \lambda a_l, c_l, a_p, c_p, a_q, c_q.\ 1 \\
(\{l \mapsto ([2,2],[1,1]), p \mapsto ([0,\infty],[0,\infty]), q \mapsto ([0,\infty],[0,\infty])\}, \\
\qquad\qquad \{(l,p) \to \top_{\mathcal{R}}, \ldots, (q,p) \to \top_{\mathcal{R}}\})\ ?\ \lambda a_l, c_l, a_p, c_p, a_q, c_q.\ 6 \\
(\{l \mapsto ([4,4],[1,1]), p \mapsto ([0,\infty],[0,\infty]), q \mapsto ([0,\infty],[0,\infty])\}, \\
\qquad\qquad \{(l,p) \to \top_{\mathcal{R}}, \ldots, (q,p) \to \top_{\mathcal{R}}\})\ ?\ \lambda a_l, c_l, a_p, c_p, a_q, c_q.\ 11 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \ldots
\end{array}
$$

The analysis is only able to prove termination for the cases where l points a list with a single cycle at the end and either no acyclic part, an acyclic part of length two or an acyclic part of length four. The problem is that the transformer for the statement $l.\mathit{next} := q$ is very imprecise for the cyclic case (cf. Equation 3.22). Improvements for the heap abstraction which might help are discussed in Section 6.1.

Example 5.2.8. As we have seen in Example 5.2.7 using the simplified grammar means the analysis only finds three terminating states of the Josephus Problem. Since the statement $p.next := p.next.next$ makes it easier to define a precise transformer than the general case $p.next := q$ we have implemented this specific transformer in FuncTion. Let us consider the following version of the Josephus Problem:

$$\begin{array}{l}\texttt{while}\ {}^{1}(l \neq \mathit{null})\ \texttt{do}\\ \quad {}^{2}l.\mathit{next} := l.\mathit{next}.\mathit{next}\\ \quad {}^{2}l := l.\mathit{next}\\ \texttt{od}^{3}\end{array}$$

For all partitions where l does not have a cycle of the resulting ranking function is undefined (which is correct, cf. Example 1.0.2). For all other partitions the analysis is able to prove termination:

$$\begin{array}{c}(\{l \mapsto ([5,\infty],[1,1])\},\emptyset)\ ?\ \lambda a_l, c_l.\ 3a_l - 5\\ (\{l \mapsto ([5,\infty],[2,2])\},\emptyset)\ ?\ \lambda a_l, c_l.\ 3a_l - 2\\ (\{l \mapsto ([5,\infty],[3,\infty])\},\emptyset)\ ?\ \lambda a_l, c_l.\ 3a_l + 3a_c - 8\\ \ldots\end{array}$$

Although the programs of Example 5.2.7 and Example 5.2.8 are equivalent, having less temporary variables makes the analysis more precise. We briefly discuss this as possible future work in Section 6.1.

Example 5.2.9. Let us consider the following program:

$$\begin{array}{l}\texttt{while}\ {}^{1}(l.\mathit{value} \leq 10)\ \texttt{do}\\ \quad {}^{2}l := l.\mathit{next}\\ \texttt{od}^{3}\end{array}$$

Note that we have $\mathit{Field} = \{\mathit{next}, \mathit{value}\}$ and that value is a numerical field. The result of the analysis gives us the following decision tree at the initial control point:

$$\top_{\mathcal{D}}\ ?\ \bot_{\mathcal{F}}$$

The analysis is unable to prove termination for any state since the heap abstraction does not abstract values in the heap.

Example 5.2.10. Let us consider the following program:

```
while [1](t ≠ null) do
  if [2]t.left ≠ null? then
    [3]t := t.left
  else
    [3]t := t.right
  fi
od[5]
```

Note that we have $Field = \{left, right\}$ and that both *left* and *right* are references. The program traverses a tree like structure preferring the left subtree. It terminates if the path, following *left* if possible and otherwise *right*, has no cycle. The result of the analysis for both the *left* and the *right* fields gives us the following decision tree at the initial control point:

$$\begin{aligned}(\{t \mapsto ([0,0],[0,0])\},\emptyset) \ ?\ \lambda a_l, c_l.\ 1\\ (\{t \mapsto ([1,\infty],[0,0])\},\emptyset) \ ?\ \bot_{\mathcal{F}}\\ (\{t \mapsto ([0,0],[1,\infty])\},\emptyset) \ ?\ \bot_{\mathcal{F}}\\ (\{t \mapsto ([1,\infty],[1,\infty])\},\emptyset) \ ?\ \bot_{\mathcal{F}}\end{aligned}$$

The analysis is unable to prove termination for any state, except *null*, since the heap abstraction is very imprecise when multiple reference fields are involved.

Other tools such as AProVE [SAF+15] and Ultimate Automizer [ea15] are more advanced, as we can see on the results of the yearly termination competition[1], but are, to our knowledge, not able to do conditional termination proofs. We were not able to directly compare the results of our analysis with either tool since in our language an uninitialized reference can have any value, while in the *C* category of the termination competition an uninitialized reference leads to undefined behavior, which has to be reported [2], and in Java all references are initialized (fields) or have to be initialized (local variables). This means that almost all of our examples can not be directly translated to *C* or Java. Most of the heap-manipulating programs of the termination competition use features such as pointer arithmetic or constructors which are not supported in our simple language (cf. Section 2.1). Still, some of our examples are inspired by existing programs used in the termination competition.

[1] http://termination-portal.org/wiki/Termination_Competition
[2] http://termination-portal.org/wiki/C_Programs

6 Conclusion

In this thesis, we have presented a new heap abstract domain which is used to extend FuncTion's [Urb15a] termination abstract domain to prove termination of heap manipulating programs by abstract interpretation.

Our first contribution is a novel heap abstract domain (cf. Chapter 3). It is parametric in the choice of a field and is used to abstract length and reachability information of paths, only using the chosen field, in the heap. The abstraction is designed to be able to abstract linked lists.

Our second contribution is the extension of the existing decision tree domain proposed in [Urb15b] to use the heap abstract domain (cf. Section 4.2). The result is an abstract domain which is able to infer piecewise-defined ranking functions for programs manipulating the heap. The experimental evaluation shows that the analysis is precise for programs with little or no field updates. It is however unable to prove termination for programs with multiple references and field updates, especially when cyclic lists are involved.

Our last contribution is a combination of the existing decision tree domain, based on numerical abstractions, and the newly developed termination abstraction for heap manipulating programs (cf. Section 4.3). The result is an abstract domain that is able to prove termination for programs not only manipulating lists but also values.

6.1 Future Work

In this section, we discuss possible future work related to this project.

Forward Analysis By defining transformers for forward assignments and a widening for the heap abstract domain we could use a forward analysis to refine the backward analysis, as is already done for the value decision trees in FuncTion. This could improve the precision since some transformers might be easier to define in a forwards manner.

Syntax Assumptions The assumptions on the syntax, i.e., that we do not have nested dereference expressions such as $p.f.f = q$, make the analysis of some programs less precise as we have seen in Example 5.2.7 and Example 5.2.8. Lifting the assumption means that the fixed partitioning, defined in Section 4.2.2, is not enough to be able to precisely represent filter statements such as $p.f.f \neq \mathit{null}$. The partitioning should be

dynamic or ideally the heap abstract domain itself should be partitioned already. We have implemented the transformer for the assignment $p.f = p.f.f$, used in Example 5.2.8, but more work is still required.

Reachability using Bounds The current heap abstract domain only distinguishes between reachable and not reachable references. By using reachable or not reachable in a number of steps we might be able to improve the precision of the transformer for the statement $p.f = q$.

Values in the Heap Another possible improvement would be to keep track of values in the heap, which is needed for examples such as Example 5.2.9. A simple implementation could track intervals for each reference, representing the possible values reachable by that reference.

6.2 Acknowledgements

Special thanks to my advisor Dr. Caterina Urban for introducing me to termination analysis by abstract interpretation, for many helpful discussions and especially for the feedback on the written report. I would also like to thank Prof. Dr. Peter Müller for giving me the opportunity to work on this interesting thesis. Last but not least, thanks to my family for their patience and support.

Bibliography

[CC76] Patrick Cousot and Radhia Cousot. Static Determination of Dynamic Properties of Programs. In *Symposium on Programming*, pages 106–130, 1976.

[CC77] Patrick Cousot and Radhia Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*, pages 238–252, 1977.

[CC92] Patrick Cousot and Radhia Cousot. Abstract Interpretation Frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[CC12] 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*, pages 245–258, Philadelphia, PA, January 25-27 2012. ACM Press, New York.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *POPL*, pages 84–96, 1978.

[Cou78] Patrick Cousot. *Méthodes Itératives de Construction et d'Approximation de Points Fixes d'Opérateurs Monotones sur un Treillis, Analyse Sémantique de Programmes*. Thèse d'État Ès Sciences Mathématiques, Université Joseph Fourier, Grenoble, France, 1978.

[ea15] Matthias Heizmann et. al. Ultimate automizer with smtinterpol. In *TACAS*, pages 641–643, 2015.

[Min06] Antoine Miné. The Octagon Abstract Domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[SAF+15] Thomas Ströder, Cornelius Aschermann, Florian Frohn, Jera Hensel, and Jürgen Giesl. Aprove: Termination and memory safety of c programs. In *TACAS*, pages 417–419, 2015.

[Tur36] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

[Tur49] Alan Turing. Checking a Large Routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, 1949.

[Urb15a] Caterina Urban. FuncTion: An abstract domain functor for termination (competition contribution). In *TACAS*, pages 464–466, 2015.

[Urb15b] Caterina Urban. *Static Analysis by Abstract Interpretation of Functional Temporal Properties of Programs.* PhD thesis, Ecole Normale Superieure, 2015.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Termination Analysis of Heap-Manipulating Programs by Abstract Interpretation

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Neukom	Lukas

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Brugg, 28.09.2016	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.