QUEEN MARY UNIVERSITY OF LONDON

Proving Termination
using
Abstract Interpretation

by

Aziem A. Chawdhary

A thesis submitted in partial fulfillment for the
degree of Doctor of Philosophy
in the
School of Electronic Engineering and Compter Science

October 2010

Declaration of Authorship

I, Aziem Chawdhary, declare that this thesis titled, 'Proving Termination using Abstract Interpretation' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

"Seek knowledge from the cradle to the grave."

Attributed to the Prophet Muhammad (pbuh).

Abstract

One way to develop more robust software is to use formal program verification. Formal program verification requires the construction of a formal mathematical proof of the programs correctness. In the past ten years or so there has been much progress in the use of automated tools to formally prove properties of programs. However many such tools focus on proving safety properties: that something bad does not happen. Liveness properties, where we try to prove that something good will happen, have received much less attention. Program termination is an example of a liveness property. It has been known for a long time that to prove program termination we need to discover some function which maps program states to a well-founded set. Essentially we need to find one global argument for why the program terminates. Finding such an argument which overapproximates the entire program is very difficult. Recently, Podelski and Rybalchenko discovered a more compositional proof rule to find disjunctive termination arguments. Disjunctive termination arguments requires a series of termination arguments that individually may only cover part of the program but when put together give a reason for why the entire program will terminate. Thus we do not need to search for one overall reason for termination but we can break the problem down and focus on smaller parts of the program.

This thesis develops a series of abstract interpreters for proving the termination of imperative programs. We make three contributions, each of which makes use of the Podelski-Rybalchenko result.

Firstly we present a technique to re-use domains and operators from abstract interpreters for safety properties to produce termination analysers. This technique produces some very fast termination analysers, but is limited by the underlying safety domain used.

We next take the natural step forward: we design an abstract domain for termination. This abstract domain is built from ranking functions: in essence the abstract domain only keeps track of the information necessary to prove program termination. However, the abstract domain is limited to proving termination for language with iteration.

In order to handle recursion we use metric spaces to design an abstract domain which can handle recursion over the unit type. We define a framework for designing abstract interpreters for liveness properties such as termination. The use of metric spaces allows us to model the semantics of infinite computations for programs with recursion over the unit type so that we can design

an abstract interpreter in a systematic manner. We have to ensure that the abstract interpreter is well-behaved with respect to the metric space semantics, and our framework gives a way to do this.

Acknowledgements

There is a famous saying amongst the Arabs: “One who has not thanked people, has not thanked God”. There are many people I would like to thank, both professionally and personally.

Firstly I would like to thank Prof. Richard Bornat and Prof. Peter O’Hearn: it was ultimately because of them that I ended up writing this thesis. They very kindly took me under their wing when I was an undergraduate and introduced me to the wonderful world of logic and program verification. They have both encouraged and guided me since then. I admire the enthusiasm they both have for computer science and programming in particular.

Secondly I would like to thank Dr. Hongseok Yang. He has in the course of my doctoral studies become my main supervisor. He has been incredibly patient in guiding me through the intricacies of program analysis and abstract interpretation in particular. Without his knowledge, effort and helping hand I would never have completed this thesis.

I would also like to thank the rest of the East London Massive and the wider Theory group at Queen Mary. They have made my time at Queen Mary very enjoyable with wide and varied discussions on many topic, particularly over lunch in various eateries in Mile End. I have worked with many members of staff at Queen Mary whilst I have been here and I have enjoyed every minute of it.

I would like to thank Microsoft Research, whose PhD Scholarship supported my doctoral studies, and also Prof. Byron Cook, who was my supervisor at Microsoft Research.

On a more personal note I would like to thank The Crew, whom I met whilst at Imperial College. Five years on we are still enjoying many colourful discussions and arguments, usually whilst eating good and bad food in one of the many restaurants on Edgeware Road.

My long suffering wife, Elizabeth, has put up with me writing and working on this thesis. She has been patient and supportive whilst completion dates have continually slipped.

Lastly we come to my parents. My parents have provided for me and given me a stable and comfortable home life. The have loved me and supported me every step of the way. Without their guidance and love I would not be in the fortunate position that I am in today.

Contents

List of Figures

For Elizabeth

Chapter 1

Introduction

Building correct and reliable software systems is extremely difficult. The lack of sound engineering principles for the design and construction of software is a major problem. Bugs in software can have significant impact in the modern world.

One possible way for dealing with the unreliability of software is formal program verification. Formal program verification tries to produce a formal, mathematical proof of a program's correctness. Ideally we would like to find proofs for full functional correctness: a proof that the program does what is intended. We would also like to do this automatically. There has been much work in automatic program verification in the last thirty years. However in the past decade or so there have been a number of significant advances [5, 8, 38] - so much so, that automatic program verification is beginning to have an impact in real-world software development.

Static analysis is one piece in the formal program verification jigsaw that has made significant impact in recent years [5, 25, 37, 38]. A static analysis attempts to discover various properties of a program without executing the program. These properties can be classed into two main categories: safety properties and liveness properties [1, 27, 40]. A safety property states that "something bad will not happen". Safety properties can be expressed as a reachability question: can this program reach an error memory state from some initial starting state? Examples of safety properties include accessing memory which has been de-allocated or division by zero. There has been much work on automatically proving safety properties.

A liveness property, on the other hand, states that "something good will eventually happen". This is a much more difficult property to check: a liveness property is a set of potentially infinite traces whose falsity cannot be witnessed by finite traces, whereas safety properties can be falsified by finite traces alone [1]. Most previous liveness analyses have focused on finite state systems: model checking logics such as CTL or LTL for example [16]. But for infinite state systems, such as computer software, there has not been much progress.

This thesis develops static analyses for termination, a specific liveness property. In the past many researchers viewed program termination as an impossible problem mainly due to Turing's proof of undecidability of the Halting Problem. However, if one only asks for a sound but not necessarily complete solution then the halting problem is not a barrier to a sound termination analyser: when the analyser returns a positive answer it means that the program terminates. However if the analyser returns a negative answer that means the program could or could not terminate.

Proving safety properties is undecidable but recent work has shown that we can have useful safety analysers. However to analyse infinite state systems such as software, safety analyses have to make use of abstraction techniques. This use of abstraction means that most practical safety analyses are sound but not complete. This successful use of abstraction is safety analyses shows that similar ideas could be used for liveness analyses for infinite state systems.

It has been known for a long time that one way of proving that a program terminates is to a measure on the program which constantly decreases on each iteration through the program. More formally one has to find a mapping from program states into some well-founded set [56]. Essentially one needs to find some global termination argument which covers the entire program.

A breakthrough was made by Podelski and Rybalchenko [47, 49] who developed a proof rule to help find *disjunctive* termination arguments: smaller measures which cover part of the program, but put together they give a reason for why the entire program terminates. Instead of searching for a global termination argument which overapproximates the entire program, the Podelski-Rybalchenko results allows us to search for smaller termination arguments which individually overapproximate part of a program but collectively they overapproximate the entire program.

The Podelski-Rybalchenko result fits naturally with path-sensitive static analyses, which is a requirement for proving termination. It also fits in naturally with counter-example guided abstraction refinement (CEGAR [15]), a successful technique in static analysis of infinite state systems. CEGAR works by iteratively checking to see if the property to be checked holds for a given program. If the answer is no, then a counterexample to the property is extracted and then analysed to see if it really is a counterexample or whether extra facts need to be added to the proof in order to remove the counterexample. This process continues until all spurious counterexamples are removed or there is a counterexample which really does show that property does not hold for a given program.

TERMINATOR [19, 20], a tool produced at Microsoft Research, was the first practical termination analyser to make use of the Podelski-Rybalchenko rule. TERMINATOR uses counter example guided abstraction refinement (CEGAR) to produce termination arguments and was used very successfully in proving termination of low-level systems code in Windows device

drivers. However, TERMINATOR like many CEGAR based approaches, has to check the validity of termination arguments it produces, which is a computationally expensive task.

This difficulty was the initial motivation for using abstract interpretation [23] to define termination analysers. Abstract interpretation is a framework for defining sound and valid static analyses. Thus using abstract interpretation we do not need to check the validity of any termination arguments the abstract interpreter produces: they are valid by construction. In this thesis we chart the progression of a series of works defining abstract interpreters for termination properties. All these abstract interpreters make use of the Podelski-Rybalchenko result.

In this thesis we will be defining a number of abstract interpreters for program termination. Each of the abstract interpreters is built on the Podelski-Rybalchenko result. We will design abstract domains specifically to prove program termination. It should be noted that when the analyses in this thesis fail to prove termination, this does not imply non-termination of the program. In order to see if the program is indeed non-terminating one would need to manually inspect the code to see if this is indeed the case or use a specific non-termination analyser [36].

1.1 Outline

- We will first present some background material on automatically proving program termination. In particular we will present the Podelski-Rybalchenko result in Section 2.1.3, and then briefly explain how TERMINATOR works in Section 2.2. We will then present work [7] that converts abstract interpreters for safety properties into termination analysers in Section 3. [1]

  Although the termination analysers produced using this approach were extremely fast there were a number of troublesome issues:

  1. Reusing abstract operations designed to find safety properties could result in the termination analyser having to compute information not relevant to termination. Also since many of the abstract domains are infinite, in order to guarantee that the analyser terminates they make use of acceleration techniques to ensure the analyser terminates. However these acceleration techniques are designed with safety properties in mind, and making use of them for termination analysis often results in crucial information being lost.

[1] The author worked on the implementation but not on the theoretical work. This is why we have put this in the background section

2. Many safety abstract interpreters are also designed to be path insensitive, that is they do not keep explicit information for each path through the program. Since termination requires that every path through the program terminates to get a useful termination analyser we require that the analysis maintains information about each path explicitly: we require the analysis to be *path sensitive*. Many safety analyses do not maintain such information, but merge information from many paths in order to be computationally efficient. There is no canonical answer regarding the best approach to maintaining such path sensitive information . There have been heuristics to try to extract path sensitive assertions from analyses which do not maintain such information [42, 52], but performing tractable and useful analyses with path sensitivity is still an open research problem.

- In Chapter 4 we address the concerns from the previous approach. We start from scratch: we design abstract domains specifically for program termination. Since we need to find well-founded relations to prove program termination, the abstract domain consists of assertions that overapproximate a well-founded relation. In essence we only keep the information necessary to prove program termination and we disregard any information that is irrelevant to finding a termination argument. We maintain disjunctions but do not need acceleration techniques to ensure we get an answer, avoiding a problem with the approach presented in Section 3. We provide a framework for designing abstract domains based for proving termination and provide an instance. We then compare the implementation to existing termination provers.

  One issue with this approach is that the soundness proof for relating a concrete trace semantics with an abstract domain for termination is non-trivial. The solution in this chapter has to relate greatest fixpoints in the concrete trace semantics with least fixpoints in the abstract interpreter. This is highly specific and non-standard and the proof is built on tricks that are non-extensible. As a consequence we can only provide abstract interpreters for languages with iteration and not recursion.

- In order to find a more elegant soundness proof we turn to metric spaces in Chapter 5. We make novel use of metric spaces in the concrete semantics and relate this semantics to abstract domains based on ranking functions.

  The use of metric spaces allows us to use Banach's unique fixpoint theorem [54]: a contractive function on a complete metric space has a unique fixpoint. In comparison to the previous approach we now need to relate a unique fixpoint in the concrete semantics with a least fixpoint in the abstract interpreter; we do not have to try to relate greatest fixpoints to least fixpoints.

However the use of metric spaces means that we have to design the abstract interpreter carefully to ensure that it is sound. We again provide a framework and an instance before showing a number of example analyses of programs.

The metric space used is based on sets of traces and follows previous work on metric spaces for programming language semantics [13, 57]. The concrete semantics is then defined as contractive functions on this metric space which gives us a unique fixpoint. An important point is that since we are using Banach's Fixpoint theorem, the fixpoint computation for loops and recursive functions can start from anywhere in the metric space: we do not require a distinct 'bottom' element as is the usual case in standard abstract interpretation theory.

Since we require a complete metric space in order to apply Banach's Fixpoint Theorem, we cannot use the standard powerset of traces. This is because the standard powerset of traces is not a complete metric space. So we have to work in a restricted powerset of traces. This means that we have to take care to design the abstract interpreter in order to overapproximate this restricted metric space. The framework defined in Chapter 5 defines conditions on both the concrete and abstract semantics which ensures we are designing sound analysers over the concrete metric semantics.

Each of these chapters provide frameworks from which we can build many termination analysers. Thus in this thesis we are presenting a methodology for designing a termination analyser. We define instances in each of the three frameworks to show that the methods can produce analysers which give meaningful results.

Chapter 2

Technical Background

In this chapter we will present preliminary and background information. In Section 2.1.3 will first define the Podelski-Rybalchenko result. We will then give a brief overview of TERMINATOR in Section 2.2, which was the first termination analyser to make use of the Podelski-Rybalchenko result. Since we are defining abstract interpreters, we will provide a brief overview of abstract interpretation in Section 2.3.

2.1 Preliminaries

2.1.1 Well-Ordered Sets

To prove the termination of a program we need to find some measure that decreases and is bounded. The theory of well-ordered sets is the foundation for proving termination is many practical termination analysers. We will define some basic notions of well-ordered sets.

Definition 2.1 (Total-Order). A set X with an order $\leq$ is a total order iff the order is reflexive, antisymmetric, transitive and total.

Definition 2.2 (Well-Ordered Set). $(X, \leq)$ is a well order iff it is a total order and every non-empty subset of X has a least element.

Mapping into a well-ordered set gives us the mathematical apparatus needed in order to define when a program is terminating.

2.1.2 Ranking Function and Ranking Relations

Traditional methods for proving the termination of a program involved showing that there is a function from the set of program states to some well-ordered set. Such a function is called a *ranking function*. This technique was first presented by Turing [56].

Definition 2.3 (**Ranking Function**). A ranking function is a function f with a range to a well-ordered set.

Example 2.1. *An example of a ranking function is* $f : \mathbf{N} \to \mathbf{N}$ *defined by*

$$f(x) = \begin{cases} 0 & \text{if } x = 0 \\ x - 1 & \text{otherwise} \end{cases}$$

Since $\mathcal{N}$ *is a well-ordered set with the usual* $\leq$ *order,* f *is a ranking function.*

From the ranking function we can define a ranking relation:

Definition 2.4 (**Ranking Relation**). Given a ranking function $f : X \to Y$ we can define f's ranking relation, $rank(f)$ as follows:

$$rank(f) = \{\, (s,t) \mid f(s) > f(t) \,\}$$

Example 2.2. *Suppose that we have states* $\sigma : Var \to \mathbf{N}$. *The assertion below is a relation that represents a program that decrements* x *until* x *is non-positive.*

$$'x > x \wedge x > 0$$

Here the pre-primed variable denotes the value of x *is a previous state and the non-primed variable denotes the current value. The above assertion can be seen as a relation over program states:*

$$\{\, (s,t) \mid s(x) > t(x) \wedge t(x) > 0 \,\}$$

To prove this program terminating we need to find a ranking function whose ranking relation overapproximates the relation above. The ranking function f *defined about in example 2.1 is one such function.*

Rank function synthesis is an algorithm that given a relation attempts to find a ranking function which overapproximates it.

Definition 2.5 (Rank Function Synthesis). Given a relation R, a rank function synthesis engine tries to find a ranking function f such that:

$$R \subseteq rank(f)$$

Rank function synthesis tries to find a ranking function, whose ranking relation overapproximates R. To see why rank function synthesis is relevant to program termination consider relation R as representing the transition relation of a program. In order to prove that R does not represent infinite computations it is enough to show that some ranking relation overapproximates R. Using rank function synthesis we can find such a ranking relation. Note however that in practice rank function synthesis usually only works for cases where R represents a non-loop program. If the program has complicated control flow then it is difficult to find a ranking relation/function that proves R terminating. However a recent mathematical result has changed this.

In [48], Podelski and Rybalchenko show that rank function synthesis is complete for ranking relations over linear constraints. This result together with the next result is the foundation for the termination analysers we produce in this thesis. We use RFS to denote a rank function synthesis algorithm.

2.1.3 Podelski-Rybalchenko Result

Some recent advances in proving termination have used a result discovered by Podelski and Rybalchenko. The result gives a proof rule for proving that a relation is well-founded.

Theorem 2.6 (Podelski-Rybalchenko Result). *Suppose we have a binary relation $R \subseteq S \times S$. Let $T_1, T_2, ..., T_n$ be a finite set of binary relations $T_i \subseteq S \times S$ such that each T_i is well-founded. Then R is well-founded iff:*

$$R^+ \subseteq T_1 \cup ... \cup T_n$$

The theorem states that to prove a relation R is well-founded, we need to find a finite set of well-founded relations that overapproximate the non-reflexive, transitive closure of R. The proof can be found in [50]. The Podelski-Rybalchenko result means that we no longer have to search for *one* global ranking function for the whole program. We can now reduce the problem to finding a ranking function for each path through the program, a much simpler task. We refer to the set $T_1 \cup ... \cup T_n$ as the *termination argument*.

```
01        while(x > 0 && y > 0) {
02          if (*) {
03             x = x - 1;
04          } else {
05             y = y - 1;
06        }
```

FIGURE 2.1: Example for Podelski-Rybalchenko Result

We now present an example to motivate why the Podelski-Rybalchenko result is useful in figure Fig.2.1 To prove this program terminating prior to the Podelski-Rybalchenko result we need to find a *lexicographic* ranking function that would show that variables x and y are decreasing and bounded by the 0. For the simple example above this is possible to do automatically [11], but for more complicated examples finding such lexicographic ranking functions is difficult. Using the Podelski-Rybalchenko result we just need to find a ranking function for each branch in the example *individually*. Thus looking at the first branch decrementing x, the ranking function is simply x, and for the second branch it is simply y. The key point is that we can look at each branch independently and generate a ranking function that combined together cover all behaviours of the program.

2.2 TERMINATOR

As mentioned previously TERMINATOR [19, 20] was the first practical static analyser to make use of the Podelski-Rybalchenko result. We will briefly explain how the TERMINATOR works. We include this section on TERMINATOR for comparison purposes but we will not be using TERMINATOR within the thesis.

Terminator works by starting with an empty candidate termination argument T_0. It tries to prove termination of the program using this termination argument. If it fails it finds a counterexample for why the proof fails, and using this counterexample it tries to generate a well-founded relation, T_{i+1} which overapproximates it and adds this relation to the set of candidate termination arguments. As TERMINATOR iterates through this process it goes through the following sequence:

$$\begin{aligned} R^+ &\subseteq T_0 \\ &\vdots \\ R^+ &\subseteq T_0 \cdots \cup T_i \\ R^+ &\subseteq T_0 \cdots \cup T_i \cup T_i + 1 \end{aligned}$$

```
01  T := ∅
02  repeat
03      if R+ ⊆ T then
04          report program terminates
05      else
06          r := a binary relation such that r ⊆ R+ but r ⊆ T
08      if r is not well-founded then
09          report not terminating
10      else
11          W := RFS(r)
12          T := T ∪ W
13  end
```

FIGURE 2.2: TERMINATOR algorithm

.

where each T_i is generated by looking at counterexamples generated in the previous iteration, and the subset inclusion check is performed by a safety analyser (SLAM [5]). If the termination arguments T_i are continually generated then TERMINATOR may diverge.

The algorithm is shown in Fig. 2.2 first presented in [19, 20]. TERMINATOR user abstraction refinement to find termination arguments which overapproximate the program. Abstraction refinement [4, 6, 33, 37] works by iteratively finding counterexamples to the property attempting to be proven. The counterexamples are then analysed to see if they are spurious, in which case an assertion is added to the set of properties proven so far to rule out the counterexample. If the counterexample is not spurious then it is a real counterexample to the property, and thus the program does not satisfy the property. This cycle continues until either we get a positive or negative answer, or else the analysers diverges.

Initially in TERMINATOR starts with the empty set as the initial termination argument (line 01). TERMINATOR then performs the following until it finds a termination argument or a counterexample to program termination:

- In line 03 TERMINATOR checks to see if the relation R is covered by the current termination argument T. This step checks to see if the termination argument T is a valid argument: that is it checks to see if T overapproximates the relational meaning of the program. If it does, we have found a termination argument, otherwise TERMINATOR finds a counterexample to termination. (Line 06) This counterexample is a relation which is a valid path in the program but is not covered by the termination argument found so far.
- To see if this path is a terminating path we check to see if it is well-founded (line 08). This can be done using a rank function synthesis engine RFS. If it is not well-founded, we have found a path in the program for which we cannot prove termination and so we report that we have failed (line 09).

- If it is well-founded, we generate a ranking relation (using RFS) which overapproximates the relation r and then add it to our termination argument (lines 11 and 12).

Line 03 is computationally expensive; TERMINATOR is checking that the set of well-founded relations it has found so far does indeed overapproximate the program. We refer to this step as the *inclusion check* of the Podelski-Rybalchenko result. One way to improve the performance of TERMINATOR is to try and remove this inclusion check, which we address in this thesis by using abstract interpretation.

2.3 Abstract Interpretation

Abstract interpretation [23] is a technique to design sound abstract programming language semantics with respect to more concrete semantics. Abstract interpretation has been used very successfully for static analysis of programs. Since abstract interpretation provides a sound and systematic way to design abstract semantics, proving the soundness of static analysis which use abstract interpretation is much more simple: they are sound by construction. In this section we will provide an overview of abstract interpretation and define some of the terminology and concepts which are used in this thesis.

2.3.1 General Overview

A static analysis using abstract interpretation has two main components: a concrete semantics and an abstract semantics. There techniques for deriving an abstract semantics in a systematic manner from the concrete semantics [22], however we will not be making use of such techniques. In this thesis we design a concrete semantics and an abstract semantics and establish a soundness condition between them.

Abstract interpretation is based on the theory of partial orders and monotone functions on these partial orders. We first provide some general definitions before showing a general recipe for defining abstract interpreters. We will then provide an example abstract interpreter for a simple while language which analyses parity information.

2.3.2 Lattices and Partial Orders

Like many programming language semantics, abstract interpretation is based on mathematical structures built on partial orders and functions on these mathematical structures. In this section we will give a brief overview of the mathematics underlying abstract interpretation.

A partial order is a set C with a binary relation $\sqsubseteq$ on C that satisfies:

- Reflexivity: $\forall c \in C : c \sqsubseteq c$
- Transitivity: $\forall c_1, c_2, c_3 \in C : c_1 \sqsubseteq c_2 \wedge c_2 \sqsubseteq c_3 \rightarrow c_1 \sqsubseteq c_3$
- Anti-symmetry: $\forall c_1, c_2 . c_1 \sqsubseteq c_2 \wedge c_2 \sqsubseteq c_1 \rightarrow c_1 = c_2$

Given a set a $X \subseteq C$, we say that $y \in C$ is an upper bound for X if we have $\forall x \in X . x \sqsubseteq y$. Likewise we can define y to be a lower bound of X if $\forall x \in X . y \sqsubseteq x$. A least upper bound, written as $\sqcup X$ is defined as

$$X \sqsubseteq \sqcup X \wedge \forall y \in C . X \sqsubseteq y \rightarrow \sqcup X \sqsubseteq y$$

and the greatest lower bound ,written as $\sqcap$, is defined as:

$$\sqcap X \sqsubseteq X \wedge \forall y \in C . y \sqsubseteq X \rightarrow y \sqsubseteq \sqcap X$$

Definition 2.7 (Lattice)**.** A lattice is a partial order for which least upper bounds and greatest lower bounds exists for all non-empty finite subsets of C. The lattice is called *complete* if least upper bounds and greatest lower bounds exist for all subsets of C. The lattice also has greatest and lowest elements, denoted by $\top$ and $\bot$ respectively, and these as are given by: $\top = \sqcup C$ and $\bot = \sqcap C$.

2.3.3 Functions on Partial Orders

We can define functions on a lattice. A function $f : C \rightarrow C$ on a lattice C is monotone if:

$$\forall x, y \in C . x \sqsubseteq y \rightarrow f(x) \sqsubseteq f(y)$$

Another useful property is continuity. A function is continuous if:

$$\forall X \subseteq C . f(\sqcup X) = \sqcup F(X)$$

The foundation of abstract interpretation is the computation of fixpoints on lattices and partial orders. The concrete and abstract semantics of recursive and looping constructs are defined using fixpoints. In order to have precise results from the abstract interpreter, we usually interpret these constructs using the *least* fixpoint. We begin by defining what a fixpoint is:

Definition 2.8. A fixpoint of a function $F : D \to D$ is an element of the domain $x \in D$ such that $F(x) = x$.

Abstract interpretation makes use of the well-known fixpoint theorem by Tarksi, which gives us a computational way of computing a least fixpoint.

Theorem 2.9 (Tarskis Fixpoint Theorem)**.** *The set of fixpoints of a monotone function F on a complete lattice $(D, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$ is a non-empty, complete lattice ordered by $\sqsubseteq$. The least fixpoint is given by:*

$$lfp(F) = \sqcap\{x \in D \mid F(x) \sqsubseteq x\}$$

The greatest fixpoint is given by:

$$gfp(F) = \sqcup\{x \in D \mid x \sqsubseteq F(x)\}$$

One way of computing the least fixpoint is to start iterating F from the least element in the abstract domain until we reach a fixpoint:

$$F(\bot), F(F(\bot)), F^3(\bot), ...$$

Since the order on the abstract domain reflects the precision between elements, the least fixpoint is usually the most precise answer an abstract interpreter could give us. However for computational reasons (the abstract domain may be infinite for example) the sequence generated by $F^n(\bot)$ may not stabilise. In order to ensure that the fixpoint computation terminates we can consider another way of ensuring that we reach a fixpoint. The intuition is to replace the sequence $F^n(\bot)$ by another computation sequence which is guaranteed to terminate. This new sequence uses an acceleration operator called widening.

Definition 2.10 (Widening Operator)**.** A widening operator is a function $\triangledown : A \times A \to A$ such that for all $a_1, a_2 \in A$, $a_1 \sqsubseteq a_1 \triangledown a_2$ and $a_2 \sqsubseteq a_1 \triangledown a_2$, and for all increasing chains $a_0 \sqsubseteq a_1 \sqsubseteq ...$, the chain $y_0 = a_0, ..., y_{i+1} = y_i \triangledown a_{i+1}$ eventually stabilises.

Using a widening operator we can now define a fixpoint computation sequence which will terminate and overapproximates the least fixpoint:

$$\begin{array}{lcll} A_0 & = & \bot & \\ A_i + 1 & = & A_i & \text{if } F(A_i) \sqsubseteq A_i \\ & = & A_i \triangledown F(A_i) & \text{otherwise} \end{array}$$

The key point about the widening is that we sacrifice some information in order to for the abstract interpreter to return an answer - even if that answer is $\top$.

2.3.4 Concrete Domains

As mentioned previously, we need to define a concrete domain and an abstract domain. The concrete domain is the reference semantics with which we define the abstract interpreter (using the abstract domain). In essence the concrete domain is used to prove that the abstract interpreter is sound.

- A concrete domain $\mathcal{C}$ with a partial order $\sqsubseteq_C$
- A bottom element $\bot_C$ and a top element $\top_C$.
- A join operator $\sqcup_C$. The join operator is used when computing the semantics of a program point with multiple input paths, such as exits from loops or the entry points of while loops.
- A function $[\![-]\!] : \mathcal{C} \to \mathcal{C}$ which defines the semantics of each operation in the programming language.

As a running example we will define a concrete domain based on sets of states. We define a state $\sigma : Var \to \mathbf{Z}$, where Var is a set of variables. The set of all states, which is the concrete domain, is denoted by Σ. The order on this domain is simply the subset order, $\subseteq$. The bottom element is the empty set $\emptyset$ and the top element is the set of all states Σ. The join operator is simply set union, $\cup$. As an example of a semantic function which could be defined for assignment statements we have:

$$[\![x := e]\!]S = \{\sigma[x \mapsto [\![e]\!]s] \mid \sigma \in S\}$$

where the notation $\sigma[x \mapsto v]$ means the state s where x has been remapped to have the value v.

2.3.5 Abstract Domains

Having defined a concrete domain, we now need to define an abstract domain which over-approximates the concrete domain.

- An abstract domain $\mathcal{A}$ with partial order $\sqsubseteq_A$.
- A bottom element $\bot_A$ and top element $\top_A$. The top element corresponds to the case that the abstract interpreter could not prove or discover the property.

- A join operator $\sqcup_A$.
- A function $[\![-]\!]^\sharp : \mathcal{A} \to \mathcal{A}$ which defines the abstract semantics of the language.

Like the concrete semantics there are $[\![-]\!]^\sharp$ is built from operations for the constructs in the language. In the case of loops or recursive function definitions we use fixpoints of some function which models the body of the loop or function. However in the abstract domain the computation of this fixpoint may not converge so we use acceleration techniques such as widening operators.

An example of an abstract domain is the interval domain. The interval domain for a single variable is defined as:

$$Interval = \{[l, h] \mid l, h \in \mathbf{Z}^\infty \wedge l \leq h\}$$

where $\mathbf{Z}^\infty = \mathbf{Z} \cup \{-\infty, \infty\}$, is the set of integers with plus and minus infinity symbols. The lattice is ordered by:

$$[l_1, h_1] \sqsubseteq_A [l_2, h_2] \text{ iff } l_2 \leq l_1 \wedge h_1 \leq h_2.$$

This is then lifted over to the lattice we would use in the abstract interpreter for intervals:

$$\Sigma^\# = Vars \to Interval$$

which is an abstraction of concrete states in the previous example: to each integer it assigns the interval the variable resides in. We can now define a function which evaluates expressions with respect to the interval domain. This function is then used to give the abstract semantics for the language begin analysed:

$$\begin{array}{lcl} eval(\sigma, var) & = & \sigma(var) \\ eval(\sigma, int) & = & [int, int] \\ eval(\sigma, E_1 op E_2) & = & absop(eval(\sigma, E_1), eval(\sigma, E_2)) \end{array}$$

where the abstract operator $absop$ is defined as:

$$absop([l_1, h_1], [l_2, h_2]) = [l_1 op l_2, h_1 op h_2]$$

Using these operators we can then define semantics for commands in the language. For example for assignment we could define the abstract semantics to be:

$$[\![x := e]\!]^\sharp \sigma^\# = \sigma^\#[x \mapsto eval(e)]$$

2.3.5.1 Disjunctive Analyses

For computational reasons many abstract interpreters do not keep explicit states for each possible path through the program. In order to for the abstract interpreter to terminate, they may merge information together. For example, suppose at a program point n there are three possible paths from the entry point of the program. A non-disjunctive analysis might infer that a variable x is within the range $[1, 100]$. This abstraction of the possible values x could have could be quite coarse. A *disjunctive* analysis might be able to infer more precise information about the range of values x could have, such as:

$$x \in ([1, 10] \vee [12, 20] \vee [98, 100])$$

Each of the disjuncts might correspond to one of the three paths that reach program point n. So a disjunctive analysis is much more precise than a non-disjunctive analysis, but comes at a great computational cost.

2.3.6 Relating Abstract and Concrete Domains

In order to show that the abstract semantics overapproximates the concrete semantics we need some way of linking the two. One way is to have a concretization function, which takes an element of the abstract domain and returns an element of the concrete domain which it overapproximates. More formally, a concretization function is a map $\gamma : \mathcal{A} \to \mathcal{C}$. This concretization function must be monotone with respect to the order in the abstract and concrete.

In the running example, we could define a concretization function to be:

$$\gamma(\sigma^{\#}) = \{\sigma \mid \sigma^{\#}(x) = [i, j] \implies \forall x \in Var. i \leq \sigma^{\#}(x) \leq j\}$$

The concretization relates abstract states with the set of states it overapproximates - given an abstract state, we return the set of states where each variable is contained within the range given in the abstract state.

Now we have the three main ingredients we need to define an abstract interpreter: a concrete semantics, an abstract semantics and a concretization function relating the two. To show that the abstract semantics is sound with respect to the concrete semantics we need to prove a theorem such as the following:

$$\forall a \in A. [\![P]\!](\gamma(a)) \sqsubseteq \gamma([\![P]\!]^{\sharp} a)$$

Such a theorem establishes that all possible answers given by the abstract interpreter are sound with respect to the concrete semantics: any behaviour reported by the abstract interpreter is a possible behaviour by the program.

Chapter 3

Liveness Analyses from Safety Analyses

In this short chapter we will give a short summary of a technique to convert an abstract interpreter [23] for safety properties into a termination analyzer. The work is based on a paper [7] published in 2007.[1]. A safety analysis computes a set of invariants at each program point. These invariants over-approximate the set of reachable states at that program point. We describe an algorithm that uses a safety analysis to produce a liveness analysis, specifically a termination analysis.

The algorithm can take any abstract interpretation based program analysis and produce a termination analysis using the results of the program analysis. As it can use any abstract interpretation, this method produces many different termination analyzers, by converting safety analyses into termination analyses.

3.1 The Algorithm

We will now describe the algorithm that is defined in Fig. 3.1. The algorithm takes in a program P, a set of locations L for which we will try to prove local termination lemmas, and starting abstract input state $I^\sharp$. The algorithm requires the following:

- SAFETYANALYSIS: The safety analyzer we will be using

[1]The author of this thesis worked on the implementations for the paper. The theoretical work was done by the co-authors. For this reason the proofs of soundness will not be included in this thesis, but the experimental results and outline of the algorithm will be included. For more detail the reader should refer to the paper [7]

```
01 TERMINATIONANALYSIS(P, L, I♯) {
02     SAs := SAFETYANALYSIS(P, I♯)
03     foreach ℓ ∈ L {
04         LTPreds[ℓ] := true
05         foreach q ∈ SAs such that pc(q) = ℓ {
06             TAs := SAFETYANALYSIS(P, STEP(P, {SEED(q)}))
07             foreach r ∈ TAs {
08                 if pc(r) = ℓ ∧ ¬WELLFOUNDED(r) {
09                     LTPreds[ℓ] := false
10                 }
11             }
12         }
13     }
14     return LTPreds
15 }
```

FIGURE 3.1: Parameterized termination analysis algorithm

.

- STEP: A single-step operation over the abstract domain of SAFETYANALYSIS. This operation usually corresponds to an abstract operation over primitive commands

We also have to define two extra operations, which are not usually given by the safety analysis.

- SEED: An operation that converts elements of the abstract domain of SAFETYANALYSIS into assertions over pairs of program states In essence we are converting a safety assertion into a liveness assertion using this operation.
- WELLFOUNDED: An operation that tries to find a well-founded relation that over-approximates a given assertion over pairs of program states.

Once we have all these ingredients we can define a termination analyzer. We givea brief outline of the algorithm in Fig 3.1:

1. We first run SAFETYANALYSIS to get invariants at each program location (line 02). We denote the results of the static analysis at program point n using SA_n.
2. For each of the program locations (line 03) (in our cut-point set), we take the invariant and perform a seeding operation using SEED to get a relation over program states (line 06). The results of the analysis are stored in an array SA_s, whose indices correspond to program locations.
3. We then take a 'step' in the program analyzer starting from this new assertion, and then re-run the original program analysis starting from the seeded state (line 06). We need to perform the step operation in order to over-approximate the non-reflexive, transitive closure, as required by the Podelski-Rybalcheko result. The results from this step in

the analysis are denoted by TA_n, which is result from the second run of the analysis at program point n. These results are stored in an array TA_s.

4. Once we have run the analysis, we then check to see if the resulting assertions are well-founded (line 08). This is done by calling some rank function synthesis engine. Note that we do not need to receive a ranking function from the rank function synthesis engine, we just need to know if the assertions generated are well-founded.

3.1.1 Safety Analyses

We assume that we are using an abstract interpretation, that soundly over-approximate the programs concrete semantics. The abstract interpretation consists of two parts: a STEP operation that over-approximates a single step transition in the program and SAFETYANALYSIS that over-approximates the reflexive transitive closure of the program. Usually SAFETYANALYSIS is defined in terms of STEP, but we do not require this to be so. Standard techniques in abstract interpretation such as widening and narrowing could be used: all we require is the output of the abstract interpreter. Proving termination is very path sensitive, so we use powerset domains or use standard techniques to lift to powerset domains [42, 52] in order to have disjunctive information. The algorithm still works if we do not have a powerset (disjunctive) domain, but more often than not we will not be able to prove termination.

3.1.2 Seeding

Seeding is an operation we must define on the abstract domain. Seeding is commonly used for recording computational history in states and is analogous to the use of ghost variables in program logics. Since safety abstract interpreters overapproximate states on programs nad liveness analyses overapproximate relations on states, we need an operation to lift assertions on states to assertions on relations. The seeding operation performs this lifting.

Seeding is usually a simple addition or modification to the abstract domain of the safety analysis. The seeding operation has to be tailored for each safety analysis being used to generate a termination analysis.

Seeding has to lift a assertion over states to an assertion over pairs of states. To do this we will assume that there are two types of variable in the abstract domain: program variables and logical variables. The program variables correspond to variables used in the program. Logical variables are variables that are not manipulated by the program, through primitive commands

such as assignment. These two sets are disjoint. The definition of seed is:

$$\textsc{Seed}(A) = A \wedge \bigwedge_{v \in Var} \text{`}v = v$$

where $\text{`}v$ is a logical variable. SEED defines the initial liveness assertion. Intuitively, the primed logical variables record the previous value of the variable, and unprimed variables correspond to the current value. The addition of equalities between variables and the corresponding primed variables takes a 'snapshot' of the memory, so that when we run the analysis again we will try to keep track of the relationship between variables over the execution of a program loop.

This new seeded assertion is now liveness assertion: a binary relation on states:

$$\gamma(A) = \{(s,t) \mid pc(s) = pc(t) \wedge ((\text{`}s,t) \models A)\}$$

For example, suppose we have a the assertion $A = x < y \wedge y > 0$ which represents the set of states where the value of x is less than the value of y and y is greater than 0. Seeding this assertion gives us:

$$\textsc{Seed}(A) = x < y \wedge y > 0 \wedge \text{`}x = x \wedge \text{`}y = y$$

Which now represents a relation between states: the value of x and y has not changed, but the current value of x is less than y and the current value of y is greater than 0. The key point is that the new primed variables are logical variables, that the abstract interpreter does not distinguish from any other variable. In a sense we are tricking the analysis to produce assertions over relations on states rather than just states. The seeding operation is deliberately simple: the seeded assertion is only a starting point for liveness analysis.

3.1.3 Checking Well-Foundedness

Recent work has shown that generating ranking functions is complete for linear expressions. In all the implementations that we have produced we have used RANKFINDER [48]. Other tools that could be used include POLYRANK [10, 12]. The choice of safety analysis will affect the choice of which rank function synthesis engine to be used. If the underlying safey analysis performs is based on linear arithmetic relationships between program variables then we need a rank function synthesis engine which produces linear ranking functions.

This, together with the choice of SEED operation give us much flexibility allowing us to produce many instances of our algorithm.

```
01          while (x>0 && y>1) {
02             if (nondet()) {
03                   x = x - 1;
04               } else {
05                    y = y - 1;
06               }
07           }
```

FIGURE 3.2: Example program fragment.

3.2 Illustrative example

In this section we will use the program in Fig. 3.2 while stepping through the TERMINATIONANALYSIS algorithm in Fig. 3.1. We will use a safety analysis based on the Octagon [43] domain, which can express conjunctions of inequalities of the form $\pm x + \pm y \leq c$ for variables x and y and constant c. The safety analysis will return a set of invariants at each program point. An assertion holds at program point ℓ if and only if it is always valid at *the beginning* of the line, before executing the code contained at that line. Also note that to prove the termination of the loop we only need to prove termination at program point 02, the top of the loop, so we will set $L = \{02\}$.

Given a set of locations L, the parameterized termination analysis attempts to establish termination arguments for each location $\ell \in L$, when the program P is run from starting states satisfying the input condition $I^\sharp$.

- **Safety analysis (Line 2 of Fig. 3.1):** We start by running a safety analysis using the Octagon domain (possibly with a disjunction-recovering post-analysis). We will set the starting point of the safety analysis as: $I^\sharp = (pc = 0)$ From this starting point the analysis produces:

  $$SA_{02} \triangleq pc = 0 \wedge x \geq 1 \wedge y \geq 1$$

  We write SA_n to represent the safety invariant at program point n discovered by the safety analysis. SA_{02}, represents the set of states:

  $$\{s \mid s(\texttt{pc}) = 02 \wedge s(\texttt{x}) \geq 1 \wedge s(\texttt{y}) \geq 1\}$$

- **Inferring liveness assertions (Lines 6 and 6)** We take all of the disjuncts in the safety assertion at location 02 (in this case there is only one, SA_{02}) and convert them into binary relations from states to states:

  $$\textsc{Seed}(SA_{02}) = (pc = 02 \wedge \text{`}pc = 02 \wedge x \geq 1 \wedge y \geq 1 \wedge \text{`}x = x \wedge \text{`}y = y)$$

SEED(SA_{02}) is a state that uses primed variables not used in the program: these variables can be thought of as logical constants. However, in another sense, SEED(SA_{02}) can be thought of as a binary relation on program states:

$$\begin{array}{lll} \{(s,t) & | & s(\mathtt{pc}) = t(\mathtt{pc}) = 02 \\ & \wedge & s(\mathtt{x}) = t(\mathtt{x}) \\ & \wedge & s(\mathtt{y}) = t(\mathtt{y}) \\ & \wedge & t(\mathtt{x}) \geq 1 \\ & \wedge & t(\mathtt{y}) \geq 1 \qquad \} \end{array}$$

Notice that we're using $`x$ to represent the value of x in s, and x to represent the value of x in t.

We then step the program once from SEED(SA_{02}) with STEP, approximating one step of the program's semantics, giving us:

$$`pc = 02 \wedge pc = 03 \wedge x \geq 1 \wedge y \geq 1 \wedge `x = x \wedge `y = y$$

The program counter has been incremented, and since we have passed through a non-deterministic choice, nothing is done to variables. Finally, we run SAFETYANALYSIS again with this new state as the starting state, and the isolated subprogram O as the program, which gives us a set of variant assertions at locations 02, 03,.... that corresponds to the set TAs in the TERMINATIONANALYSIS algorithm of Fig. 3.1. The assertions produced at program location 2 are:

$$TA^A_{02} \triangleq (`pc = 02 \wedge pc = 02 \wedge x \geq 1 \wedge y \geq 1 \wedge `x \geq x + 1 \wedge `y \geq y)$$
$$TA^B_{02} \triangleq (`pc = 02 \wedge pc = 02 \wedge x \geq 1 \wedge y \geq 1 \wedge `x \geq x \wedge `y \geq y + 1)$$

The union of these two relations

$$TA^A_{02} \vee TA^B_{02}$$

forms the liveness assertion for line 02 in P: a superset of the possible transitions from states at 02 to states also at line 02 reachable in 1 or more steps of the program's execution. The disjunction $TA^A_{02} \vee TA^B_{02}$ is a superset of the non-reflexive transitive closure of the program's transition relation restricted to pairs of reachable states both at location 02.

One important aspect of this technique is that the analysis is not aware of the intended meaning of variables like $`x$ and $`y$: it simply treats them as symbolic constants. It does not know that the assertions are representing relations.

- **Proving local termination predicates:** We have computed such a finite set: TA^A_{02}, TA^B_{02}, We know that the union of these three relations over-approximates the transitive closure

of the transition relation of the program P limited to states at location 02. Furthermore, each of the relations are, in fact, well-founded. Thus, we can reason that the program will not visit location 02 infinitely, by using the Podelski-Rybalchenko result. The final step is to prove that the relations TA^A_{02}, TA^B_{02} are well-founded. Because these relations are represented as a conjunction of linear inequalities, they can be automatically proved well-founded by rank function synthesis engines such as RANKFINDER [48] or POLYRANK [10, 12]. To see why TA^A_{02} is well-founded, we can argue as follows: '$x \geq x + 1$ means that the variable x has decreased in value: $x \geq 1$ provides a lower bound. Hence we cannot repeat TA^A_{02} infinitely often. A similar argument holds for TA^B_{02}.

3.3 Implementation

We will now present two related instances of our algorithm, both of which are based on safety analyzers for numerical properties. The purpose of this section is to give the reader a picture for the state of the art in termination proving. In general the CEGAR approach of TERMINATOR is slow but precise, while the abstract interpretation based on safety analyses are faster but less precise.

The Polyhedra [26] and Octagon [43] domains are two important numerical abstract domains for program analysis. The polyhdera domain is the more expressive but costly in terms of computation. The octagon domain is less expressive but faster. The polyhedra domain can express affine inequalities of the form: $\bigwedge_j \sum_i \alpha_{ij} X_i \leq \beta_j$, while the octagon domain can express inequalities of the form $\pm X \pm Y \leq c$.

3.3.1 Results

We have implemented two analyzers: one based on the Octagon Domain and the other on the Polyhedra Domain. The results are listed in Fig 3.3. We compared the results with two existing termination provers: one based on POLYRANK [9] and TERMINATOR [20]. We will now explain the symbols in Fig 3.3. A ✓ means that a proof of termination was found, † means that a false counterexample to termination was found. T/O means that the analyzer hit the timeout threshold that was set at 500 seconds. ⊘ means that a termination bug was found. The tools referred to in the table are:

O) OCTATERM is the termination analysis induced by OCTANAL [44] composed with a post-analysis phase (see below). OCTANAL is included in the Octagon domain library distribution. During these experiments OCTATERM was configured to return "Terminating" in

	1		2		3		4		5		6	
O	0.11	✓	0.08	✓	6.03	✓	1.02	✓	0.16	✓	0.76	✓
P	1.40	✓	1.30	✓	10.90	✓	2.12	✓	1.80	✓	1.89	✓
PR	0.02	✓	0.01	✓	T/O	-	T/O	-	T/O	-	T/O	-
T	6.31	✓	4.93	✓	T/O	-	T/O	-	33.24	✓	3.98	✓

(a) Results from experiments with termination tools on arithmetic examples from the Octagon Library distribution.

	1		2		3		4		6		7	
O	0.30	†	0.05	†	0.11	†	0.50	†	0.10	†	0.17	†
P	1.42	✓	0.82	✓	1.06	†	2.29	†	2.61	†	1.28	†
PR	0.21	✓	0.13	✓	0.44	✓	1.62	✓	3.88	✓	0.11	✓
T	435.23	✓	61.15	✓	T/O	-	T/O	-	75.33	✓	T/O	-

	8		9		10		11		12	
O	0.16	†	0.12	†	0.35	†	0.86	†	0.12	†
P	0.24	†	1.36	✓	1.69	†	1.56	†	1.05	†
PR	2.02	✓	1.33	✓	13.34	✓	174.55	✓	0.15	✓
T	T/O	-	T/O	-	T/O	-	T/O	-	10.31	†

(b) Results from experiments with termination tools on arithmetic examples from the POLYRANK distribution.

	1		2		3		4		5	
O	1.42	✓	1.67	⊘	0.47	⊘	0.18	✓	0.06	✓
P	4.66	✓	6.35	⊘	1.48	⊘	1.10	✓	1.30	✓
PR	T/O	-	T/O	-	T/O	-	T/O	-	0.10	✓
T	10.22	✓	31.51	⊘	20.65	⊘	4.05	✓	12.63	✓

	6		7		8		9		10	
O	0.53	✓	0.50	✓	0.32	✓	0.14	⊘	0.17	✓
P	1.60	✓	2.65	✓	1.89	✓	2.42	⊘	1.27	✓
PR	T/O	-	T/O	-	T/O	-	T/O	-	0.31	✓
T	67.11	✓	298.45	✓	444.78	✓	T/O	-	55.28	✓

(c) Results from experiments with termination tools on small arithmetic examples taken from Windows device drivers. Note that the examples are small as they must currently be hand-translated for the three tools that do not accept C syntax.

FIGURE 3.3: Experimental results for Four Termination Analyzers

the case that each of the assertions inferred entailed their corresponding local termination predicate. The WFCHECK operation was based on RANKFINDER.

P) POLYTERM is the termination analysis similarly induced from an safety analysis POLY based on the New Polka Polyhedra library [39].[2]

PR) A script suggested in [9] that calls the tools described in the POLYRANK distribution [10, 12] with increasingly expensive command-line options.

T) TERMINATOR [20].

These tools, except for TERMINATOR, were all run on a 2GHz AMD64 processor using Linux 2.6.16. TERMINATOR was executed on a 3GHz Pentium 4 using Windows XP SP2. Using different machines is unfortunate but somewhat unavoidable due to constraints on software library dependencies, etc. Note, however, that TERMINATOR running on the faster machine was still slower overall, so the qualitative results are meaningful.

Fig. 3.3 contains the results from the experiments performed with these provers. For example, Fig. 3.3(a) shows the outcome of the provers on example programs included in the OCTANAL distribution. Example 3 is an abstracted version of heapsort, and Example 4 of bubblesort. In this case OCTATERM is the clear winner of the tools. POLYRANK performs poorly on these cases due to the fact that any fully-general translation scheme from programs with full-fledged control-flow graphs to POLYRANK's input format will at times confuse the domain-specific rank-function search heuristics used in POLYRANK.

Fig. 3.3(b) contains the results from experiments with the four tools on examples from the POLYRANK distribution.[3] The examples can be characterized as small but famously difficult (*e.g.* McCarthy's 91 function). We can see that, in these cases, neither TERMINATOR nor the induced provers can beat POLYRANK's hand-crafted heuristics. POLYRANK is designed to support very hard but also carefully expressed examples. In this case each of these examples from the POLYRANK distribution are written such that POLYRANK's heuristics find a termination argument.

Fig. 3.3(c) contains the results of experiments on fragments of Windows device drivers. These examples are small because we currently must hand-translate them before applying all of the tools but TERMINATOR. In this case OCTATERM again beats the competition. However, we should keep in mind that the examples from this suite that were passed to TERMINATOR contained pointer aliasing, whereas aliasing was removed by hand in the translations used with POLYRANK, OCTATERM and POLYTERM.

[2] POLY uses the same code base as OCTANAL but calls an OCaml module for interfacing with New Polka, provided with the OCTANAL distribution.

[3] Note also that there is no benchmark number 5 in the original distribution. We have used the same numbering scheme as in the distribution so as to avoid confusion.

From these experiments we can see that the technique of inducing termination analyses with TERMINATIONANALYSIS is promising. For programs of medium difficulty (*i.e.* Fig. 3.3(a) and Fig. 3.3(c)), OCTATERM is many orders of magnitude faster than the existing program termination tools for imperative programs.

3.3.2 Conclusion

We have outlined an algorithm to convert a safety analysis into a termination analysis, a form of liveness analysis. The algorithm is generic and makes as few assumptions as possible about the underlying safety analysis. The termination provers produced using the algorithm are extremely fast, comparable to state of the art termination provers. The technique presented here is completely automatic. The algorithm takes in a program and tries to produce a termination argument. A key point of this technique is that no work needs to be done on producing new abstract transformers, or widening operators in order to produce a termination analyser; we re-use the existing power of the underlying safety analysis. The termination provers produced using this technique are extremely fast, There are however a number of problems with the approach presented in this chapter.

Since we are trying to prove program termination, we really need an safety analysis which is disjunctive - it returns a set of invariants at each program point rather than just performing join operations to return one invariant. Due to computational concerns many safety analyses are not disjunctive, but also perform widening and narrowing operations in order to ensure that the analysis terminates. We have a technique for extracting disjunctive information after an safety analysis has been completed, but since widening and narrowing operators may have been used during the safety analysis, this technique is limited. If the safety analysis is not disjunctive, then its widening operator will not have been designed with disjunction in mind, and so we will lose a lot of information which could potentially be useful for proving termination.

As we re-use the operations of the safety analyses, we are dependent on the acceleration methods used such as widening operators. These widening operators have been designed for proving invariant assertions. This means that, very often, for complicated programs they lose too much information which could be useful in finding a termination argument.We are also dependent on the results of the fixpoint computation performed by the safety analysis. We are using the safety analysis as a black box. It might be useful if we could do operations within the fixpoint computation loop in order to keep information necessary for proving termination.

In the next chapter we will address this issue by designing an abstract domain specifically for proving termination and also performing some operations during the fixpoint computation to help find a termination argument.

Chapter 4

Ranking Abstractions

In the previous chapter, we described a way of using an existing safety analysis, such as Octagon [44] or Polyhedra [26] analyses, to construct a new termination analysis, by borrowing the abstract operators from the safety analysis. The advantage to this approach is that the resulting termination analysis is very fast, when compared to the specialized program termination provers such as TERMINATOR [19].

However, the approach has one serious shortcoming. The constructed termination analysis often discards the information necessary for termination proofs. As a result, the resulting termination analysers fail to prove termination of some programs for which proving termination shoud be possible. The cause of this shortcoming is that the underlying safety analyses are not designed for termination, so their abstract operators do not necessarily track information related to termination proofs. For instance, a usual safety analysis is not disjunctive, meaning that it uses a very crude over-approximation of disjunction. Although the non-disjunctiveness makes the analysis fast, it forces the analysis to combine information from multiple execution paths of a program, but keeping such path-sensitive information often turns out to be important for termination proofs. The paper [7] reporting the approach in Chapter 3, proposed an ad-hoc technique to amend this precision problem, but this technique works only in some cases.

In this chapter [1], we describe abstract interpreters specially designed for proving termination. The abstract interpreters are as accurate as TERMINATOR and as fast as the analyses in Chapter 2.3.6. To achieve the high accuracy, the abstract interpreters are fully disjunctive, and they can track disjunction precisely. The abstract interpreters avoid a well-known problem of being fully disjunctive, namely using too many disjuncts, by ensuring that every disjunct keeps only the information relevant to termination proofs. Specifically, we use ranking function synthesis engines to identify termination-relevant information of each disjunct. The high performance of

[1] This work was published in [14]

```
01          while (x>0 && y>1) {
02             if (nondet()) {
03                   x := x - 1;
04                   y := *;
05               } else {
06                    y := y - 1;
07               }
08           }
```

FIGURE 4.1: Example Program

the abstract interpreters is achieved by using an overapproximating fixpoint computation, which automatically guarantees that the analysis results overapproximate the concrete meaning of programs. The algorithm of TERMINATOR does not have such automatic guarantee, so it has to perform an expensive additional inclusion check.

4.1 Informal Description of the Analysis

In this section we informally describe the termination analysis using an example. Later, in Section 4.2, we provide a more formal description.

Consider the program in Figure 4.1. This program illustrates the limitation of known termination analyses. The Octagon-based and Polyhedra-based termination analyses in Chapter 2.3.6 and the paper [7] can quickly (in comparison to TERMINATOR) infer that the relation $`x \geq 0 \wedge `x \geq x$ holds between any state at $\ell = 2$ and any previous state at $\ell = 2$, where $`x$ and $x$ denote previous and current values of $x$ respectively. (Note that $`x$ is denoting *some* previous value of x, and not necessarily the *last* value). Unfortunately, this relation is insufficient to prove termination of the loop, as it is not (disjunctively) well-founded—the condition sufficient for proving termination as required by the Podelski-Rybalchenko result 2.1.3.

In contrast TERMINATOR can prove the example terminating, but at a great computational cost. TERMINATOR finds the following disjunctively well-founded relation at $\ell = 2$:

$$(`x \geq 0 \ \wedge \ `x - 1 \geq x) \ \vee \ (`y \geq 0 \ \wedge \ `y - 1 \geq y)$$

To find this relation TERMINATOR performs three rounds of refinement on the relation itself and 9 rounds of abstraction/refinement for the checking of the 3 candidate assertions, resulting in the discovery of 21 transition predicates.

The termination analyses in this chapter gives us TERMINATOR's accuracy at the speed of the Octagon-based termination analysis. The new analysis finds the relation

$$(`x \geq 0 \ \wedge \ `x - 1 \geq x) \ \vee \ (`y \geq 0 \ \wedge \ `y - 1 \geq y \ \wedge \ `x = x)$$

in 0.02s.

Concretely, the new analysis uses a disjunctive domain of ranking relations conjoined with the information about unchanged variables. That is: disjunctions of relations of the form $\varphi_e \wedge V_X$, where

$$V_X \stackrel{def}{=} \bigwedge_{x \in X} `x = x, \qquad \varphi_e \stackrel{def}{=} `e \geq 0 \ \wedge \ `e - 1 \geq e,$$

and $`e$ is the expression $e$ with all variables $x$ replaced by their corresponding pre-primed versions $`x$. Let R represent the transition relation of the loop body of our program in DNF:

$$\begin{aligned} R &\stackrel{def}{=} R_1 \ \vee \ R_2, \\ R_1 &\stackrel{def}{=} `x > 0 \ \wedge \ `y > 0 \ \wedge \ x = `x - 1, \\ R_2 &\stackrel{def}{=} `x > 0 \ \wedge \ `y > 0 \ \wedge \ x = `x \ \wedge \ y = `y - 1. \end{aligned}$$

Our analysis begins by taking each disjunct in R and performing rank-function synthesis on it. In this case we get

$$\mathsf{RFS}(R_1) = x \quad \text{and} \quad \mathsf{RFS}(R_2) = y.$$

For each disjunct, the analysis also computes a set of variables whose values do not change. In this example, it determines that R_1 can change both x and y, but R_2 does not change variable x. Thus, we begin our analysis with the initial abstract state $\delta_0 \stackrel{def}{=} \varphi_x \vee (\varphi_y \wedge V_{\{x\}})$, that is,

$$\delta_0 \ = \ (`x \geq 0 \ \wedge \ `x - 1 \geq x) \ \vee \ (`y \geq 0 \ \wedge \ `y - 1 \geq y \ \wedge \ `x = x).$$

Note that δ_0 overapproximates the loop body R.

The meaning of this initial abstract state (*i.e.* $\gamma(\delta_0)$) is set of all finite sequences of program states $s_i s_{i+1} \ldots s_{i+n}$ such that

$$\begin{gathered} \big(s_i(x) \geq 0 \ \wedge \ s_i(x) - 1 \geq s_{i+n}(x)\big) \\ \vee \\ \big(s_i(y) \geq 0 \ \wedge \ s_i(y) - 1 \geq s_{i+n}(y) \ \wedge \ s_i(x) = s_{i+n}(x)\big). \end{gathered}$$

The analysis then computes the next abstract state δ_1 that overapproximates the relational composition of δ_0 and R. It takes each disjunction from δ_0 and each disjunction from R, composes

$$
\begin{array}{rcl}
E & ::= & x \mid r \mid E + E \mid r \times E \\
B & ::= & E = E \mid E \neq E \mid E \leq E \mid E < E \mid B \wedge B \mid B \vee B \mid \neg B \\
a & ::= & x := E \mid x := * \\
C & ::= & a \mid C; C \mid C [] C \mid \texttt{if}\ B\ C\ C \mid \texttt{while}\ B\ C
\end{array}
$$

FIGURE 4.2: Syntax of the Programming Language

them, performs rank function synthesis, infers variables that do not change, and constructs the union of the new ranking relations together with δ_0. In this case we find:

$$
\begin{array}{rclrcl}
\mathsf{RFS}(\varphi_x; R_1) & = & x & \mathsf{RFS}(\varphi_x; R_2) & = & x \\
\mathsf{RFS}((\varphi_y \wedge V_{\{x\}}); R_1) & = & x & \mathsf{RFS}((\varphi_y \wedge V_{\{x\}}); R_2) & = & y
\end{array}
$$

We also find that the last composition $((\varphi_y \wedge V_{\{x\}}); R_2)$ does not change x. Thus,

$$
\delta_1 \quad = \quad \big(\delta_0 \ \vee \ \varphi_x \ \vee \ \varphi_x \ \vee \ \varphi_x \ \vee \ (\varphi_y \ \wedge \ V_{\{x\}})\big) \quad = \quad \delta_0.
$$

Since δ_0 is a fixpoint and δ_0 overapproximates R, we know that $\forall i > 0.\ R^i \subseteq \delta_0$, that is, $R^+ \subseteq \delta_0$. Thus, because δ_0 is disjunctively well-founded, the result by Podelski and Rybalchenko [50] tells us that R is well-founded—meaning that the loop of our program guarantees termination.

Note that rank function synthesis is extremely efficient, meaning that our implementation of the analysis can compute the relation δ_0 for $\ell = 2$ as fast as the Octagon-based termination analyzer (*i.e.* in 0.02s) [7]. In contrast to the Octagon-based analyzer, however, we compute a relation that is sufficiently strong to establish termination.

To sum up, the essence of our method is that we symbolically execute the body of the loop, and then perform abstraction by calling a rank synthesis engine. This in effect abstracts all information except those that are relevant to termination.

4.2 Formal Framework

4.2.1 Programming Language and Concrete Semantics

In this chapter we consider a simple while language with no procedures, where all variables store rational numbers. Let Vars be a finite set of program variables $x, y, z, \ldots$ and let r represent rational numbers. The syntax of the language is given in Figure 4.2.

We have two forms of assignments: normal assignment $x := E$ and non-deterministic random assignment $x := *$. The non-deterministic assignment is used to model features of a common

$$
\begin{array}{rcl}
[\![C]\!] & \in & \mathcal{P}(\mathcal{T}) \\
[\![x := E]\!] & \stackrel{def}{=} & \{ss' \mid (s, s'[x \mapsto [\![E]\!]s])\} \\
[\![x := *]\!] & \stackrel{def}{=} & \{ss' \mid (s, s'[x \mapsto r]) \wedge r \in P\} \\
[\![C_1; C_2]\!] & \stackrel{def}{=} & \mathsf{seq}([\![C_1]\!], [\![C_2]\!]) \\
[\![C_1 [] C_2]\!] & \stackrel{def}{=} & [\![C_1]\!] \cup [\![C_2]\!] \\
[\![\texttt{if } B \; C_1 \; C_2]\!] & \stackrel{def}{=} & \mathsf{seq}(\{ss \mid [\![B]\!]s = \mathsf{true}\}, [\![C_1]\!]) \cup \mathsf{seq}(\{ss \mid [\![B]\!]s = \mathsf{false}\}, [\![C_2]\!]) \\
[\![\texttt{while } B \; C]\!] & \stackrel{def}{=} & \textbf{let} \;\; T_0 = \mathsf{seq}(\{ss \mid [\![B]\!]s = \mathsf{true}\}, [\![C]\!]) \\
 & & \quad\;\; F = \lambda T.\, \mathsf{States} \cup \mathsf{seq}(T_0, T) \\
 & & \textbf{in } \mathsf{seq}(\mathsf{gfix}\, F, \;\{ss \mid [\![B]\!]s = \mathsf{false}\})
\end{array}
$$

FIGURE 4.3: Concrete Collecting Semantics of Programs

programming language, such as C, that are not covered by the language in 4.2. We also point out that the language includes a non-deterministic choice $C_1[]C_2$ and that all variables in our language store rational numbers.

The semantics of our language is given in terms of traces. Let States to be the set of maps from variables to rational numbers:

$$\mathsf{States} \stackrel{def}{=} \mathsf{Vars} \to \mathsf{Rationals}$$

which is ranged over by s. And let $\mathcal{T}$ be the set of nonempty finite or infinite sequences of states:

$$\mathcal{T} \stackrel{def}{=} \mathsf{States}^{+} \cup \mathsf{States}^{\infty}$$

Define a composition operator seq for sequence sets in $\mathcal{T}$:

$$
\begin{array}{rcl}
\mathsf{seq} & : & \mathcal{P}(\mathcal{T}) \times \mathcal{P}(\mathcal{T}) \to \mathcal{P}(\mathcal{T}) \\
\mathsf{seq}(T, T') & \stackrel{def}{=} & \{\tau s \tau' \mid \tau s \in (T \cap \mathsf{States}^{+}) \wedge s\tau' \in T'\} \cup (T \cap \mathsf{States}^{\infty}).
\end{array}
$$

Here we use τ to mean possibly empty finite or infinite sequences of states. We will keep this usage of τ throughout this chapter.

Following Cousot's work [22], we define the concrete collecting trace semantics, where a command C means a subset of $\mathcal{T}$. The semantics appears in Figure 4.3. In the figure, we assume a standard interpretation of expressions $[\![E]\!]$ and booleans $[\![B]\!]$. For non-deterministic assignment, we assign the variable x one of a rational number from a large but finite set of rationals P. By selecting a value from a finite set of rationals we avoid theoretical issues with regards to non-determinism and rational numbers. We use greatest fixpoints in the semantics of loops in order to include infinite execution traces of loops if they exist. If we used the least fixpoint we would

exclude all infinite traces, so all programs would be considered terminating. The semantics defined is a standard one in the field of abstract interpretation [22], and we believe this justifies this definition using greatest fixpoints.

4.2.2 Abstract Domain

Our abstract interpreter is parameterized by a domain for representing relations on states. The parameter domain has to satisfy the following:

1. A set D and a concretization function $\gamma_r : D \to \mathcal{P}(\mathsf{States} \times \mathsf{States})$ (where the target $\mathcal{P}(\mathsf{States} \times \mathsf{States})$ is ordered by the subset relation).

2. An abstract identity element d_{id} in D, that satisfies

$$\Delta_{\mathsf{States}} \subseteq \gamma_r(d_{\mathsf{id}})$$

   where Δ_{States} is the identity relation on States. The identity element plays the role of bottom elements in abstract interpretation.

3. An operator $\mathsf{RFS}\colon D \to \mathcal{P}_{\mathsf{fin}}(D) \uplus \{\top\}$, which synthesizes ranking functions (where $\mathcal{P}_{\mathsf{fin}}(D)$ denotes the set of all finite subsets of D) . We assume the following two conditions for this operator:

   (a) RFS computes an overapproximation:

$$\mathsf{RFS}(d) \neq \top \implies \gamma_r(d) \subseteq \bigcup\{\gamma_r(d') \mid d' \in \mathsf{RFS}(d)\}.$$

   (b) $\mathsf{RFS}(d)$ denotes a well-founded relation:

$$\mathsf{RFS}(d) \neq \top \implies \bigcup\{\gamma_r(d') \mid d' \in \mathsf{RFS}(d)\} \text{ is well-founded.}$$

   The RFS operator is the key to producing a termination analyser: it will attempt to discover reasons why the program terminates by trying to synthesize ranking functions.

4. An abstract transfer function $\mathsf{trans}(a)$ for each atomic commands a (i.e., $a \equiv (x := E)$ or $a \equiv (x := *)$). The function $\mathsf{trans}(a)$ has type $D \to \mathcal{P}_{\mathsf{fin}}(D)$, and satisfies

$$\forall d \in D.\ (\gamma_r(d); [\![a]\!]) \subseteq \bigcup\{\gamma_r(d') \mid d' \in \mathsf{trans}(a)(d)\}$$

where the semicolon means the usual composition of relations and $[\![a]\!]$ is the concrete semantics of a seen as a relation on states (which is possible because $[\![a]\!]$ consists of traces of length 2.).

5. An abstract composition operator $\mathsf{comp}\colon D \times D \to D$ such that

$$\gamma_r(d); \gamma_r(d') \quad \subseteq \quad \gamma_r(\mathsf{comp}(d, d')).$$

6. A function $\mathsf{filter}_B : D \to D$ for each boolean expression B such that

$$\{(s, s') \mid (s, s') \in \gamma_r(d) \wedge [\![B]\!]s' = \mathsf{true}\} \quad \subseteq \quad \gamma_r(\mathsf{filter}_B(d)).$$

Intuitively we have a set D of relations, some of which are well-founded. This set comes with an algorithm RFS, which overapproximates a relation by a ranking relation. It also has three operators—trans, comp and filter_B, that soundly model all the atomic commands, concrete relation composition and the filtering of states by the boolean condition B in loops or conditional statements. One example of D is the set of conjunction of linear constraints. In this case, we can use a linear rank synthesis engine, such as RANKFINDER, and define RFS as will be shown in Section 4.4.

The abstract domain $\mathcal{A}$ of our analyzer is constructed using the parameter domain D as follows:

$$\mathcal{A} \stackrel{def}{=} (\mathcal{P}_{\mathsf{fin}}(D))^{\top} \quad (\text{i.e., } \mathcal{P}_{\mathsf{fin}}(D) \uplus \{\top\}).$$

It is ordered by the the subset order $\sqsubseteq$ extended with $\top$. That is, for $A, A' \in \mathcal{A}$, we have that

$$A \sqsubseteq A' \iff (A' = \top) \vee (A, A' \in \mathcal{P}_{\mathsf{fin}}(D) \wedge A \subseteq A').$$

Each abstract element A in $\mathcal{A}$ denotes a set of traces (i.e., finite or infinite sequences of states). The element $\top$ denotes the set of all traces, including infinite ones, and non-$\top$ elements A denote a set of *finite* nonempty traces whose initial and final states are related by some d in A. For non-$\top$ elements A, let $\gamma_r(A)$ be $\bigcup\{\gamma_r(d) \mid d \in A\}$, the disjunction of d's in A. Recall that $\mathcal{T}$ is the set of all nonempty traces:

$$\mathcal{T} \quad = \quad \mathsf{States}^{+} \cup \mathsf{States}^{\infty}.$$

The formal meaning of A is given by a concretization function γ:

$$\begin{array}{rcl} \gamma & : & \mathcal{A} \to \mathcal{P}(\mathcal{T}) \\ \gamma(A) & \stackrel{def}{=} & \begin{cases} \mathcal{T} & \text{if } A = \top \\ \{s.\tau'.s \mid s[\gamma_r(A)]s'\} & \text{otherwise} \end{cases} \end{array}$$

where $|\tau|$ is the length of the trace τ, and $\mathsf{proj}(\tau, n)$ is the n-th state of the trace τ, and notation $s[r]s'$ means that s, s' are related by r. For instance, when $[x : n, y : m]$ is a state mapping x and y to n and m, a finite trace

$$[x : 1, y : 1][x : 2, y : 2][x : 5, y : 3][x : -2, y : 2]$$

belongs to $\gamma(\{`x{-}1 \geq x,\ `y{-}1 \geq y\})$, because x has a smaller value in the final state than in the initial state.

Our domain $\mathcal{A}$ is a complete lattice. The join of a family $\{A_i\}_{i \in I}$ of elements in $\mathcal{A}$ is given by the union of all A_i's, if none of A_i's is $\top$ and the union is finite. Otherwise, the join is $\top$.

4.2.3 Generic Abstract Interpreter

We now define a generic abstract interpreter in a denotational style, which uses the abstract domain from the previous section.

For functions $f\colon D \to \mathcal{A}$ and $g\colon D \times D \to \mathcal{A}$, let $f^\dagger, g^\dagger$ be their liftings to $\mathcal{A}$:

$$\begin{array}{rcl} f^\dagger & : & \mathcal{A} \to \mathcal{A} \\ f^\dagger(A) & \stackrel{def}{=} & \textbf{if } (A{=}\top) \textbf{ then } \top \textbf{ else } \bigsqcup_{d \in A} f(d) \end{array}$$

$$\begin{array}{rcl} g^\dagger & : & \mathcal{A} \times \mathcal{A} \to \mathcal{A} \\ g^\dagger(A, B) & \stackrel{def}{=} & \textbf{if } (A{=}\top \vee B{=}\top) \textbf{ then } \top \textbf{ else } \bigsqcup_{d \in A, d' \in B} \{g(d, d')\}. \end{array}$$

Using these liftings, we define the generic abstract interpreter, which is shown in Figure 4.4.[2] The argument A of the interpreter represents a set of finite or infinite traces that happen before the command c. The abstract interpreter computes an overapproximation of all traces that are obtained by continuing the execution of c from the end of traces in A.

[2] In the definition, we view $\mathsf{RFS}, \mathsf{trans}(a), \mathsf{filter}_B$ as functions of type $D \to (\mathcal{P}_{\mathsf{fin}}(D))^\top$.

$$
\begin{array}{rcl}
[\![C]\!]^\sharp & : & \mathcal{A} \to \mathcal{A} \\
[\![a]\!]^\sharp A & \stackrel{def}{=} & (\mathsf{trans}(a))^\dagger A \\
[\![C_1; C_2]\!]^\sharp A & \stackrel{def}{=} & ([\![C_2]\!]^\sharp \circ [\![C_1]\!]^\sharp) A \\
[\![C_1 [] C_2]\!]^\sharp A & \stackrel{def}{=} & [\![C_2]\!]^\sharp A \sqcup [\![C_1]\!]^\sharp A \\
[\![\texttt{if } B \; C_1 \; C_2]\!]^\sharp A & \stackrel{def}{=} & ([\![C_1]\!]^\sharp \circ \mathsf{filter}^\dagger_B) A \sqcup ([\![C_2]\!]^\sharp \circ \mathsf{filter}^\dagger_{\neg B}) A \\
[\![\texttt{while } B \; C]\!]^\sharp A & \stackrel{def}{=} & \textbf{let} \quad F = \lambda A'.\ (\mathsf{RFS}^\dagger \circ [\![C]\!]^\sharp \circ \mathsf{filter}^\dagger_B)(\{d_{\mathsf{id}}\} \sqcup A') \\
& & \qquad A_s = \{d_{\mathsf{id}}\} \sqcup A \\
& & \textbf{in } \mathsf{filter}^\dagger_{\neg B}\big(\mathsf{comp}^\dagger(A_s,\ \mathsf{fix}\ F)\big)
\end{array}
$$

FIGURE 4.4: Generic Abstract Interpreter

We assume that we have a fixpoint operator, denoted fix. This operator takes a function of the form $(\mathsf{RFS}^\dagger \circ F) : \mathcal{A} \to \mathcal{A}$, and returns an abstract element A in the image of $\mathsf{RFS}^\dagger$ such that

$$A = \top \ \vee \ \Big(A \neq \top \ \wedge \ (\mathsf{RFS}^\dagger \circ F)A \neq \top \ \wedge \ \gamma_r((\mathsf{RFS}^\dagger \circ F)A) \ \subseteq \ \gamma_r(A)\Big).$$

We could use the standard fixpoint iteration to define fix. Concretely, $\mathsf{fix}\,(\mathsf{RFS}^\dagger \circ F)$ is defined by the limit of the sequence $\{A_n\}$ where $A_0 = \{\}$ and $A_{n+1} = A_n \sqcup (\mathsf{RFS}^\dagger \circ F)(A_n)$.[3] The limit satisfies the above condition, because all pre-fixpoints A of $(\mathsf{RFS}^\dagger \circ F)$ (that are in the image of $\mathsf{RFS}^\dagger$) do so. However, it is not mandatory to use this standard iteration. In fact, a more optimized fix operator is used in the abstract interpreter of Section 4.4.

The most interesting case of the abstract interpreter is the loop. The best way to understand this case is to assume that fix is the standard fixpoint operator and to see a sequence generated during the iterative fixpoint computation:

$$
\begin{aligned}
A_0 &= \{\}, \\
A_1 &= A_0 \sqcup (\mathsf{RFS}^\dagger \circ F)\{d_{\mathsf{id}}\} \\
&= (\mathsf{RFS}^\dagger \circ F)\{d_{\mathsf{id}}\} \\
A_2 &= A_1 \sqcup (\mathsf{RFS}^\dagger \circ F)\big(\{d_{\mathsf{id}}\} \sqcup (\mathsf{RFS}^\dagger \circ F)\{d_{\mathsf{id}}\}\big) \\
&= (\mathsf{RFS}^\dagger \circ F)\{d_{\mathsf{id}}\} \sqcup (\mathsf{RFS}^\dagger \circ F)^2\{d_{\mathsf{id}}\}, \\
A_3 &= A_2 \sqcup (\mathsf{RFS}^\dagger \circ F)\big(\{d_{\mathsf{id}}\} \sqcup (\mathsf{RFS}^\dagger \circ F)\{d_{\mathsf{id}}\} \sqcup (\mathsf{RFS}^\dagger \circ F)^2\{d_{\mathsf{id}}\}\big) \\
&= (\mathsf{RFS}^\dagger \circ F)\{d_{\mathsf{id}}\} \sqcup (\mathsf{RFS}^\dagger \circ F)^2\{d_{\mathsf{id}}\} \sqcup (\mathsf{RFS}^\dagger \circ F)^3\{d_{\mathsf{id}}\}, \\
&\ldots
\end{aligned}
$$

Here we used the fact that $\mathsf{RFS}^\dagger \circ F$ preserves $\sqcup$. Note that in each step, we apply the lifted rank-synthesis algorithm $\mathsf{RFS}^\dagger$ to the analysis result of the loop body $F(A_n)$. This application of RFS keeps only termination relevant information from $F(A_n)$ and discards anything not

[3] This countable sequence reaches a limit, since it is increasing and the height of $\mathcal{A}$ is the first countable ordinal ω.

relevant to termination. To see why this is the case, proving program termination requires a set of ranking functions which overapproximate the program; the use of the RFS operator generates these ranking functions that are required to prove program termination.

One should also note that the input A is not used in this fixpoint computation at all. As the expansion of A_3 shows, the fixpoint computation effectively starts with $(\mathsf{RFS}^\dagger \circ F)\{d_{\mathsf{id}}\}$, which means the results of running the loop body once on all states. The input A, together with $\{d_{\mathsf{id}}\}$, is pre-composed later to the computed fixpoint. This change of the starting point is crucial for the soundness of our abstract interpreter, because it ensures that the analyzer overapproximates the relation between any states (not just initial states) at a loop and the following states at the same loop (so that we can apply a known termination proof rule based on disjunctively well-founded relations [50]).

Our definition does not ensure the termination of the generic abstract interpreter, because the fix operator might diverge. This divergence problem, however, will not appear in the concrete instances of the abstract interpreter considered in the chapter; we will show that all those instances terminate.

Given a program C, the abstract interpreter works as follows:

$$\begin{aligned}\textsc{Analysis}(c) \stackrel{def}{=}\ & \textbf{let } A = [\![C]\!]^\sharp(\{d_{\mathsf{id}}\}) \\ & \textbf{in if } (A \neq \top) \textbf{ then } (\textbf{return } \text{“Terminates”}) \textbf{ else } (\textbf{return } \text{“Unknown”}).\end{aligned}$$

Theorem 4.1 (Soundness)**.** *If* $\textsc{Analysis}(C)$ *returns “Terminate”, then all traces in* $[\![C]\!]$ *are finite. Hence,* C *terminates on all states.*

4.3 Soundness

In this section, we prove the soundness of our abstract interpreter, stated in Theorem 4.1.

One challenge for proving the soundness theorem is to deal with *greatest* fixpoints in the concrete semantics of loops, because our abstract interpreter uses an overapproximation of *least* fixpoints. Our first step of the proof is, therefore, to rewrite the concrete trace semantics such that the new semantics does not change the meaning of any commands, but interprets loops in terms of least fixpoints.

We define an operator repeat which maps a trace set T to the set of *infinite* traces τ satisfying the following condition: there are infinitely many finite traces $s_0\tau_0 s_1$, $s_1\tau_1 s_2$, ... in T such that

$$(\tau = s_0\tau_0 s_1\tau_1 \ldots) \wedge (\forall n. |s_n\tau_n s_{n+1}| \geq 2) \wedge (\forall n.\, s_n\tau_n s_{n+1} \in T).$$

We call a trace set T *progressing* if and only if all traces in T are of length at least 2.

Proposition 4.2 (Fixpoints Correspondence)**.** *For all functions F on $\mathcal{P}(\mathcal{T})$, if F is of the form $\lambda T.\, \mathsf{States} \cup \mathsf{seq}(T_0, T)$ for some trace set T_0, and T_0 is progressing, then*

$$(\mathsf{gfix}\, F) \quad = \quad (\mathsf{lfix}\, F) \cup \mathsf{repeat}(T_0).$$

Proof. We first prove that

$$(\mathsf{gfix}\, F) \quad \supseteq \quad (\mathsf{lfix}\, F) \cup \mathsf{repeat}(T_0).$$

We will show that $(\mathsf{lfix}\, F) \cup \mathsf{repeat}(T_0)$ is a post-fixpoint of F. Then, the required relationship will follow, because the greatest fixpoint of the monotone function F is also the greatest post-fixpoint.

We notice two facts about seq, which follow from the definitions of seq and repeat:

1. $\mathsf{seq}(T_0, \bigcup_{i \in I} T_i') = \bigcup_{i \in I} \mathsf{seq}(T_0, T_i')$ for all nonempty I.
2. $\mathsf{seq}(T_0, \mathsf{repeat}(T_0)) \supseteq \mathsf{repeat}(T_0)$.

Using these facts, we show that $(\mathsf{lfix}\, F) \cup \mathsf{repeat}(T_0)$ is a post-fixpoint of F:

$$\begin{aligned}
& F((\mathsf{lfix}\, F) \cup \mathsf{repeat}(T_0)) \\
&= \mathsf{States} \cup \mathsf{seq}(T_0, (\mathsf{lfix}\, F) \cup \mathsf{repeat}(T_0)) && (\because \text{Def. of } F) \\
&= \mathsf{States} \cup \mathsf{seq}(T_0, \mathsf{lfix}\, F) \cup \mathsf{seq}(T_0, \mathsf{repeat}(T_0)) && (\because \text{Fact 1 of seq}) \\
&= F(\mathsf{lfix}\, F) \cup \mathsf{seq}(T_0, \mathsf{repeat}(T_0)) \\
&= (\mathsf{lfix}\, F) \cup \mathsf{seq}(T_0, \mathsf{repeat}(T_0)) \\
&\supseteq (\mathsf{lfix}\, F) \cup \mathsf{repeat}(T_0) && (\because \text{Fact 2 of seq}).
\end{aligned}$$

Next, we prove

$$(\mathsf{gfix}\, F) \quad \subseteq \quad (\mathsf{lfix}\, F) \cup \mathsf{repeat}(T_0).$$

Pick a trace τ in $\mathsf{gfix}\, F$. Since $\mathsf{gfix}\, F$ is a fixpoint of F,

$$\tau \;\in\; \mathsf{States} \cup \mathsf{seq}(T_0, (\mathsf{gfix}\, F)).$$

If τ is in States, it should belong to $\mathsf{lfix}\, F$, because States is a subset of $\mathsf{lfix}\, F$. Otherwise, there are two possibilities. The first possibility is that τ is an infinite trace in T_0. In this case, τ should again belong to $\mathsf{lfix}\, F$ as well, because $T_0 \cap \mathsf{States}^\infty$ is a subset of $\mathsf{seq}(T_0, \mathsf{States})$, which is included in $\mathsf{lfix}\, F$. The second possibility is that τ can be decomposed into a finite prefix $\tau' s$ and a suffix τ'' such that

$$\tau = \tau' s \tau'' \;\; \wedge \;\; \tau' s \in T_0 \;\; \wedge \;\; |\tau' s| \geq 2 \;\; \wedge \;\; s\tau'' \in \mathsf{gfix}\, F.$$

The third conjunct $|\tau' s| \geq 2$ above is obtained from the second, using the fact that T_0 is progressing. If we apply the same reasoning to $s\tau''$ recursively, then either we generate a finitely many traces $s_1\tau_1 s_2$, $s_2\tau_2 s_3$, . . ., $s_m\tau_m$ such that

$$\begin{array}{l} \tau = s_1\tau_1 s_2 ... s_m\tau_m \;\; \wedge \\ (\forall 0 < n < m.\, |s_n\tau_n s_{n+1}| \geq 2 \;\wedge\; s_n\tau_n s_{n+1} \in T_0) \;\wedge\; s_m\tau_m \in \mathsf{lfix}\, F. \end{array}$$

This implies that τ is in the result of applying $\mathsf{seq}(T_0, -)$ to $\mathsf{lfix}\, F$ m-times; since this result is a subset of $\mathsf{lfix}\, F$, trace τ has to be in $\mathsf{lfix}\, F$. Or, there are infinitely many finite traces $s_1\tau_1 s_2$, $s_2\tau_2 s_3$, . . ., such that

$$\tau = s_1\tau_1 s_2\tau_2 s_3 \ldots \;\; \wedge \;\; (\forall n > 0.\, |s_n\tau_n s_{n+1}| \geq 2 \;\wedge\; s_n\tau_n s_{n+1} \in T_0).$$

This implies that τ belongs to $\mathsf{repeat}(T_0)$. Thus, in both cases, we have shown that trace τ is in $(\mathsf{lfix}\, F) \cup \mathsf{repeat}(T_0)$, as required. □

Using the proposition, we simplify the semantics of the loops, which we will use in the remainder of this section:

Corollary 4.3. *For all loops* `while` $B\ C$*, we have that*

$$\begin{array}{rcl} [\![\texttt{while}\ B\ C]\!] & = & \textbf{let} \;\; T_0 = \mathsf{seq}(\{ss \mid [\![B]\!]s = \mathsf{true}\},\ [\![C]\!]) \\ & & \qquad F = \lambda T.\ \mathsf{States} \cup \mathsf{seq}\big(T_0, T\big) \\ & & \textbf{in}\ \mathsf{seq}((\mathsf{lfix}\, F) \cup \mathsf{repeat}(T_0),\ \{(s,s) \mid [\![B]\!]s = \mathsf{false}\}) \end{array}$$

Proof. The trace set T_0 is progressing. Function F is of the form in Proposition 4.2, thus:

$$(\mathsf{gfix}\, F) \;\; = \;\; (\mathsf{lfix}\, F) \cup \mathsf{repeat}(T_0).$$

This allows us to replace $\mathsf{gfix}\, F)$ with $(\mathsf{lfix}\, F) \cup (\mathsf{repeat}(T_0)$ in the definition for `while`. □

The remainder of the soundness proof consists of three steps. Firstly, it builds a relational semantics, and shows how the relational semantics is related to the original trace semantics. Next, we prove that our abstract interpreter overapproximates the relational semantics. Finally, we combine the results of the previous two steps and derive the soundness of the abstract interpreter. The following three subsections explain these three steps separately.

4.3.1 Relational Semantics

The relational semantics is an abstraction of the concrete trace semantics, based on the domain of relations on States. We factor the soundness proof of the generic analyzer through the soundness of this relational semantics, because firstly this factoring simplifies the proof and secondly the relational semantics describes the limit of our generic analyzer; it is more precise than all instances of our analyzer.

The relational semantics interprets commands using the domain of relations extended with $\top$:

$$\mathsf{Rels}^\top \stackrel{def}{=} \mathcal{P}^\top(\mathsf{States} \times \mathsf{States}) \quad (= \mathcal{P}(\mathsf{States} \times \mathsf{States}) \uplus \top).$$

This domain is ordered by the subset order extended with $\top$, and forms a complete lattice. The meaning of each element in the domain is given by the concretization map γ:

$$\begin{aligned} \gamma &: \mathsf{Rels}^\top \to \mathcal{P}(\mathcal{T}) \\ \gamma(r) &\stackrel{def}{=} \begin{cases} \{\ s\tau s' \mid \tau \text{ is finite } \wedge\ s[r]s'\ \} & \text{if } r \in \mathsf{Rels} \\ \mathcal{T} & \text{otherwise.} \end{cases} \end{aligned}$$

Define a binary operator rseq on $\mathsf{Rels}^\top$, which overapproximates the sequential composition operator seq for trace sets:

$$\begin{aligned} \mathsf{rseq} &: \mathsf{Rels}^\top \times \mathsf{Rels}^\top \to \mathsf{Rels}^\top \\ \mathsf{rseq}(r, r') &\stackrel{def}{=} \begin{cases} r; r' & \text{if } r \neq \top \text{ and } r' \neq \top \\ \top & \text{otherwise.} \end{cases} \end{aligned}$$

Lemma 4.4. *For all r and r' in* $\mathsf{Rels}^\top$*,*

$$\mathsf{seq}(\gamma(r), \gamma(r')) \quad \subseteq \quad \gamma(\mathsf{rseq}(r, r')).$$

$$
\begin{array}{rcl}
(\!|C|\!) & \in & \mathsf{Rels}^{\top} \\
(\!|x := E|\!) & \stackrel{def}{=} & \{\ (s,\ s[x \mapsto [\![E]\!]s])\ \mid\ s \in \mathsf{States}\ \} \\
(\!|x := *|\!) & \stackrel{def}{=} & \{\ (s,\ s[x \mapsto r])\ \mid\ s \in \mathsf{States}\ \wedge\ r \in P\ \} \\
 & & \text{where } P \text{ is a large, finite set of rationals} \\
(\!|C_1; C_2|\!) & \stackrel{def}{=} & \mathsf{rseq}((\!|C_1|\!), (\!|C_2|\!)) \\
(\!|C_1 [] C_2|\!) & \stackrel{def}{=} & (\!|C_1|\!) \sqcup (\!|C_2|\!) \\
(\!|\texttt{if}\ B\ C_1\ C_2|\!) & \stackrel{def}{=} & \mathsf{rseq}(\ \{(s,s)\ \mid\ [\![B]\!]s = \mathsf{true}\},\ (\!|C_1|\!)\ ) \\
 & & \quad \sqcup\ \mathsf{rseq}(\ \{(s,s)\ \mid\ [\![B]\!]s = \mathsf{false}\},\ (\!|C_2|\!)\ ) \\
(\!|\texttt{while}\ B\ C|\!) & \stackrel{def}{=} & \textbf{let}\ \ r_0 = \mathsf{rseq}(\ \{(s,s)\ \mid\ [\![B]\!]s = \mathsf{true}\},\ (\!|C|\!)\ ) \\
 & & \qquad\ r = \mathsf{lfix}\ \lambda r.\ \mathsf{rseq}(\ r_0,\ \Delta_{\mathsf{States}} \sqcup r\ ) \\
 & & \textbf{in if}\ (r \neq \top\ \wedge\ \textsc{DisjWellFounded}(r)) \\
 & & \quad \textbf{then}\ \ \mathsf{rseq}(\ \Delta_{\mathsf{States}} \sqcup r,\quad \{(s,s)\ \mid\ [\![B]\!]s = \mathsf{false}\}\ ) \\
 & & \quad \textbf{else}\ \ \top
\end{array}
$$

FIGURE 4.5: Relational Semantics of Programs

Proof. If r or r' is $\top$, so is $\mathsf{rseq}(r, r')$. The lemma, then, easily follows. Suppose that both r, r' are relations on States. Pick τ from $\mathsf{seq}(\gamma(r), \gamma(r'))$. Since $\gamma(r)$ and $\gamma(r')$ both contain only finite traces, τ should be of the form $\tau' s \tau''$, where $\tau' s$ and $s\tau''$ are finite traces belonging, respectively, to $\gamma(r')$ and $\gamma(r')$. By the definition of γ, the first and last states of $\tau' s$ are related by r, and those states of $s\tau''$ are related by r'. Thus, the first and last states of τ has to be related by $r; r'$. This means that $\tau \in \gamma(\mathsf{rseq}(r, r'))$, as required. □

Call a relation $r \in \mathsf{Rels}$ *disjunctively well-founded* if and only if there are finitely many well-founded relations $r_1, \ldots, r_n$ satisfying

$$r \ \subseteq\ r_1 \cup \ldots \cup r_n.$$

Let $\textsc{DisjWellFounded}(r)$ be a predicate on Rels that holds precisely when its argument r is disjunctively well-founded. Recall that Δ_{States} is the identity relation on states. The relational semantics of commands is shown in Figure 4.5.

The soundness of the relational semantics relies on the result by Podelski and Rybalchenko, applied to traces directly. We recall the slightly modified version of 2.6 below:

Lemma 4.5 (Podelski and Rybalchenko)**.** *Let r be a disjunctively well-founded relation. For every trace τ, if $\mathsf{proj}(\tau, n)[r]\mathsf{proj}(\tau, m)$ for all $1 \leq n < m \leq |\tau|$ (where $|\tau|$ is ∞ in case that τ is infinite), the trace τ is finite.*

Proof. The proof of the lemma follows from the known proof of Podelski and Rybalchenko. We put the proof here, in order to make this thesis self-contained. Consider finitely many well-founded relations $r_1, \ldots, r_k$. Pick r such that $r \subseteq r_1 \cup \ldots \cup r_k$. For the sake of contradiction, suppose that there is an *infinite* trace τ satisfying

$$\forall n, m.\ 1 \leq n < m \implies \mathsf{proj}(\tau, n)\,[r]\,\mathsf{proj}(\tau, m).$$

Define a function f that maps each pair (n, m) of indices with $n < m$ to an integer i such that $\mathsf{proj}(\tau, n)[r_i]\mathsf{proj}(\tau, m)$. Such a function should be well-defined, because $\mathsf{proj}(\tau, n)[r]\mathsf{proj}(\tau, m)$ and $r \subseteq r_1 \cup \ldots \cup r_k$. Since there are only finitely many r_i's, by infinite Ramsey's theorem, there exist infinite subtrace τ' of τ and r_l such that

$$\forall n, m.\ (1 \leq n < m) \implies \mathsf{proj}(\tau', n)[r_l]\mathsf{proj}(\tau', m).$$

Note that for all indices n, states $\mathsf{proj}(\tau', n)$ and $\mathsf{proj}(\tau', n{+}1)$ are related by r_l. Thus, r_l cannot be well-founded, which contradicts our assumption on r_l. □

Proposition 4.6 (Soundness of Relational Semantics)**.** *For all commands C,*

$$[\![C]\!] \ \subseteq\ \gamma(\langle\!|C|\!\rangle).$$

Proof. Define a relation $\mathcal{L} \subseteq \mathcal{P}(\mathcal{T}) \times \mathsf{Rels}^{\top}$ by

$$T[\mathcal{L}]r \iff T \subseteq \gamma(r).$$

With this relation, we can rewrite the main claim of this proposition by

$$\forall C.\ [\![C]\!]\,[\mathcal{L}]\,\langle\!|C|\!\rangle.$$

We will show this in the proof.

Our proof relies on the four important properties of $\mathcal{L}$. The first is that $\mathcal{L}$ is closed, downward for the left parameter and upward for the right parameter:

$$(T' \subseteq T \wedge T[\mathcal{L}]r \wedge r \sqsubseteq r') \implies T'[\mathcal{L}]r'.$$

The other three are the preservation of $\mathcal{L}$ by various operators. Relation $\mathcal{L}$ is preserved by the interpretations of atomic commands in the trace and relational semantics, the sequential composition operators for $\mathcal{P}(\mathcal{T})$ and $\mathsf{Rels}^\top$, and the join operators for $\mathcal{P}(\mathcal{T})$ and $\mathsf{Rels}^\top$. That is,

1. $[\![a]\!][\mathcal{L}](\!|a|\!)$ for all atomic commands a;
2. for all $T, T' \in \mathcal{P}(\mathcal{T})$ and $r, r' \in \mathsf{Rels}^\top$,

$$T[\mathcal{L}]r \wedge T'[\mathcal{L}]r' \implies \mathsf{seq}(T, T')[\mathcal{L}]\mathsf{rseq}(r, r');$$

3. for all families $\{T_i\}_{i\in I}$ and $\{r_i\}_{i\in I}$ with the same index set I,

$$(\forall i \in I.\, T_i[\mathcal{L}]r_i) \implies \big(\bigcup_{i\in I} T_i\big)\,[\mathcal{L}]\,\big(\bigsqcup_{i\in I} r_i\big).$$

Now, we prove the main claim of the proposition, by induction on the structure of C. All cases except the `while` loops follow from the four properties of $\mathcal{L}$ and the induction hypothesis. For instance, consider the case that C is a conditional statement $(\texttt{if}\, B\, C_1\, C_2)$. By the induction hypothesis, $[\![C_1]\!][\mathcal{L}](\!|C_1|\!)$ and $[\![C_1]\!][\mathcal{L}](\!|C_1|\!)$. Furthermore,

$$\{ss \mid [\![B]\!]s = \mathsf{true}\}\ [\mathcal{L}]\ \{(s,s) \mid [\![B]\!]s = \mathsf{true}\} \quad \text{and}$$
$$\{ss \mid [\![B]\!]s = \mathsf{false}\}\ [\mathcal{L}]\ \{(s,s) \mid [\![B]\!]s = \mathsf{false}\}.$$

Thus, we have that

$$\mathsf{seq}(\ \{ss \mid [\![B]\!]s = \mathsf{true}\},\ [\![C_1]\!]\ )\ \cup\ \mathsf{seq}(\ \{ss \mid [\![B]\!]s = \mathsf{false}\},\ [\![C_2]\!]\ )$$
$$[\mathcal{L}]$$
$$\mathsf{rseq}(\ \{ss \mid [\![B]\!]s = \mathsf{true}\},\ (\!|C_1|\!)\ )\ \sqcup\ \mathsf{rseq}(\ \{ss \mid [\![B]\!]s = \mathsf{false}\},\ (\!|C_2|\!)\ ).$$

because both the sequencing operators and the join operators preserve $\mathcal{L}$. This gives the proposition for $(\texttt{if}\, B\, C_1\, C_2)$.

Finally, consider the remaining case that C is $\texttt{while}\, B\, C'$. Let

$$T_0 = \mathsf{seq}(\{ss \mid [\![B]\!]s = \mathsf{true}\},\ [\![C']\!]), \qquad F = \lambda T.\, \mathsf{States} \cup \mathsf{seq}(T_0, T),$$
$$r_0 = \mathsf{rseq}(\{(s,s) \mid [\![B]\!]s = \mathsf{true}\},\ (\!|C'|\!)), \quad G = \lambda r.\, \Delta_{\mathsf{States}} \sqcup \mathsf{rseq}(r_0, r).$$

where Δ_{States} is the identity relation on States. Define G' to be $\lambda r.\, \mathsf{rseq}(r_0, \Delta_{\mathsf{States}} \sqcup r)$ and r_i to be $\mathsf{lfix}\, G'$. We will prove that

1. $\mathsf{lfix}\ F\ [\mathcal{L}]\ \mathsf{lfix}\ G$;
2. $(\mathsf{lfix}\, G) \sqsubseteq (\Delta_{\mathsf{States}} \sqcup r_i)$;
3. $\mathsf{repeat}(T_0) = \emptyset$ if r_i is disjunctively well-founded.

The proposition follows from these three. To see this, first consider the case that r_i is $\top$ or it is not disjunctively well-founded. In this case, $(\!|C|\!)$ is $\top$, so $T_1[\mathcal{L}](\!|C|\!)$ for all trace sets T_1. The proposition follows from this. Next consider the other case that r_i is a disjunctively well-founded relation on states. By the third property above, $\mathsf{repeat}(T_0)[\mathcal{L}]\emptyset$. Since $\mathcal{L}$ is upward closed on the right and it preserves the join operators,

$$\big(\mathsf{lfix}\ F\ \cup\ \mathsf{repeat}(T_0)\big)\ [\mathcal{L}]\ \big(\Delta_{\mathsf{States}}\ \sqcup\ r_i\ \sqcup\ \emptyset\big).$$

This implies the required

$$\begin{gathered}\mathsf{seq}\Big(\mathsf{lfix}\ F\ \cup\ \mathsf{repeat}(T_0),\ \ \{ss\ \mid\ [\![B]\!]s = \mathsf{false}\}\Big)\\ [\mathcal{L}]\\ \mathsf{rseq}\Big(\Delta_{\mathsf{States}}\ \sqcup\ \mathsf{lfix}\, G',\ \ \{(s,s)\ \mid\ [\![B]\!]s = \mathsf{false}\}\Big),\end{gathered}$$

because $\{ss\ \mid\ [\![B]\!]s = \mathsf{false}\}\ [\mathcal{L}]\ \{(s,s)\ \mid\ [\![B]\!]s = \mathsf{false}\}$ and the sequential composition operators preserve $\mathcal{L}$.

The first about the least fixpoints of F and G is a standard result. It holds because (1) F and G map $\mathcal{L}$-related values to $\mathcal{L}$-related values; (2) the empty trace set $\emptyset$ and the empty relation $\emptyset$ are related by $\mathcal{L}$; (3) both finitary and infinitary join operators preserve $\mathcal{L}$; and (4) F and G are continuous. The second holds because $\Delta_{\mathsf{States}} \sqcup r_i$ is a fixpoint of G:

$$\begin{aligned} G(\Delta_{\mathsf{States}} \sqcup r_i) &= \Delta_{\mathsf{States}} \sqcup \mathsf{seq}(r_0, \Delta_{\mathsf{States}} \sqcup r_i) && (\because \text{Def. of } G)\\ &= \Delta_{\mathsf{States}} \sqcup \mathsf{seq}(r_0, \Delta_{\mathsf{States}} \sqcup \mathsf{lfix}\, G') && (\because \text{Def. of } r_i)\\ &= \Delta_{\mathsf{States}} \sqcup \mathsf{lfix}\, G' && (\because G' = \lambda r.\, \mathsf{seq}(r_0, \Delta_{\mathsf{States}} \sqcup r))\\ &= \Delta_{\mathsf{States}} \sqcup r_i && (\because \text{Def. of } r_i). \end{aligned}$$

For the third, assume that r_i is disjunctively well-founded. For the sake of contradiction, suppose that $\mathsf{repeat}(T_0)$ is not empty. Pick a trace τ in $\mathsf{repeat}(T_0)$. By the definition of repeat, there are

infinitely many finite traces $s_0\tau_0 s_1$, $s_1\tau_1 s_2$, ... such that

$$\tau = s_0\tau_1 s_1\tau_2 s_2 ... \ \wedge \ (\forall n.\ |s_n\tau_n s_{n+1}| \geq 2 \wedge s_n\tau_n s_{n+1} \in T_0).$$

We will prove that the subtrace $s_0 s_1 \ldots$ of τ satisfy

$$\forall n, m.\ 1 \leq n < m \implies s_n[r_i]s_m.$$

This gives the desired contradiction; it implies that τ is finite because r_i is disjunctively well-founded (Lemma 4.5), but τ belongs to $\mathsf{repeat}(T_0)$ that contains only infinite traces. Let H be $\lambda T.\, \mathsf{seq}(T_0, T)$. Recall that $\mathsf{States}[\mathcal{L}]\Delta_{\mathsf{States}}$. Notice that functions H and G' maps $\mathcal{L}$-related values to $\mathcal{L}$-related ones, because of the induction hypothesis and the preservation and closedness properties of $\mathcal{L}$. Thus, we have that

$$\forall n.\ H^n(\mathsf{States})\,[\mathcal{L}]\,G'^n(\Delta_{\mathsf{States}}).$$

Also note that for all n, m with $1 \leq n < m$, finite trace

$$s_n\tau_{n+1} \ldots \tau_m s_m$$

is in $H^{m-n}(\mathsf{States})$. From these two observations and the definition of $\mathcal{L}$, it follows that

$$s_n\tau_{n+1}s_{n+1}\tau_{n+1} \ldots \tau_m s_m \ \in \ \gamma(G'^{m-n}(\Delta_{\mathsf{States}})).$$

The right hand side of this inequality is the same as $\gamma(G'^{m-n}(\emptyset))$, so that it is a subset of $\gamma(\mathsf{lfix}\, G')$. Therefore,

$$s_n\tau_{n+1}s_{n+1}\tau_{n+2} \ldots \tau_m s_m \ \in \ \gamma(r_i),$$

which means $s_n[r_i]s_m$, as desired. □

4.3.2 Overapproximation Result

Consider a generic abstract interpreter. Let $\mathcal{A}$ be the abstract domain of the abstract interpreter. For each $A \in \mathcal{A}$, define $\gamma_r(A) \in \mathsf{Rels}^\top$ by

$$\gamma_r(A) \stackrel{def}{=} \begin{cases} \top & \text{if } A = \top \\ \bigcup\{\gamma_r(d) \mid d \in A\} & \text{otherwise.} \end{cases}$$

Proposition 4.7. *The generic abstract interpreter overapproximates the relational semantics. That is, for all commands C and abstract values $A \in \mathcal{A}$,*

$$\mathsf{rseq}(\gamma_r(A), (\!|C|\!)) \quad \sqsubseteq \quad \gamma_r([\![C]\!]^\sharp(A)).$$

Proof. We prove the proposition by induction on the structure of C. Since $[\![C]\!]^\sharp$ preserves $\top$, when A is $\top$, the inequality of the proposition holds. Thus, in this proof, we focus on non-$\top$ abstract elements. Pick an abstract element A in $\mathcal{A}-\{\top\}$. Consider the case that C is an atomic command a. View the concrete semantics $[\![a]\!]$ of a as a relation on states. Then,

$$\begin{array}{rcll}
\mathsf{rseq}(\gamma_r(A), (\!|a|\!)) & = & \gamma_r(A); [\![a]\!] & (\because \text{Def. of } \mathsf{rseq}) \\
& = & (\bigcup\{\gamma_r(d) \mid d \in A\}); [\![a]\!] & (\because \text{Def. of } \gamma_r(A)) \\
& = & \bigcup\{\gamma_r(d); [\![a]\!] \mid d \in A\} & (\because -; r \text{ distributes over } \cup) \\
& \sqsubseteq & \bigsqcup\{\gamma_r(d') \mid d \in A \wedge d' \in \mathsf{trans}(a)(d)\} & (\because \text{Condition on } \mathsf{trans}) \\
& = & \gamma_r(\mathsf{trans}(a)^\dagger A) & \\
& = & \gamma_r([\![a]\!]^\sharp A) & (\because \text{Def. of the abs. interpreter}).
\end{array}$$

The cases that C is a sequential composition, a non-deterministic choice or a conditional statement follow easily from the induction hypothesis, the associativity of rseq, the preservation of $\sqcup$ by rseq and the soundness condition on filter. In the below, we prove the cases of sequential composition and conditional statement:

$$\begin{array}{rcll}
\mathsf{rseq}(\gamma_r(A), (\!|C_1; C_2|\!)) & \sqsubseteq & \mathsf{rseq}(\gamma_r(A), \mathsf{rseq}((\!|C_1|\!), (\!|C_2|\!))) & \\
& \sqsubseteq & \mathsf{rseq}(\mathsf{rseq}(\gamma_r(A), (\!|C_1|\!)), (\!|C_2|\!)) & (\because \text{Associativity of } \mathsf{rseq}) \\
& \sqsubseteq & \mathsf{rseq}(\gamma_r([\![C_1]\!]^\sharp(A)), (\!|C_2|\!)) & (\because \text{Ind. Hypo.}) \\
& \sqsubseteq & \gamma_r([\![C_2]\!]^\sharp([\![C_1]\!]^\sharp(A))) & (\because \text{Ind. Hypo.}).
\end{array}$$

Let $b_t = \{(s,s) \mid [\![B]\!]s = \mathsf{true}\}$ and $b_f = \{(s,s) \mid [\![B]\!]s = \mathsf{false}\}$.

$$
\begin{array}{lll}
& \mathsf{rseq}(\gamma_r(A),\ (\!|\texttt{if}\ B\ C_1\ C_2|\!)) & \\
\sqsubseteq & \mathsf{rseq}(\gamma_r(A),\ \mathsf{rseq}(b_t, (\!|C_1|\!)) \sqcup \mathsf{rseq}(b_f, (\!|C_2|\!))) & \\
= & \mathsf{rseq}(\gamma_r(A),\ \mathsf{rseq}(b_t, (\!|C_1|\!)))\ \sqcup\ \mathsf{rseq}(\gamma_r(A),\ \mathsf{rseq}(b_f, (\!|C_2|\!))) & (\because \text{rseq preserves } \sqcup) \\
= & \mathsf{rseq}(\mathsf{rseq}(\gamma_r(A), b_t), (\!|C_1|\!))\ \sqcup\ \mathsf{rseq}(\mathsf{rseq}(\gamma_r(A), b_f), (\!|C_2|\!)) & (\because \text{Associativity of rseq}) \\
= & \mathsf{rseq}(\gamma_r(\mathsf{filter}^\dagger_B(A)), (\!|C_1|\!))\ \sqcup\ \mathsf{rseq}(\gamma_r(\mathsf{filter}^\dagger_{\neg B}(A)), (\!|C_2|\!)) & (\because \text{Soundness of filter}) \\
\sqsubseteq & \gamma_r([\![C_1]\!]^\sharp(\mathsf{filter}^\dagger_B(A))) \sqcup \gamma_r([\![C_2]\!]^\sharp(\mathsf{filter}^\dagger_{\neg B}(A))) & (\because \text{Ind. Hypo.}) \\
\sqsubseteq & \gamma_r\big([\![C_1]\!]^\sharp(\mathsf{filter}^\dagger_B(A)) \sqcup [\![C_2]\!]^\sharp(\mathsf{filter}^\dagger_{\neg B}(A))\big) & (\because \text{Monotonicity of } \gamma_r) \\
= & \gamma_r([\![\texttt{if}\ B\ C_1\ C_2]\!]^\sharp(A)). &
\end{array}
$$

Now, it remains to prove the inductive step for the loop case, i.e., $C = \texttt{while}\, B\, C'$. Let

$$
\begin{array}{rcl}
b_t & = & \{(s,s)\ \mid\ [\![B]\!]s = \mathsf{true}\}, \\
r_0 & = & \mathsf{rseq}(b_t, (\!|C'|\!)), \\
G & = & \lambda r.\, \mathsf{rseq}(r_0, \Delta_{\mathsf{States}} \sqcup r), \\
r_i & = & \mathsf{lfix}\, G, \\
H & = & \lambda A.\, (\mathsf{RFS}^\dagger \circ C' \circ \mathsf{filter}^\dagger_B)(\{d_{\mathsf{id}}\} \sqcup A), \\
A_i & = & \mathsf{fix}\, H
\end{array}
$$

where fix is the fixpoint operator of the abstract interpreter. First, we prove that

$$\Delta_{\mathsf{States}} \sqsubseteq \gamma_r(\{d_{\mathsf{id}}\}) \quad \wedge \quad r_i \sqsubseteq \gamma_r(A_i).$$

The first conjunct follows from the soundness condition on d_{id}. For the second conjunct, we will show that the least fixpoint r_i of G is also the least fixpoint of $K = \lambda r.\mathsf{rseq}(\Delta_{\mathsf{States}} \sqcup r, r_0)$ and that $\gamma_r(A_i)$ is a pre-fixpoint of K. Since K is a monotone function on a complete lattice, this implies that the least fixpoint r_i of K is less than or equal to $\gamma_r(A_i)$. To show the coincidence between least fixpoints of G and K, we note that both G and K are continuous, so their least fixpoints can be computed by the limit of two countable sequences $\sqcup_{n\geq 0} G^n(\{\})$ and $\sqcup_{n\geq 0} K^n(\{\})$. Let L be $\lambda r.\mathsf{rseq}(r_0, r)$. Then, for all $k \geq 0$, we have that $\mathsf{rseq}(r_0, L^k(\Delta_{\mathsf{States}})) = L^{k+1}(\Delta_{\mathsf{States}})$ and also that $\mathsf{rseq}(L^k(\Delta_{\mathsf{States}}), r_0) = L^{k+1}(\Delta_{\mathsf{States}})$. (The second equality can be proved by induction on k.) Using induction on n, we prove that for all $n \geq 0$,

$$G^n(\{\}) \ =\ K^n(\{\}) \ =\ \sqcup_{1\leq k\leq n} L^k(\Delta_{\mathsf{States}}).$$

The base case is $\{\} = \{\}$, so it holds. The inductive case holds as well, because

$$\begin{aligned}
G^{n+1}(\{\}) &= \mathsf{rseq}(r_0, \Delta_{\mathsf{States}} \sqcup G^n(\{\})) \\
&= \mathsf{rseq}(r_0, \Delta_{\mathsf{States}} \sqcup (\sqcup_{1\leq k\leq n} L^k(\Delta_{\mathsf{States}}))) \\
&= \mathsf{rseq}(r_0, \Delta_{\mathsf{States}}) \sqcup (\sqcup_{1\leq k\leq n} \mathsf{rseq}(r_0, L^k(\Delta_{\mathsf{States}}))) \\
&= \mathsf{rseq}(r_0, \Delta_{\mathsf{States}}) \sqcup (\sqcup_{1\leq k\leq n} L^{k+1}(\Delta_{\mathsf{States}})) \\
&= L(\Delta_{\mathsf{States}}) \sqcup (\sqcup_{1\leq k\leq n} L^{k+1}(\Delta_{\mathsf{States}})) \\
&= (\sqcup_{1\leq k\leq n+1} L^k(\Delta_{\mathsf{States}})),
\end{aligned}$$

and

$$\begin{aligned}
K^{n+1}(\{\}) &= \mathsf{rseq}(\Delta_{\mathsf{States}} \sqcup K^n(\{\}), r_0) \\
&= \mathsf{rseq}(\Delta_{\mathsf{States}} \sqcup (\sqcup_{1\leq k\leq n} L^k(\Delta_{\mathsf{States}})),\ r_0) \\
&= \mathsf{rseq}(\Delta_{\mathsf{States}},\ r_0) \sqcup (\sqcup_{1\leq k\leq n} \mathsf{rseq}(L^k(\Delta_{\mathsf{States}}), r_0)) \\
&= \mathsf{rseq}(\Delta_{\mathsf{States}},\ r_0) \sqcup (\sqcup_{1\leq k\leq n} L^{k+1}(\Delta_{\mathsf{States}})) \\
&= \mathsf{rseq}(r_0,\ \Delta_{\mathsf{States}}) \sqcup (\sqcup_{1\leq k\leq n} L^{k+1}(\Delta_{\mathsf{States}})) \\
&= L(\Delta_{\mathsf{States}}) \sqcup (\sqcup_{1\leq k\leq n} L^{k+1}(\Delta_{\mathsf{States}})) \\
&= (\sqcup_{1\leq k\leq n+1} L^k(\Delta_{\mathsf{States}})).
\end{aligned}$$

We have just shown that the least fixpoints of F and G are the limits of the same countable sequence. Thus, they must be the same. We move on to the proof that $\gamma_r(A_i)$ is a pre-fixpoint of K.

$$\begin{aligned}
K(\gamma_r(A_i)) &= \mathsf{rseq}(\Delta_{\mathsf{States}} \sqcup \gamma_r(A_i), \mathsf{rseq}(b_t, (\!|C'|\!))) && (\because \text{Def. of } K) \\
&\sqsubseteq \mathsf{rseq}(\gamma_r(\{d_{\mathsf{id}}\}) \sqcup \gamma_r(A_i), \mathsf{rseq}(b_t, (\!|C'|\!))) && (\because \Delta_{\mathsf{States}} \sqsubseteq \gamma_r(\{d_{\mathsf{id}}\})) \\
&\sqsubseteq \mathsf{rseq}(\gamma_r(\{d_{\mathsf{id}}\} \sqcup A_i), \mathsf{rseq}(b_t, (\!|C'|\!))) && (\because \gamma_r \text{ is monotone}) \\
&\sqsubseteq \mathsf{rseq}(\mathsf{rseq}(\gamma_r(\{d_{\mathsf{id}}\} \sqcup A_i), b_t), (\!|C'|\!)) && (\because \text{Associativity of } \mathsf{rseq}) \\
&\sqsubseteq \mathsf{rseq}(\gamma_r(\mathsf{filter}_B(\{d_{\mathsf{id}}\} \sqcup A_i)), (\!|C'|\!)) && (\because \text{Soundness of } \mathsf{filter}_B) \\
&\sqsubseteq \gamma_r((\llbracket C' \rrbracket^\sharp \circ \mathsf{filter}_B)(\{d_{\mathsf{id}}\} \sqcup A_i)) && (\because \text{Ind. Hypo}) \\
&\sqsubseteq \gamma_r((\mathsf{RFS}^\dagger \circ \llbracket C' \rrbracket^\sharp \circ \mathsf{filter}_B^\dagger)(\{d_{\mathsf{id}}\} \sqcup A_i)) && (\because \text{Soundness of } \mathsf{RFS}) \\
&= \gamma_r(H(A_i)) && (\because \text{Def. of } H) \\
&\sqsubseteq \gamma_r(A_i) && (\because A_i = \mathsf{fix}\, H, \text{ and Condition on fix}).
\end{aligned}$$

Next, we show that if A_i is not $\top$, r_i has to be a disjunctively well-founded relation on states. To see this, recall that $\mathsf{fix}\, H$ is in the image of $\mathsf{RFS}^\dagger$, and suppose that $A_i (= \mathsf{fix}\, H)$ is not $\top$. Then, A_i should be a finite set of d's, each of which denotes a well-founded relation via γ_r. Thus,

$\gamma_r(A_i)$ is a finite union of well-founded relations. From this and $r_i \sqsubseteq \gamma_r(A_i)$, it follows that r_i is disjunctively well-founded.

Finally, we prove the loop case. If A_i is $\top$, so is $[\![\texttt{while}\, B\, C']\!]^\sharp A$. Thus, the proposition holds. Assume that A_i is not $\top$. Then, by what we have just shown, r_i is a disjunctively well-founded relation. Using this and letting $b_f = \{(s,s) \mid [\![B]\!]s = \mathsf{false}\}$, we prove the loop case:

$$
\begin{aligned}
&\mathsf{rseq}(\gamma_r(A), (\!|\texttt{while}\, B\, C'|\!)) \\
&= \mathsf{rseq}(\gamma_r(A), \mathsf{rseq}(\Delta_{\mathsf{States}} \cup r_i, b_f)) && (\because r_i \text{ is disj. well-founded}) \\
&\sqsubseteq \mathsf{rseq}(\gamma_r(A), \mathsf{rseq}(\gamma_r(\{d_{\mathsf{id}}\}) \sqcup \gamma_r(A_i), b_f)) && (\because \text{Mono. of } \mathsf{rseq}) \\
&\sqsubseteq \mathsf{rseq}(\gamma_r(A), \mathsf{rseq}(\gamma_r(\{d_{\mathsf{id}}\} \sqcup A_i), b_f)) && (\because \gamma_r \text{ is monotone}) \\
&\sqsubseteq \mathsf{rseq}(\gamma_r(A), \gamma_r(\mathsf{filter}_{\neg B}(\{d_{\mathsf{id}}\} \sqcup A_i))) && (\because \text{Soundness of } \mathsf{filter}_{\neg B}) \\
&\sqsubseteq \gamma_r(\mathsf{comp}^\dagger(A, \mathsf{filter}_{\neg B}(\{d_{\mathsf{id}}\} \sqcup A_i))) && (\because \text{Soundness of } \mathsf{comp}) \\
&= \gamma_r([\![\texttt{while}\, B\, C']\!]^\sharp(A)). && (\because \text{Def. of the abstract interpreter}).
\end{aligned}
$$

□

4.3.3 Proof of Theorem 4.1

We can now prove Theorem 4.1 easily. First, note that the generic analyzer overapproximates the concrete trace semantics. For all commands c,

$$
\begin{aligned}
[\![c]\!] &\sqsubseteq \gamma((\!|C|\!)) && (\because \text{Prop. 4.6}) \\
&= \gamma(\mathsf{rseq}(\Delta_{\mathsf{States}}, (\!|C|\!))) && (\because \text{Def. of } \mathsf{rseq}) \\
&\sqsubseteq \gamma(\gamma_r([\![C]\!]^\sharp\{d_{\mathsf{id}}\})) && (\because \text{Prop. 4.7}) \\
&= \gamma([\![C]\!]^\sharp\{d_{\mathsf{id}}\}).
\end{aligned}
$$

We use two kinds of γ above; γ in the last line is a map from $\mathcal{A}$ to the sets of traces, and γ in all the other places is a map from $\mathsf{Rels}^\top$ to the set of traces.

Observe that $\gamma(A)$ can contain infinite traces only when A is $\top$. By combining these two observations, we can conclude that if $\textsc{Analysis}(C)$ returns "Terminates", $[\![C]\!]$ does not contain any infinite traces, that is, C terminates.

4.4 Linear Rank Abstraction

The linear rank abstraction is an instance of our generic abstract interpreter, parametrized by the domain of linear constraints and a linear ranking synthesis algorithm RankFinder [48].

Let r represent rational numbers. Consider constraints φ defined by the grammar below:

$$\begin{aligned} E &::= x \mid {`}x \mid x' \mid r \mid E + E \mid r \times E \\ P &::= E = E \mid E \neq E \mid E < E \mid E > E \mid E \leq E \mid E \geq E \\ \varphi &::= P \mid \mathsf{true} \mid \varphi \wedge \varphi \end{aligned}$$

This grammar ensures that all the constraints are the conjunction of linear constraints. Note that a constraint can have three kinds of variables; a normal variable x denoting the current value of program variable x; a pre-primed variable ${`}x$ storing the initial value of $x$; post-primed variables $y'$ that usually denotes values which at some moment variables during computation. We assume that there are finitely many normal variables ($\mathsf{Vars}$) and finitely many pre-primed variables (${`}\mathsf{Vars}$), and that there is a one-to-one correspondence between these two kinds of variables. For post-primed variables, however, we assume an infinite set.

Each constraint denotes a relation on States. For each state s, let ${`}s$ be a function from ${`}\mathsf{Vars}$ to $\mathsf{Rationals}$ such that for every pre-primed variable ${`}x$, ${`}s({`}x)$ is $s(x)$ for the corresponding normal variable x. The meaning function γ_r of constraints φ is defined as follows:

$$\gamma_r(\varphi) \stackrel{def}{=} \{(s_0, s_1) \mid ({`}s_0, s_1 \models \exists X'.\varphi)\}$$

where X' is the set of post-primed variables in φ and $\models$ is the usual satisfaction relation in first-order logic. Note that all post-primed variables in the constraint φ are implicitly existentially-quantified. The post-primed variables are logical variables which may relate the pre-primed and non-primed variables in an assertion. Thus the post-primed variables are crucial in order for us to maintain some relationship between the value of a variable in the current state and the previous state.

The linear rank abstraction uses the set of constraints φ as the parameter set D of the generic abstract interpreter. The identity element d_{id} is the identity relation

$$d_{\mathsf{id}} \stackrel{def}{=} \bigwedge_{x \in \mathsf{Vars}} {`}x{=}x.$$

Assume that we are given an enumeration $x_0, \ldots, x_n$ of all program variables in Vars. Call an expression E *normalized,* when (1) E does not contain any pre or post primed variables and (2) it is of the form $a_{i_0} \times x_{i_0} + \ldots a_{i_k} \times x_{i_k} + a$ with $a_{i_0} = 1$ or -1 and $i_0 < i_1 \ldots < i_k$. Note that in a normalized expression E, the coefficient of the first variable in E according to the given

enumeration is 1 or -1. Conceptually, RANKFINDER implements a function of the type:[4]

$$D \to (\{(E, r) \mid E \text{ is normalized and } r \text{ is a positive rational}\}) \uplus \{\top\}.$$

The output $\top$ indicates that the algorithm fails to discover a ranking function, because (the implementation of) the algorithm is incomplete or the input constraint defines a non-well-founded relation between pre-primed variables and normal variables. The other output (E, r) means that the algorithm succeeds in finding a ranking function that overapproximates the given constraint. Concretely, for a normalized expression E and a positive rational r, let

$$T_{E,r} \stackrel{def}{=} (`E \geq 0 \ \wedge \ `E - r \geq E),$$

where expression $`E$ is $E$ with all normal variables $x$ replaced by corresponding pre-primed variables $`x$. The output (E, r) of RANKFINDER(φ) means that

$$(\exists X'.\varphi) \implies T_{E,r}$$

where X' is the set of all post-primed variables in φ.

Assume that we have chosen a fixed positive rational dec that is very small (in particular smaller than 1). Using RANKFINDER and dec, we define the operator RFS as follows:

$$\mathsf{RFS}(\varphi) \stackrel{def}{=} \begin{cases} \{\} & \text{if } \varphi \vdash \mathsf{false} \\ \{T_{E,\mathsf{dec}}\} & \text{else if RANKFINDER}(\varphi){=}(E, r) \text{ and } r \geq \mathsf{dec} \\ \top & \text{otherwise} \end{cases}$$

where $\vdash$ is a sound (but not necessarily complete) theorem prover. Note that the result of RFS is always of the form $T_{E,\mathsf{dec}}$, so the second subscript of T is not necessary. In the rest of this chapter, we write T_E for $T_{E,\mathsf{dec}}$.

The abstract transfer functions for atomic commands and filter_B are defined following Floyd's strongest postcondition semantics:

$$\begin{aligned} [\![x := *]\!]^\sharp\varphi &\stackrel{def}{=} \{\varphi[x'/x]\} \ \ (x' \text{ is fresh}) \\ [\![x := E]\!]^\sharp\varphi &\stackrel{def}{=} \{\varphi[x'/x] \wedge x{=}(E[x'/x])\} \ \ (x' \text{ is fresh}) \\ \mathsf{filter}_B(\varphi) &\stackrel{def}{=} \textbf{if } (\varphi \wedge B \vdash \mathsf{false}) \textbf{ then } \{\} \\ &\qquad \textbf{else } \{\varphi_0, \ldots, \varphi_n \mid \varphi_0 \vee \ldots \vee \varphi_n = \mathsf{norm}(\varphi \wedge B)\}. \end{aligned}$$

[4] Usually the implementation of linear rank synthesis returns a tuple (E, r, b) where E is an expression without any pre or post primed variable whose value is decreasing, r is a decrement, and b is a lower bound of E. Our abstract interpreter picks the absolute value a of the coefficient of the first variable x_i in E, transforms E/a to a normal form E', and regards $(E' - b/a, r/a)$ as an output from RANKFINDER.

Here norm is the standard transformation that takes a formula in the propositional logic and transforms the formula to disjunctive normal form. Substitution for $E[x'/x]$ is defined as:

$$\begin{aligned} {}'x[x'/x] &= {}'x \\ x'[x'/x &= x' \\ r[x'/x] &= r \\ x[x'/x] &= x' \\ (E_1 + E_2)[x'/x] &= E_1[x'/x] + E_2[x'/x] \\ (r \times E) &= r \times E[x'/x] \end{aligned}$$

For expressions P substitution is defined as:

$$(E_1 \; op \; E_2)[x'/x] = E_1[x'/x] \; op \; E_2[x'/x]$$

where $op \in \{=, \neq, <, >, \leq, \geq\}$, and for φ substitution is defined as:

$$\begin{aligned} \mathsf{true}[x'/x] &= \mathsf{true} \\ (\varphi_1 \wedge \varphi_2)[x'/x] &= \varphi_1[x'/x] \wedge \varphi_2[x'/x] \end{aligned}$$

Next, we define the abstract composition comp. Let fresh be an operator on constraints φ that renames all post-primed variables fresh. Let 'Vars be the set of pre-primed variables. The abstract composition is defined as follows

$$\mathsf{comp}(\varphi_0, \varphi_1) \stackrel{def}{=} \mathbf{let} \; \big(\varphi_2 = \mathsf{fresh}(\varphi_1)\big) \; \mathbf{in} \; \Big(\varphi_0[Y'/\mathsf{Vars}] \wedge \varphi_2[Y'/{}'\mathsf{Vars}]\Big).$$

The variable set Y' in the definition denotes a set of fresh post-primed variables, that has as many elements as Vars. The two substitutions there replace a normal variable x and the corresponding pre-primed variable $'x$ by the same post-primed variable x'.

We have a lemma which relates the comp operator and relational composition.

Lemma 4.8.

$$\gamma_r(\varphi_0); \gamma_r(\varphi_1) = \gamma_r(\mathsf{comp}(\varphi_0, \varphi_1))$$

Proof. We will first show that $\gamma_r(\varphi_0); \gamma_r(\varphi_1) \subseteq (\gamma_r(\mathsf{comp}(\varphi_0, \varphi_1)))$.
Let $(s_0, s_1) \in \gamma_r(\varphi_0); \gamma_r(\varphi_1)$. This implies that there exists a state s_2 such that:

$$(s_0, s_2) \in \gamma_r(\varphi_0)$$
$$(s_2, s_1) \in \gamma_r(\varphi_1)$$

By the definition of γ_r:

$$('s_0, s_2) \models \exists X'.\varphi_0$$
$$('s_2, s_1) \models \exists X'.\varphi_1$$

which implies that

$$('s_o, s_1) \models \exists X'.\varphi_0[Y'/Vars] \wedge \varphi_2[Y'/`Vars]$$

where $\varphi_2 = \mathsf{fresh}(\varphi_1)$. This step is justified as the renaming of the variable corresponds to finding a common state between the two relations represented by φ_0 and φ_1. This then gives us the necessary result, by the definition of γ_r and comp.

$$(s_0, s_1) \in \gamma_r(\mathsf{comp}(\varphi_0, \varphi_1))$$

Now we will show that $\gamma_r(\mathsf{comp}(\varphi_0, \varphi_1)) \subseteq \gamma_r(\varphi_0); \gamma_r(\varphi_1)$. Let $(s_0, s_1) \in \gamma_r(\mathsf{comp}(\varphi_0, \varphi_1))$. This implies that:

$$(`s_0, s_1) \models \exists X'.\varphi_0[Y'/Vars] \wedge \varphi_2[Y'/`Vars]$$

where $\varphi_2 = \mathsf{fresh}(\varphi_1)$. This implies that there exists a state s_2 such that

$$(`s_0, s_2) \models \exists X'\varphi_0$$
$$(`s_2, s_1) \models \exists X'.\varphi_1$$

The result then follows from these two facts to give:

$$(s_0, s_2) \in \gamma_r(\varphi_0)$$
$$(s_2, s_1) \in \gamma_r(\varphi_1)$$

which gives $(s_0, s_1) \in \gamma_r(\varphi_0); \gamma_r(\varphi_1)$ are required. □

Finally, we specify a fix operator. For each function $(\mathsf{RFS}^\dagger \circ F)$ on sets of constraints φ, let $\{G_n\}_n$ be the standard fixpoint iteration sequence: $G_0 = \{\}$ and $G_{n+1} = G_n \sqcup (\mathsf{RFS}^\dagger \circ F)(G_n)$. Given G, our fix operator returns the first G_n such that

$$G_n{=}\top \quad \vee \quad \Big(G_n{\neq}\top \ \wedge \ G_{n+1}{\neq}\top \ \wedge \ \forall \varphi \in G_{n+1}.\, \exists \varphi' \in G_n.\, \varphi \vdash \varphi'\Big).$$

This definition assumes that some G_n satisfies the above property. If such a G_n does not exist, the fix operator is not defined, so the abstract interpreter can diverge during the fixpoint computation. In Theorem 4.10, we will discharge this assumption and prove the termination of the linear rank abstraction.

Example 4.1. Consider the program C below:

$$\texttt{while}\ (x > 0 \wedge y > 0)\ (x{:=}x{-}1\ [\!]\ y{:=}y{-}1).$$

Given C, the abstract interpreter starts the fixpoint computation from the empty set $A_0 = \{\}$. The first iteration of the fixpoint computation is done in two steps. First, it applies the abstract transfer function of the loop body to $\{d_{\mathsf{id}}\} \cup A_0 = \{d_{\mathsf{id}}\}$:

$$\begin{aligned}
&\big([\![(x{:=}x{-}1\ [\!]\ y{:=}y{-}1)]\!]^\sharp \circ \mathsf{filter}^\dagger_{x>0\wedge y>0}\big)(\{d_{\mathsf{id}}\}) \\
&\quad = \quad [\![x{:=}x{-}1\ [\!]\ y{:=}y{-}1]\!]^\sharp\{d_{\mathsf{id}} \wedge x{>}0 \wedge y{>}0\} \\
&\quad = \quad [\![x{:=}x{-}1]\!]^\sharp\{d_{\mathsf{id}} \wedge x{>}0 \wedge y{>}0\} \cup [\![y{:=}y{-}1]\!]^\sharp\{d_{\mathsf{id}} \wedge x{>}0 \wedge y{>}0\} \\
&\quad = \quad [\![x{:=}x{-}1]\!]^\sharp\{\text{`}x{=}x \wedge \text{`}y{=}y \wedge x{>}0 \wedge y{>}0\} \cup [\![y{:=}y{-}1]\!]^\sharp\{\text{`}x{=}x \wedge \text{`}y{=}y \wedge x{>}0 \wedge y{>}0\} \\
&\quad = \quad \{\ \text{`}x{=}x' \wedge \text{`}y{=}y \wedge x'{>}0 \wedge y{>}0 \wedge x{=}x'{-}1,\quad \text{`}x{=}x \wedge \text{`}y{=}y' \wedge x{>}0 \wedge y'{>}0 \wedge y{=}y'{-}1\ \}.
\end{aligned}$$

Next, the abstract interpreter calls RANKFINDER twice with each of the two elements in the result set above. These function calls return x and y, from which the abstract interpreter constructs two ranking relations below:

$$T_x \stackrel{def}{=} (\text{`}x \geq 0 \wedge \text{`}x{-}\mathsf{dec} \geq x) \quad \text{and} \quad T_y \stackrel{def}{=} (\text{`}y \geq 0 \wedge \text{`}y{-}\mathsf{dec} \geq y).$$

The result A_1 of the first iteration is $\{T_x, T_y\}$.

The second fixpoint iteration computes:

$$A_1 \ \sqcup\ (\mathsf{RFS}^\dagger \circ [\![x{:=}x{-}1\ [\!]\ y{:=}y{-}1]\!]^\sharp \circ \mathsf{filter}^\dagger_{x>0\wedge y>0})A_1.$$

We show that the abstract element on the right hand side of the join, denoted A_2', is again A_1, so that the fixpoint computation converges here. To compute A_2', the analyzer first transforms A_1

according to the abstract meaning of the loop body. This results in a set with four elements:

$$\begin{aligned}\{\ & T_x[x'/x] \wedge x'{>}0 \wedge y{>}0 \wedge x{=}x'{-}1,\\ & T_x[y'/y] \wedge x{>}0 \wedge y'{>}0 \wedge y{=}y'{-}1,\\ & T_y[x'/x] \wedge x'{>}0 \wedge y{>}0 \wedge x{=}x'{-}1,\\ & T_y[y'/y] \wedge x{>}0 \wedge y'{>}0 \wedge y{=}y'{-}1\ \}.\end{aligned}$$

The first two elements come from transforming T_x according to the left and right branches of the loop body. The other two elements are obtained similarly from T_y. Next, the abstract interpreter calls RANKFINDER with all the four elements above. These four calls return x, x, y and y, which represent well-founded relations T_x, T_x, T_y, T_y. Thus, A'_2 is the same as T_x and T_y, and the fixpoint computation stops here.

After the fixpoint computation, the abstract interpreter composes the identity relation $\{d_{\mathsf{id}}\}$ with the result of the fixpoint computation:

$$\begin{aligned}\mathsf{comp}^\dagger(\{d_{\mathsf{id}}\},\{T_x,T_y\}) &= \{\ `x{=}x'_0 \wedge `y{=}y'_0 \wedge T_x[x'_0/`x],\quad `x{=}x'_0 \wedge `y{=}y'_0 \wedge T_y[y'_0/`y]\ \}\\ &= \{T_x,\ T_y\}.\end{aligned}$$

Finally, we apply $\mathsf{filter}^\dagger_{\neg(x>0\wedge y>0)}$ to the set above, which gives a set with four constraints:

$$\{\ T_x \wedge x \leq 0,\quad T_x \wedge y \leq 0,\quad T_y \wedge x \leq 0,\quad T_y \wedge y \leq 0\ \}.$$

Since the result is not $\top$, the abstract interpreter concludes that the given program c terminates.

In the example above, the fixpoint computation converges after two iterations. In the first iteration, which computes A_1, it finds ranking functions, and in the next iteration, it confirms that the ranking functions are preserved by the loop. In fact, we can prove that the fixpoint computation of the abstract interpreter always follows the same pattern, and finishes in two iterations. Suppose that RANKFINDER is well-behaved, such that

1. RFS always computes an optimal ranking function, in the sense that

$$(\mathsf{RFS}(\varphi) = \{T_E\} \wedge \gamma_r(\varphi) \subseteq \gamma_r(T_{E+b})) \implies b \geq 0,$$

2. RFS depends only on the (relational) meaning of its argument.

Lemma 4.9. *For all commands C, normalized expressions E and boolean expressions B, if there is a constraint $\varphi \in (\llbracket C \rrbracket^\sharp \circ \mathsf{filter}_B)(T_E)$ such that $\mathsf{RFS}(\varphi) = \{T_F\}$ and $\gamma_r(\varphi) \neq \emptyset$, then F is of the form $E - b$ for some nonnegative b.*

Proof. Before starting the proof, we note two facts. First, for all $\varphi \in (\llbracket C \rrbracket^\sharp \circ \mathsf{filter}_B)(T_E)$, there exists φ_0 such that

$$\gamma_r(\varphi) \;=\; \gamma_r(T_E); \gamma_r(\varphi_0)$$

where the semicolon means the composition of relations. One can prove this fact by the induction on the structure of C. Next, comp does not lose any information compared to relation composition (Lemma 4.8). Thus, for all φ_0, φ_1,

$$\gamma_r(\varphi_0); \gamma_r(\varphi_1) \;=\; \gamma_r(\mathsf{comp}(\varphi_0, \varphi_1)).$$

Suppose that a constraint φ in $(\llbracket C \rrbracket^\sharp \circ \mathsf{filter}_B)(T_E)$ satisfies

$$\mathsf{RFS}(\varphi) = \{T_F\} \quad \text{and} \quad \gamma_r(\varphi) \neq \emptyset.$$

Then, by the two facts mentioned above, there exists φ_0 such that

$$\gamma_r(\varphi) \;=\; \gamma_r(T_E); \gamma_r(\varphi_0) \;=\; \gamma_r(\mathsf{comp}(T_E, \varphi_0)).$$

This implies that $\mathsf{RFS}(\mathsf{comp}(T_E, \varphi_0)) = \{T_F\}$, because RFS only depends on the relational meaning of its argument and $\mathsf{RFS}(\varphi) = \{T_F\}$. From the soundness of RFS and what we have just shown so far, it follows that

$$(\gamma_r(T_E); \gamma_r(\varphi_0)) \subseteq \gamma_r(T_F) \quad \text{and} \quad (\gamma_r(T_E); \gamma_r(\varphi_0)) \neq \emptyset.$$

Using these two and $\mathsf{RFS}(\mathsf{comp}(T_E, \varphi_0)) = \{T_F\}$, we will derive the conclusion of this lemma. Since $\gamma_r(T_E); \gamma_r(\varphi_0)$ is not empty and it is a subset of $\gamma_r(T_F)$, there are states s, s', s'' such that

$$s[\gamma_r(T_E)]s', \quad s'[\gamma_r(\varphi_0)]s'' \quad \text{and} \quad s[\gamma_r(T_F)]s''.$$

Let n and m be, respectively, the value of E at s' and that of F at s''. Note that by the definition of T_E, if a state s_1 satisfies $E \geq \mathsf{max}(n{+}\mathsf{dec}, 0)$, we have that

$$s_1[\gamma_r(T_E); \gamma_r(\varphi_0)]s''.$$

Also, notice that by the definition of T_F, if $s_1[\gamma_r(T_F)]s''$, state s_1 should satisfy $F \geq \mathsf{max}(m{+}\mathsf{dec}, 0)$. Therefore, the condition $(\gamma_r(T_E); \gamma_r(\varphi_0)) \subseteq \gamma_r(T_F)$ implies that

$$E \geq \mathsf{max}(n{+}\mathsf{dec}, 0) \implies F \geq \mathsf{max}(m{+}\mathsf{dec}, 0).$$

holds in first-order logic. By Farkas's lemma [53] (which is defined the appendices A.1), this means that there exist a, b such that

$$a \geq 0 \quad \wedge \quad F = a \times E - b.$$

Number a is not 0, because $a = 0$ implies that F has to be a constant, which cannot be true since F is a ranking function. Furthermore, E and F both are normalized, so their first variables have 1 or -1 as their coefficients. Therefore, a has to be 1. It remains to show that b is nonnegative. For this, we will show that

$$\gamma_r(\mathsf{comp}(T_E, \varphi_0)) \subseteq \gamma_r(T_{F+b}).$$

This, together with the optimality of RFS, gives the required $b \geq 0$.

By the definition of T, we have that

$$\gamma_r(\mathsf{comp}(T_E, \varphi_0)) \;=\; (\gamma_r(T_E); \gamma_r(\varphi_0)) \;\subseteq\; \gamma_r(\text{`}E \geq 0) \;=\; \gamma_r(\text{`}F{+}b \geq 0).$$

Since $\gamma_r(T_E); \gamma_r(\varphi_0) \subseteq \gamma_r(T_F)$, we have that

$$\begin{aligned}\gamma_r(\mathsf{comp}(T_E, \varphi_0)) \;=\; (\gamma_r(T_E); \gamma_r(\varphi_0)) \;&\subseteq\; \gamma_r(\text{`}F{-}\mathsf{dec} \geq F) \\ &=\; \gamma_r(\text{`}F{+}b{-}\mathsf{dec} \geq F{+}b).\end{aligned}$$

Combining these two subset relationships gives $\gamma_r(\mathsf{comp}(T_E, \varphi_0)) \subseteq \gamma_r(T_{F+b})$, as desired. □

Theorem 4.10 (Fast Convergence)**.** *Suppose that the theorem prover* $\vdash$ *is complete. Then, for all commands* C*, the fixpoint iteration of*

$$G \;=\; \lambda A.\; (\mathsf{RFS}^{\dagger} \circ [\![C]\!]^{\sharp} \circ \mathsf{filter}^{\dagger}_{B})(\{d_{\mathsf{id}}\} \sqcup A)$$

terminates at most in two steps. Specifically, $G^2(\{\})$ *is* $\top$*, or the result of* $\mathsf{fix}\, G$ *is* $\{\}$ *or* $G(\{\})$*.*

Proof. Suppose that $G^2(\{\})$ is not $\top$. This implies that both $G(\{\})$ and $G^2(\{\})$ are finite sets of T_E's for normalized expressions E, because $G(= \mathsf{RFS}^{\dagger} \circ [\![C]\!]^{\sharp} \circ \mathsf{filter}^{\dagger}_{B})$ preserves $\top$. If $G(\{\})$ is empty, $\{\}$ is the fixpoint of G, thus becoming the result of $\mathsf{fix}\, G$, as claimed in the theorem. To prove the other nonempty case, suppose that $G(\{\})$ is a nonempty finite collection $A = \{T_{E_1}, \ldots, T_{E_n}\}$. We need to show that for each T_F in $G(A)$, there exists $T_{E_i} \in A$ such that $T_F \vdash T_{E_i}$, which is equivalent to $\gamma_r(T_F) \subseteq \gamma_r(T_{E_i})$ due to the completeness assumption about the prover. Pick T_F in $G(A)$. Since $G(= \mathsf{RFS}^{\dagger} \circ [\![C]\!]^{\sharp} \circ \mathsf{filter}^{\dagger}_{B})$ preserves the join operator, there exists T_{E_i} in A such that $T_F \in G(\{T_{E_i}\})$. This means that $\mathsf{RFS}(\varphi) = \{T_F\}$ for some constraint φ in $([\![C]\!]^{\sharp} \circ \mathsf{filter}_{B})(T_{E_i})$. Note that since RFS filters out all the provably inconsistent constraints and the prover is assumed complete, $\gamma_r(\varphi)$ is not empty. Thus, by Lemma 4.9, there is a nonnegative b such that $F = E - b$. This gives the required $\gamma_r(T_F) \subseteq \gamma_r(T_E)$. □

Note that the theorem suggests that we could have used a different fix operator that does not call the prover at all and just returns $G^2(\{\})$. We do not take this alternative in the chapter, since it is too specific for the RFS operator in this section; if RFS also keeps track of equality information, this two-step convergence result no longer holds.

4.4.1 Refinement with Simple Equalities

The linear rank abstraction cannot prove the termination of the program in Section 4.1. When the linear rank abstraction is run for the program, it finds the ranking functions x and y for the true and false branches of the program, but loses the information that the else branch does not change the value of x, which is crucial for the termination proof. As a result, the linear rank abstraction returns $\top$, and reports, incorrectly, the possibility of non-termination.

One way to solve this problem and improve the precision of the linear rank abstraction is to use a more precise RFS operator that additionally keeps simple forms of equalities. Concretely, this refinement keeps all the definitions of the linear rank abstraction, except that it replaces the rank

synthesizer RFS of the linear rank abstraction by RFS′ below:

$$\mathsf{RFS}'(\varphi) \stackrel{def}{=} \textbf{if } (\mathsf{RFS}(\varphi){=}\top) \textbf{ then } \top \textbf{ else } \left\{T_E \wedge (\wedge_{(\varphi \vdash \text{`}x{=}x)} \text{`}x{=}x) \mid T_E \in \mathsf{RFS}(\varphi)\right\}.$$

When this refined abstract interpreter is given the program in Section 4.1, it follows the informal description in that section and proves the termination of the program.

4.5 Experimental Evaluation

As in the previous chapter, we have implemented the abstract interpreter and compared it to other termination analysers. In particular we repeated the experiments from the previous chapter, comparing the new tool LINEARRANKTERM with analysers from the previous chapter as well as TERMINATOR and POLYRANK.

LR) LINEARRANKTERM is the new variance analysis that implements the linear rank abstraction with simple equalities in Section 4.4. This tool is implemented using CIL [45] allowing the analysis of programs written in C. However, no notion of shape is used in these implementations, restricting the input to only arithmetic programs. The tool uses RANKFINDER [48] as its linear rank synthesis engine and uses the Simplify prover [31] to filter out inconsistent states and check the implication between abstract states.

O) OCTATERM is the variance analysis [7] induced by the octagon analysis OCTANAL [44].

P) POLYTERM is the variance analysis [7] similarly induced from the polyhedra analysis POLY based on the New Polka Polyhedra library [39].

T) TERMINATOR [20].

These tools, except for TERMINATOR, were all run on a 2GHz AMD64 processor using Linux 2.6.16. TERMINATOR was executed on a 3GHz Pentium 4 using Windows XP SP2. Using different machines is unfortunate but somewhat unavoidable due to constraints on software library dependencies, etc. Note, however, that TERMINATOR running on the faster machine was still slower overall, so the qualitative results are meaningful. In any case, the running times are somewhat incomparable since on failed proofs TERMINATOR produces a counterexample path, but LINEARRANKTERM, OCTATERM and POLYTERM give a suspect pair of states

Figures 4.6 and 4.7 contain the results from the experiments performed with these analyses. For example, Figure 4.6(a) shows the outcome of the provers on example programs included in the OCTANAL distribution. Example 3 is an abstracted version of heapsort, and Example 4 of bubblesort.

	1		2		3		4		5		6	
LR	0.01	✓	0.01	✓	0.08	✓	0.09	✓	0.02	✓	0.06	✓
O	0.11	✓	0.08	✓	6.03	✓	1.02	✓	0.16	✓	0.76	✓
P	1.40	✓	1.30	✓	10.90	✓	2.12	✓	1.80	✓	1.89	✓
T	6.31	✓	4.93	✓	T/O	-	T/O	-	33.24	✓	3.98	✓

(a) Results from experiments with termination tools on arithmetic examples from the Octagon Library distribution.

	1		2		3		4		5		6	
LR	0.23	✓	0.20	⊘	0.00	⊘	0.04	✓	0.00	✓	0.03	✓
O	1.42	✓	1.67	⊘	0.47	⊘	0.18	✓	0.06	✓	0.53	✓
P	4.66	✓	6.35	⊘	1.48	⊘	1.10	✓	1.30	✓	1.60	✓
T	10.22	✓	31.51	⊘	20.65	⊘	4.05	✓	12.63	✓	67.11	✓

	7		8		9		10	
LR	0.07	✓	0.03	✓	0.01	⊘	0.03	✓
O	0.50	✓	0.32	✓	0.14	⊘	0.17	✓
P	2.65	✓	1.89	✓	2.42	⊘	1.27	✓
T	298.45	✓	444.78	✓	T/O	-	55.28	✓

(b) Results from experiments with termination tools on small arithmetic examples taken from Windows device drivers. Note that the examples are small as they must currently be hand-translated for the three tools.

LR is used to represent LINEARRANKTERM, **O** is used to represent OCTATERM, an Octagon-based variance analysis. **P** is POLYTERM, a Polyhedra-based variance analysis. The **T** represents TERMINATOR [20]. Times are measured in seconds. The timeout threshold was set to 500s. ✓="a proof was found". †="false counterexample returned". T/O = "timeout". ⊘="termination bug found". Note that pointers and aliasing from the device driver examples were removed by a careful hand translation when passed to the tools **O**, **P** and **LR**. Note that a time of 0.00 means that the analysis was too fast to be measured by the timing utilities used.

FIGURE 4.6: Experiments with 4 Termination Provers/Analyses (1/2)

Figure 4.6(b) contains the results of experiments on fragments of Windows device drivers. These examples are small because we currently must hand-translate them before applying all of the tools but TERMINATOR. LINEARRANKTERM is fastest on these examples. , but we had to insert some simple invariants by hand in order to prove termination for a few of the drivers. This is again a limitation of our implementation, but further work could solve this issue by involving a safety analyzer during the analysis. Figure 4.7(c) contains the results from experiments with the 4 tools on examples from the POLYRANK distribution.[5] The examples can be characterized as small but famously difficult (*e.g.* McCarthy's 91 function). Note that LINEARRANKTERM performs poorly on these examples because of the limitations of RANKFINDER. Many of these examples involve phase changes or tricky arithmetic in the algorithm.

[5] Note also that there is no benchmark number 5 in the original distribution. We have used the same numbering scheme as in the distribution so as to avoid confusion.

	1		2		3		4		6		7	
LR	0.19	✓	0.02	✓	0.01	†	0.02	†	0.02	†	0.01	†
O	0.30	†	0.05	†	0.11	†	0.50	†	0.10	†	0.17	†
P	1.42	✓	0.82	✓	1.06	†	2.29	†	2.61	†	1.28	†
T	435.23	✓	61.15	✓	T/O	-	T/O	-	75.33	✓	T/O	-

	8		9		10		11		12	
LR	0.04	†	0.01	†	0.03	†	0.02	†	0.01	†
O	0.16	†	0.12	†	0.35	†	0.86	†	0.12	†
P	0.24	†	1.36	✓	1.69	†	1.56	†	1.05	†
T	T/O	-	T/O	-	T/O	-	T/O	-	10.31	†

(c) Results from experiments with termination tools on arithmetic examples from the POLYRANK distribution.

FIGURE 4.7: Experiments with 4 Termination Provers/Analyses (2/2)

```
01        while (x>0 && y >0) {
02            x = x - y;
03            y = y*y;
04        }
```

FIGURE 4.8: A Program using Polynomial Expressions

From these experiments we can see that LINEARRANKTERM is fast and precise in comparison to previous approaches. The reason for the speed is that we have a fast convergence theorem 4.9 which states that the analysis terminates in two iterations at most. The prototype we have developed indicates that a termination analyzer using abstractions based on ranking functions shows a lot of promise.

4.6 Limitations

There are a number of limitations with the approach presented in this chapter.

Firstly the implementation we have produced can only find linear ranking functions, thus it is limited to programs using linear expressions only. We cannot analyse programs which use polynomial expressions such as the one in Fig.4.8. We can see that value of x is constantly decreasing, eventually breaking the condition in the loops guard, but using the RANKFINDER, we cannot find a linear ranking function to prove that this is the case. This limitation is due to rank function synthesis: there is no complete technique for finding polynomial ranking functions.

```
01        while (x>0 and z<0) {
02            x = x + y;
03            y = y + z;
04        }
```

FIGURE 4.9: A Program exhibiting Phase-Change

Another limitation is that we cannot handle programs which exhibit 'phase-changes': for some iterations the measure which would imply termination goes up before coming down and terminating the program. An example of such a program is: In this program the value of y may be positive initially, and so the value of x will go up, until the value of y becomes negative and x starts to decrease until it breaks out of the loop. The technique we have in this chapter cannot infer such phase changes and so cannot prove termination of this example.

Another limitation is that in order to find the right ranking functions we need to find some invariants for the program. In the examples we have tried, taking invariants from the Octagon domain helped in some cases. This is a known limitation for proving liveness properties: often we need to know some extra safety information in order to find proof of a liveness property.

Finally, one major limitation is that the soundness technique in this chapter can only handle programs with tail recursion/iteration. For more general recursive constructs we need to use an alternative approach as shown later in this thesis.

4.7 Conclusion

In this chapter we have defined an abstract interpreter targeted to proving program termination properties. The abstract interpreter is constructed from ranking functions that overapproximate the program being analysed. The prototype tool we have produced shows that the technique is fast and precise.

However, the analysers developed in this chapter have can only analyse programs with iteration. This limitation is due to the fact that we have greatest fixpoints in the concrete semantics and least fixpoints in the abstract interpreter. In this chapter we showed that the two are linked when we have iteration, but the same result does not hold for recursion. I n the next chapter we will address this issue using metric spaces.

Chapter 5

Metric Semantics and Termination Analysis

In the previous chapter we defined a generic abstract interpreter to prove termination properties for a programming language with iteration. In this chapter we will consider a language with recursion over the unit type, and develop a termination analysis for the language. We will focus on building a theoretical framework that answers when an abstract interpreter is sound for proving liveness properties such as termination. The soundness for safety properties is well understood using the standard framework for abstract interpretation. Our aim is to provide a similar understanding for liveness properties.

The framework in this chapter will use the theory of metric spaces[1] to define a concrete semantics of programs, and connect this semantics with the usual order-theoretic semantics of abstract interpretation. As noted in the previous chapter, in order to prove the soundness of a termination analysis in the standard theory of abstract interpretation, we need to use greatest fixpoints in the concrete trace semantics, and relate it to pre-fixpoints computed by the termination analysis. Relating these two kinds of fixpoints is non-trivial. In the previous chapter, we were able to do so, because the programming language included only iterations whose infinite behaviour can be captured by the repetition of least fixpoints. Had the language included non-tail recursive control flow, the soundness proof would not have been sound.

In this chapter, we will instead use a different type of concrete semantics based on metric spaces, where recursions are interpreted using Banach's unique fixpoint theorem. We will show a relationship between unique fixpoints in the metric semantics and pre-fixpoints computed by abstract interpreters. Based on this relationship, we will provide a set of conditions for deciding when an abstract interpreter is sound for liveness/termination properties. Our conditions are

[1] For a brief overview of metric spaces, see Appendix A.

$$
\begin{array}{lcl}
E & ::= & x \mid r \mid E + E \mid r \times E \\
B & ::= & E = E \mid E \neq E \mid E \leq E \mid E < E \mid B \wedge B \mid B \vee B \mid \neg B \\
a & ::= & x := E \mid x := * \\
C & ::= & a \mid C; C \mid C \,[]\, C \mid \texttt{if}\, B\, C\, C \mid f() \mid \texttt{fix}\, f.\, C
\end{array}
$$

FIGURE 5.1: Programming Language with General Recursions

presented in a general framework, so that they can be re-used when analysis designers develop new techniques for proving liveness properties of programs.

This chapter starts by defining a programming language with recursion over the unit type. Then, it moves on to our framework for defining sound abstract interpreters for liveness properties. The framework consists of a concrete semantics and an abstract semantics[2], and it specifies the conditions needed by both semantics. Finally, we will present an instance of the framework, and prove that the instance satisfies the conditions from the framework.[3]

5.1 Programming Language

Let PNames be the set of procedures names, ranged over by f, g, h, and let AtomicComm be the set of atomic commands, such as the assignment $x := E$, ranged over by symbol a. We consider an imperative language with parameter-less procedures, whose grammar is given in Figure 5.1. In the figure, r denotes a constant rational number.

The language is an imperative language with rational variables and parameterless recursive procedures, which are not necessarily tail recursive. Most of the commands are standard. The only unusual case is the definition of recursive procedure $\texttt{fix}\, f.\, C$. This command defines a recursive procedure f whose body is C, and then it immediately calls the defined procedure. Note that the programming language here is a superset of the language in the previous chapter, because while loops can be encoded using recursions.

We will write $\Gamma \vdash C$ for a finite subset Γ of PNames, where Γ includes all the free function names in C. This notation makes it explicit which functions can be called inside a command.

[2] We do not consider an intermediate semantics that is used to factor the soundness argument in Chapter 4.

[3] The abstract interpreter induced by this instance is not implemented yet. This is left as future work.

5.2 Framework

The central idea of this chapter is the use of metric-space semantics as a concrete semantics of programs and to relate it with the abstract semantics used for an abstract interpreter for termination properties. The metric-space semantics models infinite computations of a program accurately: the meaning of a non-terminating program includes infinite traces, whereas the meaning of a terminating program contains only finite traces. Thus, the semantics can serve as a reference point for proving the soundness of an abstract interpreter for liveness properties. More importantly, the metric space alleviates the difficulty of the main challenge in proving the soundness of a termination analysis, which is to relate the semantics of recursions in the concrete semantics with that in an abstract interpreter. This is because the metric-space semantics uses the unique fixpoints of contractive functions and under reasonable conditions (which will be identified by our framework and justified by our instance), the unique fixpoints can be overapproximated by pre-fixpoints computed by abstract interpreters.

The background materials on metric spaces can be found in Appendix A.

5.2.1 Concrete Semantics

Our framework consists of two parts: a concrete metric-space semantics and an abstract order-theoretic semantics. In this section we will define the concrete semantics.

Let ω be the set of positive integers. To define the concrete semantics we require the following:

- A pre-ordered complete metric space $(\mathcal{D}, d, \sqsubseteq, \top)$ with a top element $\top$. We require that for all Cauchy sequences $\{x_n\}_{n\in\omega}$ in $\mathcal{D}$ and all elements $x \in \mathcal{D}$, if x_∞ is the limit of $\{x_n\}$ then
$$(\forall n \in \omega.\, x_n \sqsubseteq x) \implies (x_\infty \sqsubseteq x). \tag{5.1}$$
Elements of $\mathcal{D}$ can be understood as semantic counterparts of syntactic commands C. Our concrete semantics will interpret C as an element in $\mathcal{D}$.

- A monotone non-expansive join operator $\sqcup$:
$$\sqcup : \mathcal{D} \times \mathcal{D} \to \mathcal{D}.$$
This operator is used to model a non-deterministic choice in our programming language.

- Monotone non-expansive functions seq and if_B for all boolean conditions B:
$$\mathsf{seq} : \mathcal{D} \times \mathcal{D} \to \mathcal{D}, \qquad \mathsf{if}_\mathsf{B} : \mathcal{D} \times \mathcal{D} \to \mathcal{D}.$$

They decide the meanings of sequencing and if-then-else commands in our language.

- A family of functions $\mathsf{procrun}_f$ indexed by PNames ($f \in \mathsf{PNames}$) for modelling the execution of procedures:

$$\mathsf{procrun}_f : \mathcal{D} \to \mathcal{D}.$$

  We require that $\mathsf{procrun}_f(-)$ be a monotone $\frac{1}{2}$-contractive function for all $f \in \mathsf{PNames}$. This function models the computation steps performed immediately before and after running the body of the procedure f.

- An interpretation of atomic commands, which is a family of sets ,trans_a indexed by $a \in \mathsf{AtomicComm}$, in $\mathcal{D}$:

$$\mathsf{trans}_a : \mathcal{D}.$$

  Here $\mathsf{AtomicComm}$ is $\{x := E, x := * \mid x \text{ is a variable and } E \text{ is an expression}\}$.

- A subset LIVPROPERTY of $\mathcal{D}$ that is downward closed with respect to $\sqsubseteq$:

$$x \sqsubseteq y \ \wedge \ y \in \textsc{LivProperty} \quad \Longrightarrow \quad x \in \textsc{LivProperty}.$$

  Intuitively, this subset consists of elements in $\mathcal{D}$ (which are semantic counterparts of commands) satisfying a desired liveness property, such as termination.

The requirement (5.1) on $\sqsubseteq$ and Cauchy sequences ensures a close relationship between the metric structure and the pre-order structure of $\mathcal{D}$, which is formalized in the lemma below.

Lemma 5.1. *For all $\frac{1}{2}$-contractive monotone functions $F : \mathcal{D} \to \mathcal{D}$, if x is a pre-fixpoint of F (i.e., $F(x) \sqsubseteq x$), we have that*

$$\mathsf{ufix}\, F \ \sqsubseteq \ x.$$

where $\mathsf{ufix}\, F$ *of F is the unique fixpoint of F.*

Proof. Let x be a pre-fixpoint of F. By the Banach fixpoint theorem, we know that the unique fixpoint $\mathsf{ufix}\, F$ of F exists and is also the limit of the following Cauchy sequence:

$$x,\ F(x),\ F^2(x),\ F^3(x),\ \ldots$$

Since x is a pre-fixpoint of F (i.e., $F(x) \sqsubseteq x$) and F is monotone, we also know that

$$x \ \sqsupseteq \ F(x) \ \sqsupseteq \ F^2(x) \ \sqsupseteq \ F^3(x) \ \sqsupseteq \ F^4(x) \ \ldots$$

$$
\begin{array}{rcl}
[\![\Gamma \vdash C]\!] & : & [\![\Gamma]\!] \to \mathcal{D} \\
\\
[\![\Gamma \vdash a]\!]\eta & = & \mathsf{trans}_a \\
[\![\Gamma \vdash C_1; C_2]\!]\eta & = & \mathsf{seq}([\![\Gamma \vdash C_1]\!]\eta, [\![\Gamma \vdash C_2]\!]\eta) \\
[\![\Gamma \vdash C_1 [] C_2]\!]\eta & = & [\![\Gamma \vdash C_1]\!]\eta \sqcup [\![\Gamma \vdash C_2]\!]\eta \\
[\![\Gamma \vdash \mathtt{if}\ B\ C_1\ C_2]\!]\eta & = & \mathsf{if}_{\mathsf{B}}([\![\Gamma \vdash C_1]\!]\eta, [\![\Gamma \vdash C_2]\!]\eta) \\
[\![\Gamma \vdash f()]\!]\eta & = & \eta(f) \\
[\![\Gamma \vdash \mathtt{fix}\ f.C]\!]\eta & = & \mathsf{ufix}\left(\lambda k.\ \mathsf{procrun}_f([\![\Gamma, f \vdash C]\!]\eta[f \mapsto k])\right).
\end{array}
$$

FIGURE 5.2: Concrete Semantics defined by the Framework

That is, $F^n(x) \sqsubseteq x$ for all n. Thus, the limit $\mathsf{ufix}\, F$ of $\{F^n(x)\}_{n\in\omega}$ also satisfies

$$\mathsf{ufix}\ F \ \sqsubseteq\ x$$

by the requirement (5.1) of our framework. We have just proved the lemma. □

The conditions above give rise to a metric-space semantics of programs. Let $[\![\Gamma]\!]$ be the domain for procedure environments (i.e., $\Pi_{f\in\Gamma}\mathcal{D}$), pre-ordered pointwise and given the product metric. The semantics interprets $\Gamma \vdash C$ as a non-expansive map from $[\![\Gamma]\!]$ to $\mathcal{D}$, and it appears in Figure 5.2.

We will now prove that the semantics is well-defined:

Lemma 5.2. *The semantics is well-defined.*

Proof. We need to prove that for all commands $\Gamma \vdash C$, $[\![\Gamma \vdash C]\!]$ is a well-defined non-expansive function from $[\![\Gamma]\!]$ to $\mathcal{D}$. We do this using induction on the structure of C. The cases of function call and atomic command follow from the fact that both projection functions and constant functions are well-defined and non-expansive. The induction goes through for the cases of the sequential composition, the non-deterministic choice and the if statement, because of the induction hypothesis and the non-expansiveness requirements on seq, $\sqcup$ and if_{B}.

The remaining case is the recursion: $\Gamma \vdash \mathtt{fix}\, f.C$. Pick environments η, η' and let F, G be functions defined by

$$F(k) \ =\ \mathsf{procrun}_f([\![\Gamma, f \vdash C]\!]\eta[f \mapsto k]), \quad G(k) \ =\ \mathsf{procrun}_f([\![\Gamma, f \vdash C]\!]\eta'[f \mapsto k]).$$

Firstly, we prove the well-definedness. For this, it is sufficient to prove that F is $\frac{1}{2}$-contractive, so that we can apply the Banach fixpoint theorem, which implies that the unique fixpoint of F

exists. We can use this unique fixpoint to interpret recursion. To prove the contractiveness of F, consider k, k'. By induction hypothesis, $[\![\Gamma, f \vdash C]\!]$ is a well-defined non-expansive map. Thus:

$$d(k, k') \quad \geq \quad d\big([\![\Gamma, f \vdash C]\!]\eta[f \mapsto k],\ [\![\Gamma, f \vdash C]\!]\eta[f \mapsto k']\big). \tag{5.2}$$

In our framework we have required that $\mathsf{procrun}_f(-)$ be $\frac{1}{2}$-contractive. Hence:

$$\begin{aligned} & d([\![\Gamma, f \vdash C]\!]\eta[f \mapsto k],\ [\![\Gamma, f \vdash C]\!]\eta[f \mapsto k']) \\ & \geq\ \tfrac{1}{2} \times d(\mathsf{procrun}_f([\![\Gamma, f \vdash C]\!]\eta[f \mapsto k]),\ \mathsf{procrun}_f([\![\Gamma, f \vdash C]\!]\eta[f \mapsto k'])) \\ & \geq\ \tfrac{1}{2} \times d(F(k), F(k')). \end{aligned} \tag{5.3}$$

Putting the conclusions of (5.2) and (5.3), we get that $d(k, k') \geq (\frac{1}{2} \times d(F(k), F(k')))$, the $\frac{1}{2}$-contractiveness of F.

Secondly, we prove that $[\![\Gamma \vdash \texttt{fix}\, f.C]\!]$ defines a non-expansive function. Since η, η' are chosen arbitrarily, it is sufficient to show that

$$d(\eta,\, \eta') \ \geq\ d(\mathsf{ufix}\ F,\ \mathsf{ufix}\ G).$$

Pick k from $\mathcal{D}$. By the Banach fixpoint theorem, both $\{F^n(k)\}_{n\in\omega}$ and $\{G^n(k)\}_{n\in\omega}$ are Cauchy sequences with limits $\mathsf{ufix}\ F$ and $\mathsf{ufix}\ G$, respectively.

We claim that the n-th elements of these two sequences are close to each other:

$$\forall n \in \omega.\quad d(F^n(k), G^n(k)) \ \leq\ d(\eta,\, \eta'). \tag{5.4}$$

This claim can be proved by induction on n. When $n = 0$, the LHS of the inequality is zero, so the claim holds. Suppose that $n > 0$ and also that the claim holds for all $m < n$. By the induction hypothesis,

$$d(F^{n-1}(k), G^{n-1}(k)) \ \leq\ d(\eta,\, \eta').$$

Then,

$$d\big(\ \eta[f \mapsto F^{n-1}(k)],\ \ \eta'[f \mapsto G^{n-1}(k)]\ \big) \ \leq\ d(\eta,\, \eta').$$

Now, the non-expansiveness of $[\![\Gamma, f \vdash C]\!]$ and $\mathsf{procrun}_f$ implies that

$$d\big(\ F(F^{n-1}(k)),\ G(G^{n-1}(k))\ \big) \ \leq\ d\big(\ \eta[f \mapsto F^{n-1}(k)],\ \ \eta'[f \mapsto G^{n-1}(k)]\ \big).$$

Combining the two inequalities above gives the claim (5.4) for n.

Using (5.4), we can complete the proof of non-expansiveness. Pick $\epsilon > 0$. Then, there exists N such that for all $n > N$,

$$d(\mathsf{ufix}\ F,\ F^n(k)) \leq \epsilon/2 \quad \text{and} \quad d(G^n(k),\ \mathsf{ufix}\ G) \leq \epsilon/2.$$

By the triangular inequality, we have that

$$\begin{aligned} d(\mathsf{ufix}\ F,\ \mathsf{ufix}\ G) &\leq d(\mathsf{ufix}\ F,\ F^n(k)) + d(F^n(k), G^n(k)) + d(G^n(k),\ \mathsf{ufix}\ G) \\ &\leq \epsilon/2 + d(\eta,\ \eta') + \epsilon/2 \\ &= d(\eta,\ \eta') + \epsilon. \end{aligned}$$

Thus, $d(\mathsf{ufix}\ F,\ \mathsf{ufix}\ G) \leq d(\eta,\ \eta') + \epsilon$. Since this holds for all $\epsilon > 0$, we have the required

$$d(\mathsf{ufix}\ F,\ \mathsf{ufix}\ G) \leq d(\eta,\ \eta').$$

□

Lemma 5.3. *For all commands $\Gamma \vdash C$, their meanings $[\![\Gamma \vdash C]\!]$ are monotone functions.*

Proof. The proof is by induction on the structure of C. The monotonicity is immediate in the cases of function calls and atomic commands. For the cases of the sequential composition, the non-deterministic choice and the if statement, it follows from the induction hypothesis and the monotonicity of seq, $\sqcup$ and if_{B}. Now, it remains to show the monotonicity for the recursion case:

$$\Gamma \vdash \mathtt{fix}\ f.C$$

Consider η, η' such that $\eta \sqsubseteq \eta'$. Let F, G be functions on $\mathcal{D}$ given by

$$F(k) = \mathsf{procrun}_f([\![\Gamma, f \vdash C]\!]\eta[f \mapsto k]), \qquad G(k) = \mathsf{procrun}_f([\![\Gamma, f \vdash C]\!]\eta'[f \mapsto k]).$$

Define x and y to be the unique fixpoints of F and G respectively. By the induction hypothesis and the monotonicity of $\mathsf{procrun}_f$, we have that

$$k \sqsubseteq k' \implies \eta[f \mapsto k] \sqsubseteq \eta'[f \mapsto k'] \implies F(k) \sqsubseteq G(k'). \tag{5.5}$$

We need to prove that $x \sqsubseteq y$. By the Banach fixpoint theorem, x is the limit of the below Cauchy sequence:

$$y,\ F(y),\ F^2(y),\ F^3(y),\ \ldots.$$

By the requirement (5.1) of our framework, it suffices to show that

$$F^k(y) \ \sqsubseteq\ y.$$

We do this by induction on k. When $k = 0$, $F^k(y) = y$ so the inequality above holds. Suppose that $k > 0$. By the induction hypothesis on k, we have that $F^{k-1}(y) \sqsubseteq y$. Thus, by (5.5), this implies the required inequality:

$$F^k(y) \ = \ F(F^{k-1}(y)) \ \sqsubseteq\ G(y) \ = \ y$$

where the last equality uses the fact that y is the fixpoint of G. □

5.2.2 Abstract Semantics

The second part of our framework is the abstract semantics. For a function $f : X^n \to X$ and a subset X_0 of X, we say that f can be restricted to X_0 if for all $\vec{x} \in X_0^n$, we have that $f(\vec{x}) \in X_0$. Our framework requires the following data for the abstract semantics:

- A set $(\mathcal{A}, \bot, \top)$ with two distinguished different elements $\bot$ and $\top$.
- Subset $\mathcal{A}_t$ of $\mathcal{A}$ such that $\top \in \mathcal{A}_t$ but $\bot \notin \mathcal{A}_t$. We write $\mathcal{A}_p$ for $\mathcal{A} - \mathcal{A}_t$, and call elements in $\mathcal{A}_t$ *total*.
- An algorithm $\mathsf{checktot}$ that answers the membership to $\mathcal{A}_t$ soundly but not necessarily in a complete way. That is, $\mathsf{checktot}(A) = \mathsf{true}$ means that $A \in \mathcal{A}_t$, but $\mathsf{checktot}(A) \neq \mathsf{true}$ does not mean that $A \notin \mathcal{A}_t$.
- Concretization function $\gamma : \mathcal{A}_t \to \mathcal{D}$, such that $\gamma(\top) = \top$. Note that the domain $\mathcal{A}_t$ of γ does not include $\bot$ and any abstract elements in $\mathcal{A}_p$.
- Functions $\mathsf{seq}^\sharp$, $\sqcup^\sharp$ and $\mathsf{if}_\mathsf{B}^\sharp$ for all boolean conditions B:

  $$\mathsf{seq}^\sharp : \mathcal{A} \times \mathcal{A} \to \mathcal{A}, \qquad \sqcup^\sharp : \mathcal{A} \times \mathcal{A} \to \mathcal{A}, \qquad \mathsf{if}_\mathsf{B}^\sharp : \mathcal{A} \times \mathcal{A} \to \mathcal{A}.$$

  They give the abstract meanings of the sequential composition, the non-deterministic choice and the conditional statement in our language. We require that these functions

satisfy two conditions. Firstly, they can be restricted to $\mathcal{A}_t$. Secondly, the functions overapproximate their concrete counterparts:

$$\begin{aligned}\forall A_0, A_1 \in \mathcal{A}_t. \quad & \mathsf{seq}(\gamma(A_0), \gamma(A_1)) \sqsubseteq \gamma(\mathsf{seq}^\sharp(A_0, A_1)) \\ & \text{and} \quad \gamma(A_0) \sqcup \gamma(A_1) \sqsubseteq \gamma(A_0 \sqcup^\sharp A_1) \\ & \text{and} \quad \mathsf{if}_\mathsf{B}(\gamma(A_0), \gamma(A_1)) \sqsubseteq \gamma(\mathsf{if}^\sharp_\mathsf{B}(A_0, A_1)).\end{aligned}$$

Note that this soundness condition considers only total elements.

- Function $\mathsf{procrun}^\sharp$ indexed by PNames, for modelling the execution of procedures:

$$\mathsf{procrun}^\sharp_\mathsf{f} : \mathcal{A} \to \mathcal{A}.$$

For all $f \in \mathsf{PNames}$, we require that $\mathsf{procrun}^\sharp_\mathsf{f}$ can be restricted to $\mathcal{A}_t$, and that it overapproximate $\mathsf{procrun}_f$:

$$\forall f \in \mathsf{PNames}. \quad \forall A \in \mathcal{A}_t. \quad \mathsf{procrun}_f(\gamma(A)) \sqsubseteq \gamma(\mathsf{procrun}^\sharp_\mathsf{f}(A)).$$

- Interpretation of atomic commands, $\mathsf{trans}^\sharp{}_a$ in $\mathcal{A}_t$,

$$\mathsf{trans}^\sharp{}_a : \mathcal{A}_t.$$

The interpretation is required to overapproximate trans:

$$\mathsf{trans}_a \sqsubseteq \gamma(\mathsf{trans}^\sharp{}_a).$$

- Binary operator $\triangledown : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$, usually called widening. This operator needs to satisfy three conditions. Firstly, it can be restricted to a map from $\mathcal{A}_t$. Secondly, it overapproximates an upper bound of its arguments:

$$\forall A_1, A_2 \in \mathcal{A}_t. \quad \gamma(A_1) \sqsubseteq \gamma(A_1 \triangledown A_2) \quad \text{and} \quad \gamma(A_2) \sqsubseteq \gamma(A_1 \triangledown A_2).$$

Finally, it turns any sequences in $\mathcal{A}$ into one with a stable element. That is, for all $\{A_n\}_{n\in\omega}$ in $\mathcal{A}$, the below widened sequence

$$A'_1 = A_1, \qquad A'_{n+1} = A_n \triangledown A_{n+1}$$

contains an index m with $A'_m = A'_{m+1}$.

$$
\begin{array}{rcl}
[\![\Gamma \vdash C]\!]^\sharp & : & [\![\Gamma]\!]^\sharp \to \mathcal{A} \\
[\![\Gamma \vdash f()]\!]^\sharp\eta^\sharp & = & \eta^\sharp(f) \\
[\![\Gamma \vdash a]\!]^\sharp\eta^\sharp & = & \mathsf{trans}^\sharp_a \\
[\![\Gamma \vdash C_1; C_2]\!]^\sharp\eta^\sharp & = & \mathsf{seq}^\sharp([\![\Gamma \vdash C_1]\!]^\sharp\eta^\sharp,\ [\![\Gamma \vdash C_2]\!]^\sharp\eta^\sharp) \\
[\![\Gamma \vdash C_1\ [\,]\ C_2]\!]^\sharp\eta^\sharp & = & [\![\Gamma \vdash C_1]\!]^\sharp\eta^\sharp \sqcup^\sharp [\![\Gamma \vdash C_2]\!]^\sharp\eta^\sharp \\
[\![\Gamma \vdash \mathtt{if}\ B\ C_1\ C_2]\!]^\sharp\eta^\sharp & = & \mathsf{if}^\sharp_{\mathsf{B}}([\![\Gamma \vdash C_2]\!]^\sharp\eta^\sharp,\ [\![\Gamma \vdash C_2]\!]^\sharp\eta^\sharp) \\
[\![\Gamma \vdash \mathtt{fix}\ f.C]\!]^\sharp\eta^\sharp & = & \lceil \mathsf{widenfix}\ F \rceil \\
 & & (\text{where } F(A) = \mathsf{procrun}^\sharp_{\mathsf{f}}([\![\Gamma, f \vdash C]\!]^\sharp\eta^\sharp[f \mapsto A])
\end{array}
$$

FIGURE 5.3: Abstract Semantics defined by the Framework

- A predicate SATISFYLIV$^\sharp$ on $\mathcal{A}_t$ such that

$$\forall A \in \mathcal{A}_t.\quad \textsc{satisfyLiv}^\sharp(A) = \mathsf{true} \implies \gamma(A) \in \textsc{LivProperty}.$$

  Intuitively, SATISFYLIV$^\sharp$ identifies abstract elements denoting commands with a desired liveness property.

The data above are enough to give an abstract semantics of programs, but to do so, we need to define two operators using the data. The first operator is the ceiling $\lceil - \rceil$, which replaces non-total elements in $\mathcal{A}_p$ by $\top$:

$$\lceil A \rceil = \begin{cases} A & \text{if } \mathsf{checktot}(A) = \mathsf{true} \\ \top & \text{otherwise.} \end{cases}$$

The second is the widened fixpoint operator widenfix. For every function $F : \mathcal{A} \to \mathcal{A}$, the operator constructs the sequence

$$A_1 = \bot, \qquad A_{n+1} = A_n \triangledown F(A_n)$$

and returns the first w_m with $A_m = A_{m+1}$. The condition on $\triangledown$ ensures that such A_m exists.

Let $[\![\Gamma]\!]^\sharp$ be the abstract domain for procedure environments (i.e., $[\![\Gamma]\!]^\sharp = \Pi_{f \in \Gamma}\mathcal{A}$). The abstract semantics interprets programs $\Gamma \vdash C$ as (not necessarily monotone nor continuous) functions from $[\![\Gamma]\!]^\sharp$ to $\mathcal{A}$. The defining clauses in the semantics are given in Figure 5.3.

One important property of the abstract semantics is that the meaning function maps environments with total components to total elements in $\mathcal{A}$.

Lemma 5.4. *For all $\Gamma \vdash C$ and $\eta^\sharp \in [\![\Gamma]\!]^\sharp$,*

$$(\forall f \in \Gamma.\ \eta^\sharp(f) \in \mathcal{A}_t) \implies [\![\Gamma \vdash C]\!]^\sharp\eta^\sharp \in \mathcal{A}_t.$$

Proof. We prove the lemma by induction on the structure of C. Suppose we have a procedure environment $\eta^\sharp$ that map procedure names to elements in $\mathcal{A}_t$. We will consider each case of C separately and prove that $[\![\Gamma \vdash C]\!]^\sharp\eta^\sharp \in \mathcal{A}_t$.

- Case $C \equiv f()$. By assumption, $\eta^\sharp(f) \in \mathcal{A}_t$. Thus, $[\![C]\!]^\sharp\eta^\sharp \in \mathcal{A}_t$.
- Case $C \equiv a$. Our framework requires that $\mathsf{trans}^\sharp{}_a$ be in $\mathcal{A}_t$. The lemma follows from this requirement.
- Case $C \equiv C_1; C_2$. By the induction hypothesis, both $[\![C_1]\!]^\sharp\eta^\sharp$ and $[\![C_2]\!]^\sharp\eta^\sharp$ is in $\mathcal{A}_t$. Furthermore, our framework requires that $\mathsf{seq}^\sharp$ map pairs of total elements to total elements. Hence,
$$[\![C]\!]^\sharp\eta^\sharp \;=\; \mathsf{seq}^\sharp([\![C_1]\!]^\sharp\eta^\sharp,\ [\![C_2]\!]^\sharp\eta^\sharp) \;\in\; \mathcal{A}_t.$$
- Cases $C \equiv C_1 [] C_2$ and $C \equiv \texttt{if}\ B\ C_1\ C_2$. These cases are similar to the previous one. The desired conclusion follows from the induction hypothesis and the requirements on $\sqcup^\sharp$ and $\mathsf{if}^\sharp_{\mathsf{B}}$ with respect to total elements.
- Case $C \equiv \texttt{fix}\ f.C_1$. In this case, $[\![C]\!]^\sharp\eta^\sharp$ is always total, even when some component of $\eta^\sharp$ is not total. This is because of the $\lceil - \rceil$ operator in the semantics of $[\![\texttt{fix}\ f.C_1]\!]^\sharp$, whose range contains only total elements.

□

Now we state the soundness of the abstract semantics:

$$\forall \eta^\sharp \in [\![\Gamma]\!]^\sharp.\ (\forall f \in \Gamma.\ \eta^\sharp(f) \in \mathcal{A}_t) \implies [\![\Gamma \vdash C]\!]\gamma(\eta^\sharp) \sqsubseteq \gamma([\![\Gamma \vdash C]\!]^\sharp\eta^\sharp). \tag{5.6}$$

In $\gamma(\eta^\sharp)$, we use the componentwise extension of γ to procedure environments. Note that although γ is not defined on non-total elements, the soundness claim above is well-formed, because Lemma 5.4 ensures that $[\![\Gamma \vdash C]\!]^\sharp\eta^\sharp$ is total. We prove the soundness in the next theorem:

Theorem 5.5. *The abstract semantics is sound. That is,* (5.6) *holds for all commands* $\Gamma \vdash C$.

Proof. Our proof is by induction on the structure of C.

- Case $C \equiv f()$. $[\![f()]\!]\gamma(\eta^\sharp) \;=\; \gamma(\eta^\sharp)(f) \;=\; \gamma(\eta^\sharp(f)) \;=\; \gamma([\![f()]\!]^\sharp\eta^\sharp)$.

- Case $C \equiv a$. This case follows from the requirement of our framework that $\mathsf{trans}^\sharp_a$ should overapproximate trans_a.

- Case $C \equiv C_1; C_2$. This case follows from three ingredients – the induction hypothesis, the monotonicity of seq and the requirement that $\mathsf{seq}^\sharp$ should overapproximate seq. The below derivation shows how these ingredients give the desired conclusion.

$$\begin{array}{rcll}
[\![C_1; C_2]\!]\gamma(\eta^\sharp) & = & \mathsf{seq}([\![C_1]\!]\gamma(\eta^\sharp),\ [\![C_2]\!]\gamma(\eta^\sharp)) & \\
& \sqsubseteq & \mathsf{seq}(\gamma([\![C_1]\!]^\sharp\eta^\sharp),\ \gamma([\![C_2]\!]^\sharp\eta^\sharp)) & \text{(by ind. hypo and mono. of } \mathsf{seq}) \\
& \sqsubseteq & \gamma(\mathsf{seq}^\sharp([\![C_1]\!]^\sharp\eta^\sharp,\ [\![C_2]\!]^\sharp\eta^\sharp)) & \text{(since } \mathsf{seq}^\sharp \text{ overapproximates } \mathsf{seq}) \\
& = & \gamma([\![C_1; C_2]\!]^\sharp\eta^\sharp). &
\end{array}$$

- Cases $C \equiv C_1 [] C_2$ and $C \equiv \mathtt{if}\ B\ C_1\ C_2$. These cases are very similar to the above. It follows from the induction hypothesis, the monotonicity of $\sqcup$ and if_B, and the overapproximation properties of $\sqcup^\sharp$ and $\mathsf{if}^\sharp_\mathsf{B}$, as shown below.

$$\begin{array}{rcll}
[\![C_1\ [] \ C_2]\!]\gamma(\eta^\sharp) & = & [\![C_1]\!]\gamma(\eta^\sharp) \sqcup [\![C_2]\!]\gamma(\eta^\sharp) & \\
& \sqsubseteq & \gamma([\![C_1]\!]^\sharp\eta^\sharp) \sqcup \gamma([\![C_2]\!]^\sharp\eta^\sharp) & \text{(by ind. hypo and mono. of } \sqcup) \\
& \sqsubseteq & \gamma([\![C_1]\!]^\sharp\eta^\sharp \sqcup^\sharp [\![C_2]\!]^\sharp\eta^\sharp) & \text{(since } \sqcup^\sharp \text{ overapproximates } \sqcup) \\
& = & \gamma([\![C_1\ [] \ C_2]\!]^\sharp\eta^\sharp). &
\end{array}$$

$$\begin{array}{rcll}
[\![\mathtt{if}\ B\ C_1\ C_2]\!]\gamma(\eta^\sharp) & = & \mathsf{if}_\mathsf{B}([\![C_1]\!]\gamma(\eta^\sharp),\ [\![C_2]\!]\gamma(\eta^\sharp)) & \\
& \sqsubseteq & \mathsf{if}_\mathsf{B}(\gamma([\![C_1]\!]^\sharp\eta^\sharp),\ \gamma([\![C_2]\!]^\sharp\eta^\sharp)) & \text{(by ind. hypo and mono. of } \mathsf{if}_\mathsf{B}) \\
& \sqsubseteq & \gamma(\mathsf{if}^\sharp_\mathsf{B}([\![C_1]\!]^\sharp\eta^\sharp,\ [\![C_2]\!]^\sharp\eta^\sharp)) & \text{(since } \mathsf{if}^\sharp_\mathsf{B} \text{ overapproximates } \mathsf{if}_\mathsf{B}) \\
& = & \gamma([\![\mathtt{if}\ B\ C_1\ C_2]\!]^\sharp\eta^\sharp). &
\end{array}$$

- Case $C \equiv \mathtt{fix}\ f.C_1$. Let

$$\begin{array}{rcl}
F(x) & = & \mathsf{procrun}_f([\![\Gamma, f \vdash C_1]\!]\gamma(\eta^\sharp)[f \mapsto x]), \\
G(A) & = & \mathsf{procrun}^\sharp_\mathsf{f}([\![\Gamma, f \vdash C_1]\!]^\sharp\eta^\sharp[f \mapsto A]).
\end{array}$$

We need to prove that

$$(\mathsf{ufix}\ F) \ \sqsubseteq\ \gamma(\lceil \mathsf{widenfix}\ G \rceil). \tag{5.7}$$

If $\mathsf{checktot}(\mathsf{widenfix}\ G) \neq \mathsf{true}$, then $\gamma(\lceil \mathsf{widenfix}\ G \rceil) = \gamma(\top) = \top$. Thus, (5.7) holds. Suppose that $\mathsf{checktot}(\mathsf{widenfix}\ G) = \mathsf{true}$, which implies that $\mathsf{widenfix}\ G \in \mathcal{A}_t$. In this

case, it is sufficient to prove that $\gamma(\mathsf{widenfix}\ G)$ is a pre-fixpoint of F. Because then, the inequality (5.7) follows from Lemma 5.1. By the definition of widenfix,

$$(\mathsf{widenfix}\ G) \quad = \quad (\mathsf{widenfix}\ G) \triangledown G(\mathsf{widenfix}\ G).$$

Because of the condition on $\triangledown$, this implies that

$$\gamma(G(\mathsf{widenfix}\ G)) \ \sqsubseteq\ \gamma(\mathsf{widenfix}\ G).$$

The LHS of this inequality is greater than or equal to $F(\gamma(\mathsf{widenfix}\ G))$ as shown below:

$$\begin{aligned}
\gamma(G(\mathsf{widenfix}\ G)) &= \gamma\big(\mathsf{procrun}^\sharp_{\mathsf{f}}\ [\![\Gamma, f \vdash C_1]\!]^\sharp \eta^\sharp [f \mapsto (\mathsf{widenfix}\ G)]\big)\big) \\
&\sqsupseteq \mathsf{procrun}_f\big(\gamma\big([\![\Gamma, f \vdash C_1]\!]^\sharp \eta^\sharp [f \mapsto (\mathsf{widenfix}\ G)]\big)\big) \\
&\sqsupseteq \mathsf{procrun}_f\big([\![\Gamma, f \vdash C_1]\!] \gamma(\eta^\sharp) [f \mapsto \gamma(\mathsf{widenfix}\ G)]\big) \\
&= F(\gamma(\mathsf{widenfix}\ G)).
\end{aligned}$$

The first inequality holds because $\mathsf{procrun}^\sharp$ overapproximates $\mathsf{procrun}$. The second inequality follows from the induction hypothesis and the monotonicity of $\mathsf{procrun}_f$. This shows $F(\gamma(\mathsf{widenfix}\ G)) \sqsubseteq \gamma(\mathsf{widenfix}\ G)$, as desired.

□

5.2.3 Generic Analysis

Let $\eta^\sharp_*$ be the unique element in the abstract semantics of the empty environment type $\Gamma = \emptyset$. Our generic analysis takes a command C with no free procedures, and computes the function:

$$\textsc{LivAnalysis}(C) \quad = \quad \textsc{SatisfyLiv}^\sharp([\![C]\!]^\sharp \eta^\sharp_*).$$

The result is a boolean value, indicating whether C satisfies a liveness property specified by LivProperty.

Theorem 5.6. *Let η_* be the unique element in the concrete semantics of the empty environment $\Gamma = \emptyset$. Then, for all commands C with no free procedures, we have that*

$$\textsc{LivAnalysis}(C) = \mathsf{true} \quad \Longrightarrow \quad [\![C]\!]\eta_* \in \textsc{LivProperty}.$$

Proof. Consider a command C that do not contain free procedure names. Suppose that

$$\textsc{LivAnalysis}(C) = \mathsf{true}.$$

Then, by Theorem 5.5,

$$[\![C]\!]\eta_* \;=\; [\![C]\!]\gamma(\eta^\sharp_*) \;\sqsubseteq\; \gamma([\![C]\!]^\sharp\eta^\sharp_*). \tag{5.8}$$

In the first equality, we use the fact that $\gamma(\eta^\sharp_*) = \eta_*$. Furthermore, since $\textsc{SatisfyLiv}^\sharp$ is a sound checker for the membership of $\textsc{LivProperty}$ and $\textsc{LivAnalysis}(C) = \mathsf{true}$, we also have that

$$\gamma([\![C]\!]^\sharp\eta^\sharp_*) \;\in\; \textsc{LivProperty}. \tag{5.9}$$

From (5.8), (5.9) and the downward closure of $\textsc{LivProperty}$, it follows that $[\![C]\!]\eta_*$ is in $\textsc{LivProperty}$, as desired. □

5.2.4 Discussion on using a Metric Space in the Framework

Using a metric space in our framework entails that a user of the framework needs to discharge new proof obligations when defining a concrete semantics. The user has to prove that the semantic domain $\mathcal{D}$ for the meaning of commands in the concrete semantics is a complete metric space, in addition to having a standard order structure. Also, the user should show that all the semantic operators are non-expansive operators.

These new proof obligations often make it impossible to re-use a existing concrete semantics. For instance, the trace semantics in the previous chapter uses the powerset of traces as a semantic universe for commands, but this powerset cannot be used in our framework, because it does not form a complete metric space, when it is given a natural notion of distance measure. In order to use the framework in this chapter, one has to modify the powerset of traces, such that it has a good metric-theoretic structure, as will be done in the next section of the chapter.

The need for changing an existing concrete semantics also has a direct implication on the design of an abstract semantics. For instance, if we consider only certain "good subsets" of traces in the (concrete) semantic domain but we want to re-use ideas from existing abstract domains, we usually end up with an abstract domain with ill-behaving elements, in the sense that the concretizations of those elements are not "good subsets" of traces. To address the presence of these ill-behaving elements, our framework classifies elements in the abstract domain $\mathcal{A}$ to total elements in $\mathcal{A}_t$ and non-total ones in $\mathcal{A}_p$. Then, it allows the concretization γ to ignore non-total ones in $\mathcal{A}_p$, by asking γ to be a function from $\mathcal{A}_t$, not $\mathcal{A}$.

5.3 Instance of the Framework

5.3.1 Concrete Semantics

In this section, we give an instance of concrete semantics of our framework, where commands are interpreted as trace sets satisfying certain healthiness conditions. We will first introduce the notion of *tagged states*, define *well-formed traces* as certain sequences of tagged states, and describe a metric on those traces. As we wish to interpret commands as sets of well-formed traces, we will then lift this metric to sets of traces. Interestingly, this lifted metric do not necessarily satisfy the required axioms for being a metric space, if we consider trace sets without any conditions. Hence, we will restrict our attention to those trace sets satisfying two conditions—closedness and fullness, both of which will be explained later in the section. Finally, we provide the meaning of semantic operators required by our framework, such as seq and $\mathsf{if_B}$.

5.3.1.1 Tagged States, and Well-formed Pre-traces and Traces

Let Vars be a finite set of program variables and $\mathsf{Rationals}$ the set of rationals, which will be stored in those variables. *States* are mappings from program variables to rational numbers and *tagged states* are pairs of states and tags:

$$\begin{aligned} \mathsf{Tags} &= \{none\} \cup (\mathsf{PNames} \times \{call, ret\}), \\ \mathsf{States} &= \mathsf{Vars} \to \mathsf{Rationals}, \\ \mathsf{tagStates} &= \mathsf{States} \times \mathsf{Tags}. \end{aligned}$$

The tag of a tagged state indicates whether the state is the initial or the final state of a procedure call, or just a normal one not related to a call. The $(f, call)$ and (f, ret) tags mean that the state is, respectively, the initial and the final state of the call $f()$, and the $none$ tag indicates that the state is a normal state, i.e., it is neither the initial nor the final state of a procedure call. We use symbol σ (with subscripts or superscripts) to denote elements in $\mathsf{tagStates}$, and use s to denote elements in States.

In this chapter, we will often omit the adjective "tagged" in "tagged state" and say simply "states". If we intend to talk about elements in States, we will always use the phrase "untagged states".

A *well-formed pre-trace* τ is defined to be a nonempty finite or infinite sequence of tagged states, such that τ starts with a $none$-tagged state and if it is finite, it ends with a $none$-tagged state.

We write preTraces for the set of all well-formed pre-traces, i.e.,

$$\begin{aligned} \mathsf{nStates} &= \mathsf{States} \times \{none\}, \\ \mathsf{preTraces} &= \mathsf{nStates}(\mathsf{tagStates}^*)\mathsf{nStates} \cup \mathsf{nStates}(\mathsf{tagStates}^\infty). \end{aligned}$$

For $\tau \in \mathsf{preTraces}$ and $n \in \omega \cup \{\infty\}$,[4] the projection $\tau[n]$ is the n-prefix of τ; in case that $|\tau| < n$, $\tau[n] = \tau$. Note that $\tau[n]$ does not necessarily belong to preTraces, but this will not cause problems for our results. Using this projection, we define the distance function on well-formed pre-traces as follows:

$$d_B(\tau, \tau') \;=\; 2^{-\max\{n \,|\, \tau[n]=\tau'[n]\}}$$

where we regard $2^{-\infty} = 0$. Note that this is the same as the Baire metric defined in A.7.

A *well-formed trace* τ is a well-formed pre-trace that satisfies two additional conditions. To define these conditions, we consider the sets $\mathcal{W}, \mathcal{O}$ of sequences of tagged states that are the least fixpoints of the below equations:

$$\begin{aligned} \mathcal{W} &= \mathsf{nStates}^* \cup \mathcal{W}\mathcal{W} \cup \big(\bigcup_{f \in \mathsf{PNames}, s, s_1 \in \mathsf{States}} \{(s, (f, call))\}\, \mathcal{W}\, \{(s_1, (f, ret))\}\big), \\ \mathcal{O} &= \mathcal{W} \cup \mathcal{O}\mathcal{O} \cup \big(\bigcup_{f \in \mathsf{PNames}, s \in \mathsf{States}} \{(s, (f, call))\}\mathcal{O}\big). \end{aligned}$$

Intuitively, $\mathcal{W}$ describes sequences where every procedure call has a matching return and calls and returns satisfy the well-bracketedness condition. The other set $\mathcal{O}$ defines a bigger set; in each trace in $\mathcal{O}$, some procedure calls might not have matching returns, but calls and returns should satisfy the well-bracketedness condition.

Definition 5.7 (Well-formed Trace)**.** A well-formed pre-trace τ is a *well-formed trace* if and only if τ is finite and belongs to $\mathcal{W}$, or τ is infinite and all of its finite prefixes are in $\mathcal{O}$. We write Traces for the set of well-formed traces.

Note that in our definition of well-formed traces we require that they be well-formed pre-traces, in addition to being a member of $\mathcal{W}$ or $\mathcal{O}$. Hence, when a well-formed trace is finite, it should start and end with $none$-tagged states (which is a condition for being a well-formed pre-trace). This property of well-formed traces will enable us to avoid any housekeeping with tags in our semantics, especially when we define the meaning of the sequencing operator.

In the rest of this chapter, we will omit "well-formed" in "well-formed pre-traces" and "well-formed traces", and simply call them "pre-traces" and "traces". Our use of the word "traces" in this chapter should not be confused with the notion of traces from the previous chapter, where

[4] In this thesis, ω means the set of positive integers.

traces meant nonempty sequences of untagged states. In this chapter, traces will always mean well-formed traces of tagged states.

We will now show that the sets preTraces and Traces are well-defined metric spaces.

Lemma 5.8. $(\mathsf{preTraces}, d_B)$ *is a metric space.*

Proof. The symmetry of d_B is immediate from the definition. Also,

$$d_B(\tau,\tau') = 0 \iff (\max\{n \mid \tau[n] = \tau'[n]\}) = \infty \iff \tau = \tau'.$$

Thus, to show that d_B is a metric on preTraces, it remains to prove the triangular inequality. Consider τ, τ', τ'' in preTraces. We will prove a stronger-than-required property that

$$d_B(\tau,\tau') \;\leq\; \max\big(d(\tau,\tau''), d_B(\tau'',\tau')\big).$$

Equivalently,

$$\max\{n \mid \tau[n] = \tau'[n]\} \;\geq\; \min\big(\max\{n \mid \tau[n] = \tau''[n]\},\; \max\{n \mid \tau''[n] = \tau'[n]\}\big).$$

Let m be the minimum on the RHS of the above inequality. Then, $\tau[m] = \tau''[m]$ and $\tau''[m] = \tau'[m]$. Thus, $\tau[m] = \tau'[m]$. This means that the LHS of the above inequality should be greater than or equal to m. □

Corollary 5.9. (Traces, d_B) *is a metric space.*

Proof. The corollary holds, because Traces is a subset of preTraces and it inherits the metric from preTraces. □

Next, we will prove that metric spaces $(\mathsf{preTraces}, d_B)$ and (Traces, d_B) are complete. As a preparation for this proof, we notice that our definition of the distance d_B allows simpler characterization of Cauchy sequence. Recall that ω is the set of positive integers.

Lemma 5.10. *A sequence* $\{\tau_i\}_{i\in\omega}$ *in* preTraces *is Cauchy if and only if*

$$\forall m \in \omega.\;\; \exists n \in \omega.\;\; \forall n' \geq n.\;\; \tau_{n'}[m] = \tau_n[m].$$

Proof. Let R^+ be positive real numbers. Recall that by the definition of Cauchy sequence a sequence $\{\tau_i\}_{i\in\omega}$ in preTraces is Cauchy if and only if

$$\forall \epsilon \in R^+.\ \ \exists n \in \omega.\ \ \forall n' \geq n.\ \ d(\tau_{n'}, \tau_n) \ \leq\ \epsilon. \tag{5.10}$$

Restricting ϵ in (5.10) to those of the form 2^{-m} for some $m \in \omega$ preserves the meaning. Thus, (5.10) is equivalent to

$$\forall m \in \omega.\ \ \exists n \in \omega.\ \ \forall n' \geq n.\ \ d_B(\tau_{n'}, \tau_n) \ \leq\ 2^{-m}. \tag{5.11}$$

But by the definition of the distance d, we have that

$$\begin{aligned} d_B(\tau_{n'}, \tau_n) \ \leq\ 2^{-m} \quad &\iff \quad \max(\{m' \mid \tau_{n'}[m'] = \tau_n[m']\}) \ \geq\ m \\ &\iff \quad \tau_{n'}[m] = \tau_n[m]. \end{aligned}$$

Thus, (5.11) is equivalent to

$$\forall m \in \omega.\ \ \exists n \in \omega.\ \ \forall n' \geq n.\ \ \tau_{n'}[m] = \tau_n[m].$$

This gives the claimed equivalence in this lemma. □

Lemma 5.11. $(\mathsf{preTraces}, d_B)$ *is complete.*

Proof. Consider a Cauchy sequence $\{\tau_n\}_{n\in\omega}$. By the definition of Cauchy sequence, we have that

$$\forall m \in \omega.\ \ \exists n_m \in \omega.\ \ \forall n \geq n_m.\ \ \tau_{n_m}[m] = \tau_n[m]. \tag{5.12}$$

For each m, define

$$m^* = \max(\{n_{m'} \mid m' \leq m\} \cup \{m\})$$

where $n_{m'}$ is the index in (5.12). Using this notation, we define a sequence τ_∞ as follows:

$$\mathsf{proj}(\tau_\infty, m) \ = \ \begin{cases} \mathsf{proj}(\tau_{m^*}, m) & \text{if } \mathsf{proj}(\tau_{m^*}, m) \text{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

where $\mathsf{proj}(\tau_\infty, m)$ means the m-th element of τ_∞. Note that since every τ_n has length at least 1 and it starts with a state in nStates, $\mathsf{proj}(\tau_\infty, 1)$ is defined and it is a state in nStates. To show

that τ_∞ is the limit, it is sufficient to prove that

$$\forall m \in \omega.\ \exists n_m \in \omega.\ \forall n \geq n_m.\ \tau_n[m] = \tau_\infty[m]. \tag{5.13}$$

Before proving this, we note that it implies that τ_∞ is a pre-trace, i.e., $\tau_\infty \in \mathsf{preTraces}$. If τ_∞ is infinite, τ_∞ belongs to $\mathsf{nStates}(\mathsf{tagStates}^\infty) \subseteq \mathsf{preTraces}$, because τ_∞ starts with a state in $\mathsf{nStates}$. If τ_∞ is finite, (5.13) implies that $\tau_\infty = \tau_n$ for some $n \in \omega$, so $\tau_\infty \in \mathsf{preTraces}$.

Now, let's go back to our task of proving (5.13). Pick m. We claim that m^* is the witness n_m of the existential quantification in (5.13). To prove our claim, consider $n \geq m^*$. We need to show that

$$\forall k \in \omega.\ 1 \leq k \leq m \implies \mathsf{proj}(\tau_n, k) = \mathsf{proj}(\tau_\infty, k) \tag{5.14}$$

where the equality should be interpreted as both undefined or both defined and equal. (In the rest of the proof, we use the same interpretation of equality.) By the definition of m^*, if $1 \leq k \leq m$, then $k^* \leq m^*$, so $k^* \leq n$. This implies that

$$\mathsf{proj}(\tau_{k^*}, k) \ = \ \mathsf{proj}(\tau_n, k).$$

But by definition, $\mathsf{proj}(\tau_\infty, k) = \mathsf{proj}(\tau_{k^*}, k)$. From this, the desired (5.14) follows. □

Lemma 5.12. (Traces, d_B) *is complete.*

Proof. Consider a Cauchy sequence $\{\tau_n\}_{n\in\omega}$ in Traces. Let τ_∞ be the limit of this sequence in $\mathsf{preTraces}$, which exists because of Lemma 5.11. It remains to prove that τ_∞ belongs to Traces. By the definition of metric d and Lemma 5.10, we have that

$$\forall m \in \omega.\ \exists n_m \in \omega.\ \forall n \geq n_m.\ \tau_\infty[m] = \tau_n[m]. \tag{5.15}$$

Thus, if τ_∞ is finite, it has to be the same as some τ_n. So, it has to be in Traces, as desired. Otherwise, τ_∞ is infinite. In this case, (5.15) implies that all prefixes of τ_∞ are also prefixes of some traces. But, prefixes of traces always belong to $\mathcal{O}$, by the definition of traces. Thus, all prefixes of τ_∞ are in $\mathcal{O}$. This implies that τ_∞ is a trace. □

We finish this section with a remark on unusual features of the metric space (Traces, d_B). Firstly, Traces does not include the empty sequence. Technically, this is because the empty sequence makes it difficult to define a non-expansive sequencing operator. Secondly, all traces start with

$none$-tagged states in $\mathsf{nStates}$. This captures that we use states with call or return tags only in the intermediate steps of computation, in order to mark call and return points of procedure invocation. Finally, calls and returns in traces are well-bracketed, so we can talk about a pair of matching call and return in a single trace. Furthermore, all calls in a finite trace have matching returns in the trace. Our abstract semantics later will exploit these properties of traces to improve the precision.

5.3.1.2 Full Closed Sets of Well-formed Traces

The concrete semantics in this chapter will use a restricted powerset of traces, whose elements satisfy two conditions, called closedness and fullness. In this section, we will explain this restricted powerset.

Before moving onto the explanation, we discuss the motivation behind the restriction. Recall that in the previous chapter, we used the entire powerset of nonempty finite or infinite sequences of $none$-tagged states to interpret commands in the concrete semantics. Unfortunately, we cannot use the entire powerset of traces in this chapter, because the semantics here uses the metric-space structure but the powerset of such traces do not form a complete metric space, when a standard lifting of d_A in (Traces, d) is used as a distance for trace sets.[5] Technically, it is to get a good mathematical structure, namely a complete metric space, that we consider only restricted sets of traces.

A subset $T_0 \subseteq \mathsf{Traces}$ of traces is *closed* if for all Cauchy sequences of traces in T_0, their limits belong to T_0 as well. A trace set $T_0 \subseteq \mathsf{Traces}$ is *full* if for every $none$-tagged state $\sigma \in \mathsf{nStates}$, there is a trace $\tau \in T_0$ starting with σ. Note that since $\mathsf{nStates}$ is not empty, a full trace set T_0 is also nonempty.

The semantic domain $\mathcal{D}$ for interpreting commands in our concrete semantics is the set of full closed sets of traces.

$$\mathcal{D} = \mathcal{P}_{fcl}(\mathsf{Traces})$$

This domain has the lifted Baire Metric A.13:

$$d_B^+(T, T') \;=\; \begin{cases} 0 & \text{if } T = T' \\ 2^{-\max\{n \,|\, T[n]=T'[n]\}} & \text{otherwise} \end{cases}$$

where $T[n]$ is the result of taking the prefix of every trace in T (i.e., $T = \{\tau[n] \mid \tau \in T\}$). The closedness ensures that the d just defined satisfies the axioms for being a metric. Also, the

[5] Concretely, the axiom

$$d_B^+(T_0, T_1) = 0 \iff T_0 = T_1$$

breaks for some trace sets T_0 and T_1 and the lifted metric d.

condition about being full allows us to meet the contractiveness requirement for procrun in our framework, as will be shown later.

Our domain $\mathcal{D}$ is ordered by the subset relation $\subseteq$. It has the top element with respect to this $\subseteq$ order, which is the set Traces of all traces. It also has the join operator given by the set union.

We will now prove that

$$(\mathcal{D}, d_B^+, \subseteq, \mathsf{Traces}) \quad \text{and} \quad \cup$$

are instantiations of the first and second components of our framework. We first define a slightly simpler characterization of Cauchy sequences in $(\mathcal{D}, d)$, which we will use in the proofs of our results.

Lemma 5.13. *A sequence $\{T_n\}_{n\in\omega}$ in $\mathcal{D}$ is Cauchy if and only if*

$$\forall m \in \omega.\ \ \exists n \in \omega.\ \ \forall n' \geq n.\ \ T_{n'}[m] = T_n[m].$$

Proof. The proof is almost identical to that of Lemma 5.10, except that we replace τ_n, $\tau_{n'}$ and their m prefix projections by T_n, $T_{n'}$ and the m prefix projections of T_n and $T_{n'}$. □

Next, we prove that $(\mathcal{D}, d_B^+)$ is a complete metric space. In order to make the thesis self-contained, we will reply only on elementary definitions of metric spaces in the proof. However, we point out that the proof can be simplified, if one uses existing nontrivial results on metric spaces (in particular Hahn's theorem A.12 in Appendix A).

Lemma 5.14. *$(\mathcal{D}, d_B^+)$ is a metric space.*

Proof. The symmetry of d_B^+ is immediate from the definition. Next, we show that

$$d_B^+(T, T') = 0 \iff T = T'.$$

By the definition of d_B^+, $d_B^+(T, T) = 0$ for all $T \in \mathcal{D}$. The right-to-left direction of the equivalence follows from this. For the other direction, suppose that $d_B^+(T, T') = 0$. Pick $\tau \in T$. Then, for all $n \in \omega$,

$$\tau[n] \ \in \ T[n] \ = \ T'[n].$$

This implies that

$$\forall n \in \omega.\ \ \exists \tau'_n \in T'.\ \ (\tau'_n)[n] = \tau[n].$$

Thus, $\{\tau'_n\}_{n\in\omega}$ is a Cauchy sequence with τ as its limit. Since T' is closed and the sequence $\{\tau'_n\}_{n\in\omega}$ is in T', the limit τ should be in T' as well. We have just shown that $T \subseteq T'$. The other inclusion can be proved similarly.

Finally, we prove that d satisfies the triangular inequality. In fact, we prove a stronger property that for all $T, T', T'' \in \mathcal{D}$,

$$d_B^+(T,T') \;\leq\; \max(d_B^+(T,T''),\, d_B^+(T'',T')),$$

which is equivalent to

$$\max\{n \mid T[n]=T'[n]\} \;\geq\; \min\big(\max\{n \mid T[n]=T''[n]\},\; \max\{n \mid T''[n]=T'[n]\}\big).$$

Let m be the value of the RHS of the above inequality. Then, $T[m] = T''[m]$ and $T''[m] = T'[m]$. Thus, $T[m] = T'[m]$. This means that the LHS of the above inequality should be at least m. □

Lemma 5.15. *For all Cauchy sequences $\{T_n\}_{n\in\omega}$ in $\mathcal{D}$ and indices $m, k \in \omega$, if*

$$\forall k' \geq k.\;\; T_k[m] = T_{k'}[m],$$

then for all $\tau \in T_k$, there exists a Cauchy sequence $\{\tau_i\}_{i\in\omega}$ in Traces *such that*

1. *$\tau[m] = \tau_i[m]$ for all $i \in \omega$, and*
2. *the sequence is taken from an infinite subsequence of $\{T_n\}_{n\in\omega}$, i.e.,*

$$\exists\{k_i\}_{i\in\omega}.\;\; (\forall i \in \omega.\;\; \tau_i \in T_{k_i}) \;\;\wedge\;\; (\forall i,j \in \omega.\;\; i<j \implies k_i < k_j).$$

Proof. Let $\{T_n\}_{n\in\omega}$ and m, k be the ones satisfying the conditions of the lemma. Pick τ from T_k. Using these data, we will construct two desired sequences—the Cauchy sequence $\{\tau_i\}_{i\in\omega}$ and the sequence $\{k_i\}_{i\in\omega}$ of indices. Note that since $\{T_n\}_{n\in\omega}$ is Cauchy,

$$\forall o \in \omega.\;\; \exists n_o \in \omega.\;\; \forall n \geq n_o.\;\; T_{n_o}[o] = T_n[o].$$

Using n_o's, we define the desired sequence of indices by

$$k_1 \;=\; n_m \quad \text{and} \quad k_{i+1} \;=\; \max(n_{(m+i)},\; k_i + 1\,).$$

Note that this sequence is strictly increasing. It remains to construct the other sequence $\{\tau_i\}_{i\in\omega}$ of traces. We do this inductively. By the assumption on T_k and the choice of n_m, we must have that

$$T_k[m] \;=\; T_{\max(k,n_m)}[m] \;=\; T_{(n_m)}[m] \;=\; T_{(k_1)}[m].$$

Hence,

$$\tau[m] \;=\; \tau'[m] \quad \text{for some } \tau' \in T_{(k_1)}.$$

We define the first element of the sequence by

$$\tau_1 \;=\; \tau'.$$

For the rest, we assume that τ_i is chosen from $T_{(k_i)}$, and we inductively pick trace τ_{i+1} from $T_{(k_{i+1})}$ as follows. Since $k_{i+1} > k_i \geq n_{(m+i-1)}$, we have that

$$T_{(k_{i+1})}[m+i-1] \;=\; T_{n_{(m+i-1)}}[m+i-1] \;=\; T_{(k_i)}[m+i-1].$$

Furthermore, τ_i is in $T_{(k_i)}$. Thus, there must exist $\tau'' \in T_{(k_{i+1})}$ such that

$$\tau_i[m+i-1] \;=\; \tau''[m+i-1].$$

We define τ_{i+1} to be this τ''.

By construction, it is immediate that $\{\tau_i\}_{i\in\omega}$ is from the infinite subsequence $\{T_{(k_i)}\}_{i\in\omega}$ of $\{T_n\}_{n\in\omega}$. Furthermore, $\{\tau_i\}_{i\in\omega}$ is Cauchy, because

$$\forall n \in \omega.\;\; \forall i \geq n.\;\; \tau_i[m+i-1] = \tau_{i+1}[m+i-1],$$

and so,

$$\forall n \in \omega.\;\; \forall i \geq n.\;\; \tau_n[n] = \tau_i[n].$$

(Remember here that $m \in \omega$ and so $m \geq 1$.) Finally, by construction, $\tau_1[m] = \tau_i[m]$ for all $i \in \omega$. But $\tau_1[m] = \tau'[m] = \tau[m]$. Thus, $\tau_i[m] = \tau[m]$ for all $i \in \omega$, as desired. □

Proposition 5.16. $(\mathcal{D}, d_B^+)$ *is complete.*

Proof. Consider a Cauchy sequence $\{T_n\}_{n\in\omega}$ in $\mathcal{D}$. Define T_∞ as follows:

$$T_\infty \;=\; \{\, \lim_{i\to\infty} \tau_i \;\mid\; \{\tau_i\}_{i\in\omega} \text{ is Cauchy } \wedge \\ \exists\{k_i\}_{i\in\omega}.\; (\forall i\in\omega.\; \tau_i \in T_{k_i}) \;\wedge\; (\forall i,j\in\omega.\; i<j \implies k_i<k_j)\,\}.$$

Firstly, we show that T_∞ is closed. Consider a Cauchy sequence $\{\alpha_n\}_{n\in\omega}$ in T_∞. Let α_∞ be the limit of this sequence. To prove the closedness, we need to show that α_∞ belongs to T_∞. Equivalently, we need to find a Cauchy sequence $\{\tau_i\}_{i\in\omega}$ such that

1. the limit of the sequence is α_∞, and
2. the sequence is taken from an infinite subsequence of $\{T_n\}_{n\in\omega}$, i.e., it satisfies that

$$\exists\{k_i\in\omega\}_{i\in\omega}.\;\; (\forall i\in\omega.\; \tau_i\in T_{k_i}) \;\wedge\; (\forall i,j\in\omega.\; i<j \implies k_i<k_j).$$

Since $\{\alpha_n\}_{n\in\omega}$ converges to α_∞, we have that

$$\forall m\in\omega.\;\; \exists n_m\in\omega.\;\; \forall n'\geq n_m.\;\; \alpha_{n'}[m]=\alpha_\infty[m].$$

For each $m\in\omega$, we let

$$m^* \;=\; \max(\{n_{m'} \mid m'\leq m\}\cup\{m\}).$$

The maximum is used to ensure that $-^*$ is monotone with respect to $\leq$. Since α_n is in T_∞, the definition of T_∞ implies the existence of a Cauchy sequence $\{\tau_i^n\}_{i\in\omega}$ such that the limit of the sequence is α_n and the sequence satisfies that

$$\exists\{k_i^n\in\omega\}_{i\in\omega}.\;\; (\forall i\in\omega.\; \tau_i^n\in T_{(k_i^n)}) \;\wedge\; (\forall i,j\in\omega.\; i<j \implies k_i^n<k_j^n).$$

Thus,

$$\forall n\in\omega.\;\; \forall m\in\omega.\;\; \exists i_{n,m}\in\omega.\;\; \forall i'\geq i_{n,m}.\;\; \tau_{i'}^n[m]=\alpha_n[m].$$

For each $m\in\omega$, define

$$m^\dagger \;=\; i_{m^*,m}$$

Also, construct an increasing sequence $\{j_i\}_{i\in\omega}$ of natural numbers by

$$j_1 \;=\; 1^\dagger \quad\text{and}\quad j_{i+1} \;=\; \min\{\, j' \mid k_{(j')}^{(i+1)^*} > k_{(j_i)}^{(i^*)} \;\wedge\; j'\geq (i+1)^\dagger\,\}.$$

Using these $-^*$ and $\{j_i\}_{i\in\omega}$, we construct the required $\{\tau_i\}_{i\in\omega}$ as follows:

$$\tau_i \;=\; \tau_{(j_i)}^{(i^*)}$$

Then, for all $m \in \omega$ and all $i \geq m$,

$$\tau_i[i] \;=\; \tau_{(j_i)}^{(i^*)}[i] \;=\; \alpha_{(i^*)}[i] \;=\; \alpha_\infty[i].$$

The first equality is just the unrolling of the definition of τ_i. The second equality holds, because $j_i \geq i^\dagger$ and so $\tau_{(j_i)}^{(i^*)}[i] = \alpha_{(i^*)}[i]$. The third equality follows from the definition of i^*. We have just shown that $\tau_i[i] = \alpha_\infty[i]$, and since $i \geq m$, this implies

$$\tau_i[m] = \alpha_\infty[m].$$

Thus, $\{\tau_i\}_{i\in\omega}$ is a Cauchy sequence that converges to α_∞. Furthermore, this sequence is taken from an infinite subsequence of $\{T_n\}_{n\in\omega}$. Concretely, the indices of this infinite subsequence are

$$k_{(j_i)}^{(i^*)} \qquad \text{for all } i \in \omega.$$

By the choice of j_i, the index sequence is strictly increasing: if $i < l$, then $k_{(j_i)}^{(i^*)} < k_{(j_l)}^{(l^*)}$.

Secondly, we prove that T_∞ is full. Note that this implies that $T_\infty \in \mathcal{D}$. Choose a $none$-tagged state $\sigma \in \mathsf{nStates}$. It is sufficient to construct a Cauchy sequence $\{\tau_i\}_{i\in\omega}$ such that

1. for all $i \in \omega$, $\tau_i[1]$ is the singleton trace σ, and
2. there exists $\{k_i\}_{i\in\omega}$ satisfying that

$$\forall i, j \in \omega.\quad \tau_i \in T_{(k_i)} \;\wedge\; (i < j \implies k_i < k_j).$$

We will construct the desired sequence $\{\tau_i\}_{i\in\omega}$ using Lemma 5.15. Note that since $\{T_n\}_{n\in\omega}$ is Cauchy,

$$\exists n_1 \in \omega.\quad \forall n \geq n_1.\quad T_n[1] = T_{(n_1)}[1].$$

Since all T_n's are full, there must be a trace τ in $T_{(n_1)}$ whose starting state is σ. Now, Lemma 5.15 implies the existence of a Cauchy sequence $\{\tau_i\}_{i\in\omega}$ such that

1. $\tau_i[1] = \tau[1]$ for all $i \in \omega$, and

2. there exists $\{k_i\}_{i\in\omega}$ satisfying

$$\forall i, j \in \omega.\ \ \tau_i \in T_{(k_i)} \ \wedge\ (i < j \implies k_i < k_j).$$

But, $\tau_i[1] = \tau[1]$ means that $\tau_i[1]$ is the singleton trace σ. Thus, $\{\tau_i\}_{i\in\omega}$ is the sequence that we are looking for.

Finally, we prove that T is the limit of $\{T_n\}_{n\in\omega}$. Pick $m \in \omega$. We need to find $n_m \in \omega$ such that

$$\forall n \geq n_m.\ \ T_n[m] \ =\ T_\infty[m].$$

Since $\{T_n\}_{n\in\omega}$ is Cauchy, there exists $k \geq 1$ such that

$$\forall n \geq k.\ \ T_n[m] \ =\ T_k[m].$$

We claim that k is the desired n_m. Let n be an index such that $n \geq k$. To show the inclusion

$$T_n[m] \ \supseteq\ T_\infty[m],$$

pick τ from $T_\infty[m]$. This means that $\tau = \tau'[m]$ for some $\tau' \in T_\infty$. Then, by the definition of T_∞, there must be a Cauchy sequence $\{\tau_i\}_{i\in\omega}$, that is taken from an infinite subsequence $\{T_{(k_i)}\}_{i\in\omega}$, and that converges to τ'. Thus,

$$\exists j \in \omega.\ \ \tau_j \in T_{(k_j)} \ \wedge\ \tau_j[m] = \tau'[m] \ \wedge\ (k_j \geq n).$$

Since $T_n[m] = T_k[m] = T_{(k_j)}[m]$, there exists $\tau'' \in T_n$ such that

$$\tau''[m] \ \ =\ \ \tau_j[m] \ \ =\ \ \tau'[m] \ \ =\ \ \tau.$$

Thus, $\tau \in T_n[m]$. It remains to show the other inclusion

$$T_n[m] \ \subseteq\ T_\infty[m].$$

Pick τ from $T_n[m]$. This means that $\tau = \tau'[m]$ for some $\tau' \in T_n$. By Lemma 5.15, there exists a Cauchy sequence $\{\tau_i\}_{i\in\omega}$ such that

1. $\tau_i[m] = \tau'[m]$ for all $i \in \omega$, and

2. the sequence is taken from an infinite subsequence of $\{T_n\}_{n\in\omega}$.

By definition, the limit τ_∞ of $\{\tau_i\}_{i\in\omega}$ should belong to T_∞. Furthermore, since the first m-prefixes of τ_i's equal $\tau'[m]$, we should also have that $\tau_\infty[m] = \tau'[m]$. Since $\tau'[m] = \tau$, it follows that $\tau \in T_\infty[m]$, as desired. □

Reference page not just equation number

Lemma 5.17. *The condition* (5.1) *of our framework on the distance and the pre-order holds for* $(\mathcal{D}, d_B^+, \subseteq, \mathsf{Traces})$.

Proof. Let $T \in \mathcal{D}$ and consider a Cauchy sequence $\{T_n\}_{n\in\omega}$ in $\mathcal{D}$ such that $T_n \subseteq T$ for all n. Also, let T_∞ be the limit of this sequence. We need to prove that $T_\infty \subseteq T$. Pick τ from T_∞. Since T_∞ is the limit of $\{T_n\}_{n\in\omega}$, we have that

$$\forall m \in \omega.\ \ \exists n \in \omega.\ \ T_n[m] \ = \ T_\infty[m].$$

Hence, for all $m \in \omega$, there exist n_m and $\tau_{(n_m)} \in T_{(n_m)}$ such that

$$\tau_{(n_m)}[m] \ = \ \tau[m].$$

Because $T_{(n_m)}$ is a subset of T, $\tau_{(n_m)}$ belongs to T as well. Furthermore, $\{\tau_{(n_i)}\}_{i\in\omega}$ is a Cauchy sequence converging to τ. This is because for all $m \in \omega$ and $k \geq m$, $\tau_{(n_k)}[k] = \tau[k]$, so

$$\tau_{(n_k)}[m] \ = \ \tau[m].$$

Now, the closedness of T implies that the limit τ of the Cauchy sequence $\{\tau_{(n_i)}\}_{i\in\omega}$ in T should belong to T as well. Since τ is chosen arbitrarily, this membership of τ to T means that $T_\infty \subseteq T$, as desired. □

Lemma 5.18. *The set union is the join operator (i.e., the least upper bound) of* $(\mathcal{D}, \subseteq, \mathsf{Traces})$.

Proof. We first prove that $\cup$ is a well-defined operator on $\mathcal{D}$. Pick $T_1, T_2 \in \mathcal{D}$. Then, both T_1 and T_2 are full. This implies that their union $T_1 \cup T_2$ is full as well. To show the closedness, consider a Cauchy sequence $\{\tau_n\}_{n\in\omega}$ in $T_1 \cup T_2$. Then, there is an infinite subsequence $\{\tau_{k_n}\}_{n\in\omega}$ of $\{\tau_n\}_{n\in\omega}$ such that either all of $\tau'_{k_n}s$ belong to T_1, or they all belong to T_2. This implies that the limit τ_∞ of the original sequence $\{\tau_n\}_{n\in\omega}$ should be in T_1 or T_2. That is, the limit is in $T_1 \cup T_2$. We have just shown that $T_1 \cup T_2$ is in $\mathcal{D}$.

Next, we show that $\cup$ is the join. This follows easily from the fact that the order on $\mathcal{D}$ is given by the subset relation. □

5.3.1.3 Sequencing Operator

We define the sequencing operator $\mathsf{seq} : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ by

$$\mathsf{seq}(T, T') \;=\; \{\, \tau\sigma\tau' \mid (\tau\sigma \in T \cap \mathsf{tagStates}^{+}) \;\wedge\; (\sigma\tau' \in T') \,\} \;\cup\; (T \cap \mathsf{tagStates}^{\infty}).$$

The operator takes two arguments T and T' in $\mathcal{D}$, and returns a set consisting of two kinds of traces: traces that have finite prefixes in the first argument T and suffices in the second T', and infinite traces from the first T only.

Lemma 5.19. $\mathsf{seq}(T, T')$ *consists of traces.*

Proof. Pick τ from $\mathsf{seq}(T, T')$. It is immediate from the definition of seq that τ is a pre-trace. We will show that τ satisfies the additional condition for traces as well. If τ belongs to $T \cap \mathsf{tagStates}^{\infty}$, it should be in T as well. This implies that τ is a trace. Suppose that $\tau \notin (T \cap \mathsf{tagStates}^{\infty})$. Then, there exist τ_0, σ_0, τ_1 such that

$$(\tau_0\sigma_0 \in (T \cap \mathsf{tagStates}^{+})) \;\;\wedge\;\; (\sigma_0\tau_1 \in T') \;\;\wedge\;\; \tau = \tau_0\sigma_0\tau_1.$$

Since $\sigma_0\tau_1$ is a pre-trace, σ_0 should be tagged with $none$. Furthermore, since $\tau_0\sigma_0$ is a finite trace, it has to be in $\mathcal{W}$. We now do the case analysis depending on whether $\sigma_0\tau_1$ is finite. If $\sigma_0\tau_1$ is finite, $\sigma_0\tau_1$ has to be in $\mathcal{W}$. Hence, $\tau = \tau_0\sigma_0\tau_1$ is a finite sequence belonging to $\mathcal{W}$. From this, it follows that τ is a trace. If $\sigma_0\tau_1$ is infinite, all prefixes of $\sigma_0\tau_1$ belong to $\mathcal{O}$. Thus, all prefixes of $\tau = \tau_0\sigma_0\tau_1$ also belong to $\mathcal{O}$. This implies that τ is a trace. □

Lemma 5.20. *For all* $T, T' \in \mathcal{D}$, $\mathsf{seq}(T, T')$ *is in* $\mathcal{D}$*, i.e., it is closed and full.*

Proof. Let T, T' be trace sets in $\mathcal{D}$. Firstly, we prove that $\mathsf{seq}(T, T')$ is full. Pick a $none$-tagged state $\sigma \in \mathsf{nStates}$. Since T is full, there is a trace τ in T that starts with σ. If τ is infinite, it also belongs to $\mathsf{seq}(T, T')$, so we have just found a trace in $\mathsf{seq}(T, T')$ starting with σ. If τ is finite, it must be of the form $\tau_0\sigma_0$ for some $none$-tagged state $\sigma_0 \in \mathsf{nStates}$. This is because τ is a finite pre-trace, so it should start and end with $none$-tagged states. But, T' is full. Hence,

$\sigma_0\tau' \in T'$ for some sequence τ' of tagged states. Now, the definition of $\mathsf{seq}(T, T')$ implies that

$$\tau_0\sigma_0\tau' \ \in \ \mathsf{seq}(T, T').$$

Since $\tau_0\sigma_0\tau'$ starts with σ, it is the trace that we are looking for.

Secondly, we show that $\mathsf{seq}(T, T')$ is closed. Consider a Cauchy sequence $\{\tau_n\}_{n\in\omega}$ in $\mathsf{seq}(T, T')$. Let τ_∞ be the limit of this sequence. We need to show that $\tau_\infty \in \mathsf{seq}(T, T')$. There are two cases to consider.

The first case is that there is an infinite subsequence $\{\tau_{(n_i)}\}_{i\in\omega}$ of $\{\tau_n\}_{n\in\omega}$ such that

$$\forall i \in \omega. \quad \tau_{(n_i)} \ \in \ (T \cap \mathsf{tagStates}^\infty).$$

Since the original sequence $\{\tau_n\}_{n\in\omega}$ is Cauchy, the subsequence $\{\tau_{(n_i)}\}_{i\in\omega}$ is Cauchy as well. Furthermore, the two sequences have the same limit τ_∞. This limit has to be an infinite sequence, because every member of $\{\tau_{(n_i)}\}_{i\in\omega}$ is infinite. It also belongs to T, since T is closed. Hence,

$$\tau_\infty \ \in \ (T \cap \mathsf{tagStates}^\infty) \ \subseteq \ \mathsf{seq}(T, T').$$

The second case is that all elements of $\{\tau_n\}_{n\in\omega}$ except finitely many are from

$$\{\, \tau\sigma\tau' \mid (\tau\sigma \in T \cap \mathsf{tagStates}^+) \ \wedge \ (\sigma\tau' \in T') \,\}.$$

This means that there is some $n_0 \in \omega$ such that

$$\forall n \geq n_0. \quad \exists \tau_n^0, \sigma_n, \tau_n^1. \quad (\tau_n = \tau_n^0\sigma_n\tau_n^1) \ \wedge \ (\tau_n^0\sigma_n \in T \cap \mathsf{tagStates}^+) \ \wedge \ (\sigma_n\tau_n^1 \in T').$$

We sub-divide this case based on whether there is some $u \in \omega$ with

$$\forall n \geq n_0. \quad |\tau_n^0\sigma_n| \ \leq \ u. \tag{5.16}$$

Suppose that there exists such an upper bound u. Since $\{\tau_n\}_{n\in\omega}$ is Cauchy, this implies that there is some $n_1 \geq n_0$ such that

$$\forall n \geq n_1. \quad (\tau_n^0\sigma_n = \tau_{n_1}^0\sigma_{n_1}).$$

In this sub-case, $\{\sigma_n\tau_n^1\}_{n\geq n_1}$ is also Cauchy, its limit τ'_∞ starts with σ_n, and it satisfies the below relationship with the limit τ_∞ of $\{\tau_n\}_{n\in\omega}$:

$$\tau_{n_1}^0\tau'_\infty \;=\; \tau_\infty.$$

Note that the sequence $\{\sigma_n\tau_n^1\}_{n\geq n_1}$ is in T', which is a closed set. Thus, τ'_∞ is in T'. Because τ'_∞ starts with σ_{n_1} and $\tau_{n_1}\sigma_{n_1}$ is in T, we have that

$$\tau_\infty \;=\; \tau_{n_1}\tau'_\infty \;\in\; \mathsf{seq}(T,T').$$

The other sub-case is that there does not exist u satisfying (5.16). In this sub-case, $\{\tau_n^0\sigma_n\}_{n\geq n_0}$ becomes a Cauchy sequence in T with τ_∞ as its limit. Since T is closed, τ_∞ is in T. Also, $|\tau_n^0\sigma_n|$ goes to the infinity as n increases, so τ_∞ belongs to $T\cap\mathsf{tagStates}^\infty$. Hence, τ_∞ is in $\mathsf{seq}(T,T')$, as desired. □

Lemma 5.21. *The function* seq *is non-expansive.*

Proof. By the definition of the distance d_B^+ on $\mathcal{D}$, proving the non-expansive of seq is equivalent to showing that for all T_0,T'_0,T_1,T'_1 in $\mathcal{D}$ and all $m\in\omega$,

$$(T_0[m] \;=\; T_1[m] \;\wedge\; T'_0[m] \;=\; T'_1[m]) \;\Longrightarrow\; (\mathsf{seq}(T_0,T'_0)[m] \;=\; \mathsf{seq}(T_1,T'_1)[m]).$$

Let T_0,T'_0,T_1,T'_1,m be the data in the above equivalent statement, and assume the condition of the implication. We need to show that $\mathsf{seq}(T_0,T'_0)[m] \;=\; \mathsf{seq}(T_1,T'_1)[m]$. We will prove that $\mathsf{seq}(T_0,T'_0)[m] \;\subseteq\; \mathsf{seq}(T_1,T'_1)[m]$. The other subset inclusion can be proved similarly. Pick $\tau\in\mathsf{seq}(T_0,T'_0)[m]$. This means that

$$\exists\tau'\in\mathsf{seq}(T_0,T'_0).\quad \tau'[m] \;=\; \tau.$$

Since $\tau'\in\mathsf{seq}(T_0,T'_0)$, we have

$$(\tau'\in T_0\cap\mathsf{tagStates}^\infty) \;\vee\; (\exists\tau_0,\sigma,\tau'_0.\;\tau'=\tau_0\sigma\tau'_0 \;\wedge\; \tau_0\sigma\in T_0 \;\wedge\; \sigma\tau'_0\in T'_0). \tag{5.17}$$

Suppose that the first disjunct holds. Since $T_0[m]=T_1[m]$, there is $\tau''\in T_1$ such that

$$|\tau''|\geq m \quad\wedge\quad \tau''[m] \;=\; \tau'[m] \;=\; \tau.$$

If τ'' is infinite, it is also in $\mathsf{seq}(T_1, T_1')$. So, $\tau = \tau''[m] \in \mathsf{seq}(T_1, T_1')[m]$ as desired. Consider the other case that τ'' is finite. In this case, we note two facts. Firstly, since τ'' is a trace, it should end with a $none$-tagged state, say, $\sigma \in \mathsf{nStates}$. Secondly, since T_1 is full, there is $\sigma\tau''' \in T_1'$ for some sequence τ''' of tagged states. Hence, $\tau''\tau'''$ is in $\mathsf{seq}(T_1, T_1')$. But $|\tau''| \geq m$, which means that

$$(\tau''\tau''')[m] \;=\; \tau''[m] \;=\; \tau.$$

So, τ is in $\mathsf{seq}(T_1, T_1')$.

Now, suppose that the second disjunct of (5.17) holds. Let τ_0, σ, τ_0' be the witnesses of the existential quantification in (5.17). Since $m \geq 1$, $T_0[m] = T_1[m]$ and $T_0'[m] = T_1'[m]$,

$$\exists \tau_1, \tau_1'.\;\; \tau_1 \in T_1 \;\wedge\; (\sigma\tau_1') \in T_1' \;\wedge\; (\tau_0\sigma)[m] = \tau_1[m] \;\wedge\; (\sigma\tau_0')[m] = (\sigma\tau_1')[m].$$

Let m_0 be $|\tau_0\sigma|$. If $m_0 \geq m$, we can ignore τ_1', and complete the proof similarly as in the previous case, just doing the case-analysis on whether τ_1 is infinite or not. Suppose that $m_0 < m$. In this case,

$$\tau_1 \;=\; \tau_0\sigma \quad\wedge\quad \tau_0'[m - m_0] \;=\; \tau_1'[m - m_0].$$

Thus,

$$\tau \;=\; \tau'[m] \;=\; (\tau_0\sigma)(\tau_0'[m - m_0]) \;=\; (\tau_1)(\tau_1'[m - m_0]) \;=\; (\tau_1\tau_1')[m].$$

Since $\tau_1\tau_1' \in \mathsf{seq}(T, T')$, this means that $\tau \in \mathsf{seq}(T, T')[m]$. □

Lemma 5.22. *For all $T \in \mathcal{D}$, if all traces in T have length at least 2 (i.e., $\forall \tau \in T.\, |\tau| \geq 2$), the specialization $\mathsf{seq}(T, -)$ by T is $1/2$-contractive on $\mathcal{D}$, i.e.,*

$$\forall T_1, T_2 \in \mathcal{D}.\;\; d(\mathsf{seq}(T, T_1),\, \mathsf{seq}(T, T_2)) \;\;\leq\;\; (1/2 \times d(T_1, T_2)).$$

Proof. Let T be a trace set satisfying the condition in the lemma. Pick T_1, T_2 from $\mathcal{D}$. We need to prove that:

$$d_B^+(\mathsf{seq}(T, T_1), \mathsf{seq}(T, T_2)) \;\leq\; (\frac{1}{2} \times d_B^+(T_1, T_2)). \tag{5.18}$$

Let A and B be trace sets defined by

$$\begin{aligned} A \;&=\; \{\, \tau\sigma\tau' \;\mid\; (\tau\sigma \in T \cap \mathsf{tagStates}^+) \;\wedge\; (\sigma\tau' \in T_1) \,\}, \\ B \;&=\; \{\, \tau\sigma\tau' \;\mid\; (\tau\sigma \in T \cap \mathsf{tagStates}^+) \;\wedge\; (\sigma\tau' \in T_2) \,\}. \end{aligned}$$

Then, by the definition of seq,

$$\mathsf{seq}(T, T_1) \;=\; A \cup (T \cap \mathsf{tagStates}^\infty) \quad \text{and} \quad \mathsf{seq}(T, T_2) \;=\; B \cup (T \cap \mathsf{tagStates}^\infty).$$

Thus, to prove (5.18), it is sufficient to show that for all $m \in \omega$,

$$T_1[m] = T_2[m] \implies \big((A \cup (T \cap \mathsf{tagStates}^\infty))[m+1] = (B \cup (T \cap \mathsf{tagStates}^\infty))[m+1]\big). \quad (5.19)$$

Suppose that $T_1[m] = T_2[m]$. We will show that $A[m+1] = B[m+1]$. From this, the equality in the conclusion of the implication (5.19) follows, because the "$-[m+1]$" operator distributes over $\cup$.

Here we will show only one inclusion $A[m+1] \subseteq B[m+1]$; the other inclusion can be shown similarly. Suppose that we have a trace $\tau \in A[m+1]$. This means that there is a trace $\tau' \in A$ such that

$$\tau \;=\; (\tau'[m+1]).$$

By the definition of A,

$$\exists \tau_0, \sigma_0, \tau_1.\;\; (\tau' = \tau_0\sigma_0\tau_1) \;\wedge\; (\tau_0\sigma_0 \in T \cap \mathsf{tagStates}^+) \;\wedge\; (\sigma_0\tau_1 \in T_1). \quad (5.20)$$

Since $T_1[m] = T_2[m]$ by assumption (and $m \geq 1$), we also have that

$$\exists \tau_2.\;\; (\sigma_0\tau_2 \in T_2) \;\wedge\; (\sigma_0\tau_2[m] \;=\; \sigma_0\tau_1[m]). \quad (5.21)$$

Note that $\tau_0\sigma_0\tau_2 \in B$ by the definition of B, (5.20) and (5.21). Since $\tau = (\tau'[m+1])$ and $\tau' = \tau_0\sigma_0\tau_1$, we can show the desired $\tau \in B[m+1]$, if we prove that

$$(\tau_0\sigma_0\tau_1)[m+1] \;=\; (\tau_0\sigma_0\tau_2)[m+1]. \quad (5.22)$$

Now, recall that T contains traces of length at least 2, so $|\tau_0\sigma_0| \;\geq\; 2$. This means that the second conjunct of (5.21) implies (5.22). □

Lemma 5.23. *The operator* seq *is* $\subseteq$*-monotone*

Proof. In the definition of $\mathsf{seq}(T, T')$, the argument trace sets T and T' appear only in positive

positions, i.e., they do not occur under the left of implication or under negation. The monotonicity follows from this. □

5.3.1.4 Conditional Statement

For each boolean condition B, we define the semantic conditional statement $\mathsf{if_B}$ in a standard way:

$$\mathsf{if_B}(T_0, T_1) \;=\; \{\, \sigma\tau \mid (\sigma\tau \in T_0 \;\wedge\; [\![B]\!](\mathsf{first}(\sigma)) = \mathsf{true}) \\ \vee\; (\sigma\tau \in T_1 \;\wedge\; [\![B]\!](\mathsf{first}(\sigma)) = \mathsf{false})\}$$

Here we assume the semantics $[\![B]\!]$ of boolean expressions B given by maps from untagged states (i.e., those in $\mathsf{Vars} \to \mathsf{Rationals}$) to $\{\mathsf{true}, \mathsf{false}\}$. Note that since $\mathsf{if_B}(T_0, T_1)$ is a subset of $T_0 \cup T_1$, it contains traces only.

Lemma 5.24. *For all $T_0, T_1 \in \mathcal{D}$, the trace set $\mathsf{if_B}(T_0, T_1)$ is in $\mathcal{D}$, i.e., it is closed and full.*

Proof. Let $T = \mathsf{if_B}(T_0, T_1)$. We consider the closedness property of T first. Consider a Cauchy sequence $\{\tau_i\}_{i\in\omega}$ in T. By the definition of Cauchy sequence, there exists an index n in ω such that

$$\forall i \geq n. \quad \tau_i[1] = \tau_n[1].$$

Let σ be $\tau_n[1]$. If $[\![B]\!](\mathsf{first}(\sigma)) = \mathsf{true}$, all the elements of $\{\tau_i\}_{i\in\omega}$ except the first $n-1$ belong to T_0 and have the same σ as their starting state. This implies that the limit τ_∞ of the sequence belongs to T_0 and it has σ as its starting state. But $[\![B]\!](\mathsf{first}(\sigma)) = \mathsf{true}$ by assumption. Thus, τ_∞ is also in T. The other case that $[\![B]\!](\mathsf{first}(\sigma)) = \mathsf{false}$ is similar.

Next, we prove that T is full. Pick a $none$-tagged state $\sigma \in \mathsf{nStates}$. We consider the case that $[\![B]\!](\mathsf{first}(\sigma)) = \mathsf{false}$ only, because the other case $[\![B]\!](\mathsf{first}(\sigma)) = \mathsf{true}$ can be proved similarly. Since T_1 is full, there is a trace of the form $\sigma\tau \in T_1$. Since $[\![B]\!](\mathsf{first}(\sigma)) = \mathsf{false}$, the trace $\sigma\tau$ is also included in T. We have just found a trace in T that starts with σ. □

Lemma 5.25. *The function $\mathsf{if_B}$ is non-expansive.*

Proof. By the definition of distance d_B^+ on $\mathcal{D}$, proving the non-expansiveness is equivalent to showing that for all (T_0, T_1) and (T_0', T_1') in $\mathcal{D} \times \mathcal{D}$ and all $m \in \omega$,

$$(T_0[m] \;=\; T_0'[m] \;\wedge\; T_1[m] \;=\; T_1'[m]) \;\;\Longrightarrow\;\; (\mathsf{if_B}(T_0, T_1)[m] \;=\; \mathsf{if_B}(T_0', T_1')[m]).$$

Pick $\sigma\tau$ from $\mathsf{if}_{\mathsf{B}}(T_0, T_1)[m]$. Since $m \geq 1$, this means that there exists $\sigma\tau'$ in $\mathsf{if}_{\mathsf{B}}(T_0, T_1)$ such that

$$\sigma\tau \;=\; (\sigma\tau')[m].$$

If $[\![B]\!](\mathsf{first}(\sigma)) = \mathsf{true}$, the trace $\sigma\tau'$ is from T_0. Since $T_0[m] = T_0'[m]$ (and $m \geq 1$), there is $\sigma\tau'' \in T_0'$ such that

$$(\sigma\tau'')[m] \;=\; (\sigma\tau')[m] \;=\; \sigma\tau.$$

Furthermore, since $[\![B]\!](\mathsf{first}(\sigma)) = \mathsf{true}$, this trace $\sigma\tau''$ should be in $\mathsf{if}_{\mathsf{B}}(T_0', T_1')$ as well. Putting all these together, we can conclude that

$$\sigma\tau \;=\; (\sigma\tau'')[m] \;\in\; \mathsf{if}_{\mathsf{B}}(T_0'.T_1').$$

The case that $[\![B]\!](\mathsf{first}(\sigma)) = \mathsf{false}$ can be proved similarly. Hence,

$$\mathsf{if}_{\mathsf{B}}(T_0, T_1)[m] \;\subseteq\; \mathsf{if}_{\mathsf{B}}(T_0', T_1')[m]$$

Using a similar argument, we can prove the other inclusion. □

Lemma 5.26. *The operator* if_{B} *is monotone with respect to the subset order* $\subseteq$.

Proof. In the definition of $\mathsf{if}_{\mathsf{B}}(T_0, T_1)$, the argument trace sets T_0 and T_1 are used only positively. From this, the monotonicity of the lemma follows. □

5.3.1.5 Function procrun for Procedure Invocations

For a $none$-tagged state σ and a procedure name $f \in \mathsf{PNames}$, let $\sigma^{(f,call)}$ and $\sigma^{(f,ret)}$ be, respectively, $(\mathsf{first}(\sigma), (f, call))$ and $(\mathsf{first}(\sigma), (f, ret))$. That is, this superscript notation replaces the tag of $none$-tagged states by the one for procedure call or return. Using this tag replacement, we define the procrun function for procedure f:

$$\begin{aligned}\mathsf{procrun}_f(T) \;=\; &\{\; \sigma\sigma^{(f,call)}\sigma^{(f,ret)}\sigma \;\mid\; \sigma \in T \;\} \\ &\cup\; \{\; \sigma\sigma^{(f,call)}\tau\sigma_1^{(f,ret)}\sigma_1 \;\mid\; \sigma\tau\sigma_1 \in (T \cap \mathsf{tagStates}^{+}) \;\} \\ &\cup\; \{\; \sigma\sigma^{(f,call)}\tau \;\mid\; \sigma\tau \in (T \cap \mathsf{tagStates}^{\infty}) \;\}.\end{aligned}$$

Lemma 5.27. $\mathsf{procrun}_f(T)$ *consists of traces only.*

Proof. Note that the definition of $\mathsf{procrun}_f(T)$ is given by the union of three sets. The first two sets there consist of finite pre-traces, because all traces in T start and end with $none$-tagged states and so, σ, σ_1 in the description of the first two sets have $none$ as their tags. Furthermore, they are subsets of $\mathcal{W}$, because all finite traces in T belong to $\mathcal{W}$. Hence, the sets contain traces only. For the third set in the definition of $\mathsf{procrun}_f(T)$, we note that all pre-traces there are infinite, because again traces in T start with $none$-tagged states and so σ in the definition of the third set should be tagged with $none$. Furthermore, all prefixes of infinite traces in T belong to $\mathcal{O}$, so that all prefixes of traces in the third set should be in $\mathcal{O}$ as well. From these observation, it follows that $\mathsf{procrun}_f(T)$ consists of traces. □

Lemma 5.28. *For all procedures $f \in \mathsf{PNames}$ and $T \in \mathcal{D}$, $\mathsf{procrun}_f(T)$ belongs to $\mathcal{D}$. That is, it contains traces only, and it is full and closed. Furthermore, for all $f \in \mathsf{PNames}$, function $\mathsf{procrun}_f(-)$ is $1/2$-contractive and monotone.*

Proof. Pick $f \in \mathsf{PNames}$ and $T \in \mathcal{D}$. Firstly, we prove that $\mathsf{procrun}_f(T)$ is full and closed. Let

$$\begin{aligned} \mathsf{prolog}_f &= \{\, \sigma\sigma^{(f,call)}\sigma \mid \sigma \in \mathsf{nStates} \,\}, \\ \mathsf{epilog}_f &= \{\, \sigma\sigma^{(f,ret)}\sigma \mid \sigma \in \mathsf{nStates} \,\}. \end{aligned}$$

Note that these sets consist of pre-traces, not traces. But,

$$\mathsf{procrun}_f(T) = \mathsf{seq}(\mathsf{seq}(\mathsf{prolog}_f, T), \mathsf{epilog}_f)$$

when we use the definition of seq for sets of pre-traces as well. Furthermore, the proof of Lemma 5.20 does not rely on the fact that its arguments contain traces only, so it also works when we change the lemma such that the arguments of seq are full closed sets of *pre-traces*. Hence, to show that $\mathsf{procrun}_f(T)$ is full and closed, it is sufficient to prove that prolog_f and epilog_f are full and closed. Note that all sequences in prolog_f or epilog_f have length 3. So, every Cauchy sequence in the sets converges to the n-th element in the sequence for some $n \in \omega$, which means that prolog_f and epilog_f are closed. The remaining condition that prolog_f and epilog_f are full is an immediate consequence of their definitions.

Secondly, we prove that $\mathsf{procrun}_f(-)$ is $\frac{1}{2}$-contractive and monotone. Recall that $\mathsf{procrun}_f(T)$ is $\mathsf{seq}(\mathsf{seq}(\mathsf{prolog}_f, T), \mathsf{epilog}_f)$ for all T. We again rely on the observation that the proofs of Lemmas 5.21 and 5.22 can be generalized to full closed sets of pre-traces. Both proofs are independent of the fact that the arguments of seq consist of traces. They work equally well, when we

change the lemmas such that seq takes full closed sets of *pre-traces* as its parameters. Hence, the generalization of Lemma 5.21 and 5.22 implies the $1/2$-contractiveness of $\mathsf{procrun}_f(-)$, because every pre-trace in prolog_f has length greater than 2. The remaining monotonicity condition is immediate from the definition of $\mathsf{procrun}_f(-)$. □

5.3.1.6 Interpretation of Atomic Commands

In this concrete semantics, we interpret atomic commands as follows:

$$\begin{aligned}\mathsf{trans}(x := E) &= \{\ \sigma\sigma' \mid \sigma, \sigma' \in \mathsf{nStates} \ \wedge\ \mathsf{first}(\sigma') = \mathsf{first}(\sigma)[x \mapsto [\![E]\!]\mathsf{first}(\sigma)]\ \}\\ \mathsf{trans}(x := *) &= \{\ \sigma\sigma' \mid \sigma, \sigma' \in \mathsf{nStates} \ \wedge\ \mathsf{first}(\sigma') = \mathsf{first}(\sigma)[x \mapsto r] \ \wedge\ r \in \mathsf{Rationals}\ \}\end{aligned}$$

In the interpretation, we assume the standard semantics $[\![E]\!]$ for expressions E, given by a map from untagged states to rational numbers.

Lemma 5.29. *Both* $\mathsf{trans}(x := E)$ *and* $\mathsf{trans}(x := *)$ *are in* $\mathcal{D}$.

Proof. From the definitions of trans, it is immediate that $\mathsf{trans}(x := E)$ and $\mathsf{trans}(x := *)$ consist of traces and that they are full. For the closedness, we note that $\mathsf{trans}(x := E)$ and $\mathsf{trans}(x := *)$ contain traces of size 2. Thus, every Cauchy sequence $\{\tau_n\}_{n\in\omega}$ in those sets should contain some τ_i that is the limit of the sequence. From this property of Cauchy sequence, the closedness follows. □

5.3.1.7 Liveness Property

We say that a trace τ includes an infinite subsequence of open calls if there exists $\{\tau_n\sigma_n\}_{n\in\omega}$ such that

1. $\tau = \tau_1\sigma_1\tau_2\sigma_2\tau_3\sigma_3 \ldots \tau_n\sigma_n \ldots$,
2. for all $i \in \omega$, there exists some $f \in \mathsf{PNames}$ such that $\mathsf{second}(\sigma_i) = (f, \mathit{call})$, and
3. for all $i \in \omega$, the corresponding return for σ_i does not appear in τ after σ_i, i.e., the return does not occur in the sequence $\tau_{i+1}\sigma_{i+1}\tau_{i+2}\sigma_{i+2}\ldots$.

$$
\begin{array}{lcl}
E & ::= & r \mid x \mid \text{`}x \mid x' \mid E + E \mid r \times E \\
P & ::= & E = E \mid E \neq E \mid E < E \mid E > E \mid E \leq E \mid E \geq E \\
\varphi & ::= & P \mid \mathsf{true} \mid \varphi \wedge \varphi \mid \mathsf{false} \mid \varphi \vee \varphi \mid \exists x'.\, \varphi
\end{array}
$$

FIGURE 5.4: Syntax for Linear Constraints. r represents rational numbers

We specify a desired liveness property of (semantic) commands, using the following subset LIVPROPERTY of $\mathcal{D}$:[6]

$T \in$ LIVPROPERTY $\iff$

For all $\tau \in T$, the trace τ does not include an infinite subsequence of open calls.

5.3.2 Abstract Semantics with Linear Ranking Functions

In this section, we will define an instance abstract semantics of our framework. The instance is based on linear ranking functions, and it gives an abstract interpreter for proving program termination.

Recall that linear ranking functions are also used in the abstract interpreters in Chapter 4. However, there will be noticeable differences between the abstract semantics in this section and the abstract interpreters in Chapter 4, because the former has to deal with new requirements coming from the metric space structure of the concrete semantics. For instance, when the abstract domains in Chapter 4 are adjusted to the setting of this chapter, elements in the adjusted domains mean (i.e., concretize into) *non-closed* sets of traces, unless they are $\top$. This means that the abstract interpreters from those domains and the framework of this chapter cannot prove termination of any programs. In order to prove the termination of a program, an abstract domain should find non-$\top$ pre-fixpoints of recursive procedures in the programs.

The abstract semantics in this section is built using the syntax defined in Figure 5.4 where φ expresses linear constraints. Note that the linear constraints can use three kinds of variables: normal program variables x; pre-primed ones $\text{`}x$ for denoting the value of $x$ before running a program; primed ones $x'$ that can be existentially quantified. We assume that the set $\mathsf{Vars}$ of normal variables and the set $\text{`}\mathsf{Vars}$ of pre-primed variables are finite and that there is an one-to-one correspondence between Vars and $\text{`}\mathsf{Vars}$, which maps $x$ to $\text{`}x$.

Let $\mathsf{LinForm}$ be the set of formulas φ that do not contain primed variables. Each $\varphi \in \mathsf{LinForm}$ defines a relation from untagged states with pre-primed variables (i.e., $\text{`}\mathsf{Vars} \to \mathsf{Rationals}$) to

[6] In general, the property is different from the usual termination requirement that all traces in T should be finite. Some $T \in$ LIVPROPERTY could contain an infinite trace, as long as the trace does not have an infinite subsequence of open calls. However, if we restrict our attention to $T = [\![C]\!]\eta$ of some command C with no free procedure names, the membership $T \in$ LIVPROPERTY does imply that T consists of finite traces only. This is because every infinite trace in such T's includes an infinite subsequence of open calls.

untagged states with normal variables (i.e., ‘Vars $\rightarrow$ Rationals). This means that the semantics of a formula φ in LinForm is given by the satisfaction relation

$$(\text{‘}s, s) \models \varphi,$$

where ‘s is an element in ‘Vars $\rightarrow$ Rationals. The satisfaction relation $\models$ is standard, so we omit it in this section. For $\varphi, \psi \in \mathsf{LinForm}$, we write $\varphi \models \psi$ to express that for all $(\text{‘}s, s)$, if $(\text{‘}s, s) \models \varphi$, then $(\text{‘}s, s) \models \psi$.

Define TLinForm to be a subset of LinForm consisting of total formulas:

$$\mathsf{TLinForm} = \{ \varphi \in \mathsf{LinForm} \mid \forall \text{‘}s \in (\text{‘}\mathsf{Vars} \rightarrow \mathsf{Rationals}).\ \exists s \in (\mathsf{Vars} \rightarrow \mathsf{Rationals}).\ (\text{‘}s, s) \models \varphi \}.$$

We assume a sound but possibly-incomplete theorem prover that can answer the queries of the following types:

$$\varphi \vdash \psi, \qquad \varphi \vdash \mathsf{false}, \qquad \vdash \forall \text{‘}X.\, \exists X.\, \varphi.$$

where ‘X and X are the sets of free pre-primed variables and normal variables in φ, respectively. Note that by asking the query of the last type, we can use a prover to check, soundly, whether a formula φ belongs to TLinForm.

By using what we have defined or assumed so far, we define the abstract domain $\mathcal{A}$ and its subset $\mathcal{A}_t$ of total abstract elements as follows:

$$\begin{aligned} \mathcal{A} &= \mathsf{LinForm} \times \mathsf{LinForm} \times \mathsf{LinForm}, \\ \mathcal{A}_t &= \mathsf{TLinForm} \times \mathsf{LinForm} \times \mathsf{LinForm} \end{aligned}$$

The element $(\mathsf{false}, \mathsf{false}, \mathsf{false})$ in $\mathcal{A}$ serves the role of $\bot$, and $(\mathsf{true}, \mathsf{true}, \mathsf{true})$ the role of $\top$. The algorithm for soundly checking the totality of abstract elements is defined using the assumed prover as follows:

$$\mathsf{checktot}(\varphi_1, \varphi_2, \varphi_3) = \begin{cases} \mathsf{true} & \text{if } \vdash \forall \text{‘}X.\exists X.\, \varphi_1 \\ \mathsf{unknown} & \text{otherwise} \end{cases}$$

where ‘X and X are the sets of free pre-primed and normal variables in φ_1.

The meaning of each total abstract element is given by the concretization map γ. To describe γ, we need to introduce some notations and terminologies. For a tagged state σ, let ‘σ be the one obtained from σ by renaming normal variables by corresponding pre-primed ones. For each trace τ, define $\mathsf{first}(\tau)$ and $\mathsf{last}(\tau)$ to be the first and the last states of a trace τ; if τ is infinite, $\mathsf{last}(\tau)$ is not defined. Write $\sigma \in \tau$ to mean that σ is a tagged state appearing in τ, and $\mathsf{iscall}(\sigma)$

to mean that the tag for σ is a procedure call:

$$\mathsf{iscall}(\sigma) \iff \exists f \in \mathsf{PNames}.\ (\mathsf{second}(\sigma) = (f, \mathit{call})).$$

Finally, for all tagged states σ_1, σ_2 in a trace τ, we say that σ_1 is an open call with respect to σ_2 in τ, denoted $\mathsf{open}(\sigma_1, \sigma_2, \tau)$, if both σ_1 and σ_2 are tagged with procedure calls, σ_1 appears strictly before σ_2 in τ but the corresponding return for σ_1 does not appear before σ_2. The concretization function is defined as follows:

$$\begin{aligned}
&\gamma(\varphi_1, \varphi_2, \varphi_3) = \\
&\quad \{\, \tau \in \mathsf{Traces} \mid (\tau \in \mathsf{tagStates}^+ \implies (\text{`}\mathsf{first}(\mathsf{first}(\tau)), \mathsf{first}(\mathsf{last}(\tau))) \models \varphi_1) \ \wedge \\
&\qquad\qquad (\forall \sigma \in \tau.\, \mathsf{iscall}(\sigma) \implies (\text{`}\mathsf{first}(\mathsf{first}(\tau)), \mathsf{first}(\sigma)) \models \varphi_2) \ \wedge \\
&\qquad\qquad \forall \sigma_1, \sigma_2.\, \mathsf{open}(\sigma_1, \sigma_2, \tau) \implies (\text{`}\mathsf{first}(\sigma_1), \mathsf{first}(\sigma_2)) \models \varphi_3 \,\}.
\end{aligned}$$

Lemma 5.30. *For every $(\varphi_1, \varphi_2, \varphi_3) \in \mathcal{A}$, we have that*

$$(\varphi_1, \varphi_2, \varphi_3) \in \mathcal{A}_t \implies \gamma(\varphi_1, \varphi_2, \varphi_3) \in \mathcal{D}.$$

Proof. Consider $(\varphi_1, \varphi_2, \varphi_3)$ in $\mathcal{A}_t$. Let T be $\gamma(\varphi_1, \varphi_2, \varphi_3)$. Firstly, we prove that T is full. Since $(\varphi_1, \varphi_2, \varphi_3) \in \mathcal{A}_t$, its first component φ_1 should be total:

$$\forall \text{`}s \in (\text{`}\mathsf{Vars} \to \mathsf{Rationals}).\ \exists s \in (\mathsf{Vars} \to \mathsf{Rationals}).\ (\text{`}s, s) \models \varphi_1.$$

This implies that for every none-tagged state σ_0, there exists a none-tagged state σ_1 such that

$$(\text{`}\mathsf{first}(\sigma_0), \mathsf{first}(\sigma_1)) \models \varphi_1.$$

Furthermore, when $\sigma_0\sigma_1$ is viewed as a trace, it does not contain any procedure calls, so it satisfies the requirements imposed by φ_2 and φ_3. Hence, $\sigma_0\sigma_1$ is in T. Note that σ_0 is chosen arbitrarily. Hence, we have just shown that T is full.

Next, we show that T is closed. Let $\{\tau_n\}_{n\in\omega}$ be a Cauchy sequence in T, and let τ_∞ be the limit of the sequence. We will prove that τ_∞ is in T. If τ_∞ is finite, it has to be the same as some τ_n in the sequence. Thus, τ_∞ is in T. Suppose that τ_∞ is infinite. We have to prove that τ_∞ satisfies the two requirements imposed by φ_2 and φ_3. To discharge the requirement from φ_2, pick $\sigma \in \tau_\infty$ such that the tag of σ is a procedure call. Let m be the position of σ in the trace τ_∞. Then, since τ_∞ is the limit of $\{\tau_n\}_{n\in\omega}$, there exists τ_n such that $\tau_n[m] = \tau_\infty[m]$. But, τ_n

is in T, and so,

$$(\text{`}\mathsf{first}(\mathsf{first}(\tau_n)), \mathsf{first}(\sigma)) \;\models\; \varphi_2.$$

The LHS of $\models$ is the same as $(\text{`}\mathsf{first}(\mathsf{first}(\tau_\infty)), \mathsf{first}(\sigma))$. Hence, the requirement from φ_2 holds for τ_∞. Now, it remains to prove that τ_∞ satisfies the requirement from φ_3. Pick σ_1, σ_2 such that $\mathsf{open}(\sigma_1, \sigma_2, \tau_\infty)$. Let m be the position of σ_2 in the trace τ_∞. Again, we use the fact that τ_∞ is the limit of $\{\tau_n\}_{n\in\omega}$, so there exists τ_n such that $\tau_n[m] = \tau_\infty[m]$, which implies that $\mathsf{open}(\sigma_1, \sigma_2, \tau_n)$. Thus,

$$(\text{`}\mathsf{first}(\sigma_1), \mathsf{first}(\sigma_2)) \;\models\; \varphi_3.$$

This completes the proof that τ_∞ satisfies the condition for φ_3. □

Having defined the abstract domain we now turn our attention to the abstract operators of our language, and prove that they meet the requirements of our framework.

5.3.2.1 Abstract Sequencing Operator

To define an abstract sequencing operator, we need to define the relational composition of formulas $\varphi, \psi \in \mathsf{LinForm}$:

$$\varphi; \psi \;=\; \exists Y'.(\varphi[Y'/X] \wedge \psi[Y'/\text{`}X]),$$

where X and $\text{`}X$ respectively contain normal variables in $\varphi$ and pre-primed variables in $\psi$, $Y'$ is the set of fresh primed variables, and the cardinalities of these three sets are the same so that the substitution in $\varphi; \psi$ is well-defined. For instance, $(\text{`}x \geq x); (\text{`}x + 1 = y)$ becomes $\exists x'.\ (\text{`}x \geq x') \wedge (x' + 1 = y)$, which means the relational composition of two relations $(\text{`}x \geq x)$ and $(\text{`}x + 1 = y)$.

We define the abstract sequencing as follows:

$$\mathsf{seq}^\sharp\big((\varphi_1, \varphi_2, \varphi_3),\ (\psi_1, \psi_2, \psi_3)\big) \;=\; \big(\varphi_1; \psi_1,\ \varphi_2 \vee (\varphi_1; \psi_2),\ \varphi_3 \vee \psi_3\big).$$

The following two lemmas show that $\mathsf{seq}^\sharp$ satisfies the conditions from our framework.

Lemma 5.31. *If both $(\varphi_1, \varphi_2, \varphi_3)$ and (ψ_1, ψ_2, ψ_3) are in $\mathcal{A}_t$, their sequential composition $\mathsf{seq}^\sharp\big((\varphi_1, \varphi_2, \varphi_3),\ (\psi_1, \psi_2, \psi_3)\big)$ is in $\mathcal{A}_t$ as well.*

Proof. Suppose that $(\varphi_1, \varphi_2, \varphi_3)$ and (ψ_1, ψ_2, ψ_3) are in $\mathcal{A}_t$. This means that both φ_1 and ψ_1 belong to $\mathsf{TLinForm}$. Then, $\varphi_1; \psi_1$ is also in $\mathsf{TLinForm}$, because the $-;-$ operator for formulas

means the composition of state relations and the composition of two total relations is total. Hence,

$$\mathsf{seq}^\sharp\big((\varphi_1,\varphi_2,\varphi_3),\ (\psi_1,\psi_2,\psi_3)\big) \ = \ \big(\varphi_1;\psi_1,\ \ \varphi_2 \vee (\varphi_1;\psi_2),\ \ \varphi_3 \vee \psi_3\big)$$

belongs to $\mathcal{A}_t$ as well. □

Lemma 5.32. *For all* $(\varphi_1,\varphi_2,\varphi_3)$ *and* (ψ_1,ψ_2,ψ_3) *in* $\mathcal{A}_t$*, we have that*

$$\mathsf{seq}\big(\,\gamma(\varphi_1,\varphi_2,\varphi_3),\ \gamma(\psi_1,\psi_2,\psi_3)\,\big) \ \subseteq \ \gamma\big(\,\mathsf{seq}^\sharp\big(\,(\varphi_1,\varphi_2,\varphi_3),\ (\psi_1,\psi_2,\psi_3)\,\big)\,\big).$$

Proof. Let $T_0 = \gamma(\varphi_1,\varphi_2,\varphi_3)$ and $T_1 = \gamma(\psi_1,\psi_2,\psi_3)$. Pick a trace τ from $\mathsf{seq}(T_0,T_1)$. By the definition of $\mathsf{seq}^\sharp$,

$$\mathsf{seq}^\sharp\big((\varphi_1,\varphi_2,\varphi_3),\ (\psi_1,\psi_2,\psi_3)\big) \ = \ \big(\varphi_1;\psi_1,\ \varphi_2 \vee (\varphi_1;\psi_2),\ \varphi_3 \vee \psi_3\big).$$

Hence, it suffices to prove that τ satisfies the three requirements in the definition of γ, which are determined by $(\varphi_1;\psi_1)$, $\varphi_2 \vee (\varphi_1;\psi_2)$, and $(\varphi_3 \vee \psi_3)$. To do this, we do the case analysis on τ.

1. The first case is that $\tau = \tau_0\sigma\tau_1$ for some *finite* trace $\tau_0\sigma$ in T_0 and a trace $\sigma\tau_1$ in T_1. Let's start with the first requirement given by $\varphi_1;\psi_1$:

$$\tau \in \mathsf{tagStates}^+ \implies (\text{`}\mathsf{first}(\mathsf{first}(\tau)),\ \mathsf{first}(\mathsf{last}(\tau))) \models (\varphi_1;\psi_1), \tag{5.23}$$

   Note that when τ is infinite, the requirement holds vacuously. Suppose that τ is finite. In this case, the suffix $\sigma\tau_1$ is finite as well. Since $\tau_0\sigma \in T_0$, $\sigma\tau_1 \in T_1$ and both traces are finite, these two traces satisfy the below condition imposed by φ_1 and ψ_1 in the definition of γ:

$$(\text{`}\mathsf{first}(\mathsf{first}(\tau_0\sigma)),\ \mathsf{first}(\sigma)) \models \varphi_1 \quad \wedge \quad (\text{`}\mathsf{first}(\sigma),\ \mathsf{first}(\mathsf{last}(\sigma\tau_1))) \models \psi_1.$$

   This implies that

$$(\text{`}\mathsf{first}(\mathsf{first}(\tau_0\sigma)),\ \mathsf{first}(\mathsf{last}(\sigma\tau_1))) \models (\varphi_1;\psi_1), \tag{5.24}$$

   because the $-;-$ operator for formulas models relational composition correctly. But $\mathsf{first}(\tau_0\sigma) = \mathsf{first}(\tau)$ and $\mathsf{last}(\tau) = \mathsf{last}(\sigma\tau_1)$. Thus, (5.23) follows from (5.24).

Next, we prove the second requirement given by $\varphi_2 \vee (\varphi_1; \psi_2)$:

$$\forall \sigma_0 \in \tau.\ \mathsf{iscall}(\sigma_0) \implies (\text{`}\mathsf{first}(\mathsf{first}(\tau)), \mathsf{first}(\sigma_0)) \models (\varphi_2 \vee (\varphi_1; \psi_2)). \tag{5.25}$$

Note that σ_0 appears in the prefix $\tau_0\sigma$ or in the suffix $\sigma\tau_1$. If the former holds,

$$(\text{`}\mathsf{first}(\mathsf{first}(\tau)), \mathsf{first}(\sigma_0)) = (\text{`}\mathsf{first}(\mathsf{first}(\tau_0\sigma)), \mathsf{first}(\sigma_0)) \models \varphi_2.$$

Hence, (5.25) holds. Now, suppose that σ_0 appears in the suffix $\sigma\tau_1$. Since $\sigma\tau_1$ satisfies the second requirement on ψ_2,

$$(\text{`}\mathsf{first}(\mathsf{first}(\sigma\tau_1)), \mathsf{first}(\sigma_0)) \models \varphi_2. \tag{5.26}$$

Furthermore, since $\tau_0\sigma$ satisfies the requirement from φ_1,

$$(\text{`}\mathsf{first}(\mathsf{first}(\tau_0\sigma)), \mathsf{first}(\mathsf{last}(\tau_0\sigma))) \models \varphi_1. \tag{5.27}$$

The desired (5.25) follows from (5.26) and (5.27), because the sequential composition $\varphi_1; \psi_2$ precisely means the relational composition.

Finally, we show the third requirement $\varphi_3 \vee \psi_3$. Pick σ_1 and σ_2 such that $\mathsf{open}(\sigma_1, \sigma_2, \tau)$. We should show that

$$(\text{`}\mathsf{first}(\sigma_1),\ \mathsf{first}(\sigma_2)) \models (\varphi_3 \vee \psi_3). \tag{5.28}$$

Note that $\tau = \tau_0\sigma\tau_1$ for $\tau_0\sigma$ in T_0 and $\sigma\tau_1$ in T_1. Furthermore, since $\tau_0\sigma$ is finite, if a state in $\tau_0\sigma$ is tagged with a procedure call, the corresponding return should appear in $\tau_0\sigma$ as well, because this is one of the conditions in the definition of the concrete semantic domain $\mathcal{D}$. Hence, either the states σ_1 and σ_2 tagged with procedure calls appear in $\tau_0\sigma$, or they both appear in $\sigma\tau_1$. In the first case,

$$(\text{`}\mathsf{first}(\sigma_1),\ \mathsf{first}(\sigma_2)) \models \varphi_3, \tag{5.29}$$

because $\tau_0\sigma$ satisfies the requirement regarding two call states. Similarly, in the second case, we have that

$$(\text{`}\mathsf{first}(\sigma_1),\ \mathsf{first}(\sigma_2)) \models \psi_3. \tag{5.30}$$

The desired (5.28) follows from (5.29) and (5.30).

2. The second case is that τ is an infinite trace from T_0. In this case, the first requirement given by $(\varphi_1; \psi_1)$ holds vacuously. The other two requirements also hold for a simple reason. Since the trace τ is in T_0, it satisfies the second and third requirements regarding φ_2 and φ_3. But, we have that

$$\varphi_2 \models \varphi_2 \vee (\varphi_1; \psi_2)$$

and

$$\varphi_3 \models \varphi_3 \vee \psi_3.$$

The second and third requirements are monotone with respect to formulas. Thus, τ also satisfies the two requirements given by weaker formulas $\varphi_2 \vee (\varphi_1; \psi_2)$ and $\varphi_3 \vee \psi_3$.

□

5.3.2.2 Abstract Join

We define the abstract join operator $\sqcup^\sharp$ as follows:

$$(\varphi_1,\ \varphi_2,\ \varphi_3)\ \sqcup^\sharp\ (\psi_1,\ \psi_2,\ \psi_3)\ \ =\ \ (\varphi_1 \vee \psi_1,\ \ \varphi_2 \vee \psi_2,\ \ \varphi_3 \vee \psi_3).$$

Lemma 5.33. *If both $(\varphi_1, \varphi_2, \varphi_3)$ and (ψ_1, ψ_2, ψ_3) are in $\mathcal{A}_t$, their join*

$$(\varphi_1, \varphi_2, \varphi_3)\ \sqcup^\sharp\ (\psi_1, \psi_2, \psi_3)$$

is also in $\mathcal{A}_t$.

Proof. Suppose that both $(\varphi_1, \varphi_2, \varphi_3)$ and (ψ_1, ψ_2, ψ_3) are in $\mathcal{A}_t$. This means that $\varphi_1, \psi_1 \in \mathsf{TLinForm}$. To prove this lemma, it suffices to show that $\varphi_1 \vee \psi_1 \in \mathsf{TLinForm}$. But,

$$\varphi_1 \models \varphi_1 \vee \psi_1.$$

Thus, from the totality of φ_1 (i.e., $\varphi_1 \in \mathsf{TLinForm}$), it follows that $\varphi_1 \vee \psi_1$ is total as well. □

Lemma 5.34. *For all $(\varphi_1, \varphi_2, \varphi_3)$ and (ψ_1, ψ_2, ψ_3) in $\mathcal{A}_t$, we have that*

$$\gamma(\varphi_1, \varphi_2, \varphi_3)\ \cup\ \gamma(\psi_1, \psi_2, \psi_3)\quad \subseteq\quad \gamma\big(\,(\varphi_1, \varphi_2, \varphi_3)\ \sqcup^\sharp\ (\psi_1, \psi_2, \psi_3)\,\big).$$

Proof. We will show that $\gamma(\varphi_1, \varphi_2, \varphi_3)$ is a subset of the RHS of the claimed inclusion in the lemma. The other case can be proved similarly. Pick a trace τ from $\gamma(\varphi_1, \varphi_2, \varphi_3)$. Then, for all $i \in \{1, 2, 3\}$, We have that

$$\varphi_i \;\models\; \varphi_i \vee \psi_i.$$

But, γ is monotone with respect to the implication order between formulas. Thus, from our assumption that $\tau \in \gamma(\varphi_1, \varphi_2, \varphi_3)$, the desired conclusion

$$\tau \;\in\; \gamma((\varphi_1, \varphi_2, \varphi_3) \;\sqcup^\sharp\; (\psi_1, \psi_2, \psi_3))$$

follows. □

5.3.2.3 Abstract Conditional Statement

The abstract conditional operator is defined by

$$\begin{aligned}
&\mathsf{if}^\sharp_{\mathsf{B}}((\varphi_1, \varphi_2, \varphi_3),\ (\psi_1, \psi_2, \psi_3)) \;=\\
&\quad \textbf{let } B_1 = \mathsf{preprime}(B) \text{ and } B_2 = \mathsf{preprime}(\mathsf{neg}(B))\\
&\quad \textbf{in } \big(\ (B_1 \wedge \varphi_1) \vee (B_2 \wedge \psi_1),\ \ (B_1 \wedge \varphi_2) \vee (B_2 \wedge \psi_2),\ \ \varphi_3 \vee \psi_3\ \big).
\end{aligned}$$

Here $\mathsf{preprime}(B)$ renames all the variables with the corresponding pre-primed variables, and $\mathsf{neg}(B)$ is the negation of B where $\neg$ is removed by being pushed all the way down to atomic predicates.

Lemma 5.35. *If both $(\varphi_1, \varphi_2, \varphi_3)$ and (ψ_1, ψ_2, ψ_3) are in $\mathcal{A}_t$, the conditional statement*

$$\mathsf{if}^\sharp_{\mathsf{B}}\big((\varphi_1, \varphi_2, \varphi_3),\ (\psi_1, \psi_2, \psi_3)\big)$$

is in $\mathcal{A}_t$ as well.

Proof. Suppose that both $(\varphi_1, \varphi_2, \varphi_3)$ and (ψ_1, ψ_2, ψ_3) are in $\mathcal{A}_t$. This means that $\varphi_1, \psi_1 \in \mathsf{TLinForm}$. To prove this lemma, it suffices to show that

$$(B_1 \wedge \varphi_1) \vee (B_2 \wedge \psi_1) \;\in\; \mathsf{LinForm}, \tag{5.31}$$

where B_1 and B_2 are defined as in the lemma. Note that B_2 is equivalent to $\neg B_1$. Thus, for all ‘s in ‘$\mathsf{Vars} \to \mathsf{Rationals}$, ‘$s$ satisfies B_1 or B_2. In the first case, the totality of φ_1 (i.e.,

$\varphi_1 \in \mathsf{TLinForm}$) implies that

$$\exists s \in (\mathsf{Vars} \to \mathsf{Rationals}).\ \ (`s, s) \models (B_1 \wedge \varphi_1) \vee (B_2 \wedge \psi_1).$$

In the second case, the same conclusion follows from the totality of ψ_1. Since $`s$ is chosen arbitrarily, we have just shown the required (5.31). □

Lemma 5.36. *For all* $(\varphi_1, \varphi_2, \varphi_3)$ *and* (ψ_1, ψ_2, ψ_3) *in* $\mathcal{A}_t$*, we have that*

$$\mathsf{if_B}\big(\gamma(\varphi_1, \varphi_2, \varphi_3),\ \gamma(\psi_1, \psi_2, \psi_3)\big) \ \subseteq\ \gamma\big(\mathsf{if}^\sharp_\mathsf{B}\big((\varphi_1, \varphi_2, \varphi_3),\ (\psi_1, \psi_2, \psi_3)\big)\big).$$

Proof. Pick $(\varphi_1, \varphi_2, \varphi_3)$ and (ψ_1, ψ_2, ψ_3) from $\mathcal{A}_t$. Let

$$A = (\varphi_1, \varphi_2, \varphi_3),\ \ A' = (\psi_1, \psi_2, \psi_3),\ \ B_1 = \mathsf{preprime}(B),\ \ B_2 = \mathsf{preprime}(\mathsf{neg}(B)).$$

We will show that every $\tau \in \mathsf{if_B}(\gamma(A), \gamma(A'))$ belongs to $\gamma(\mathsf{if}^\sharp_\mathsf{B}(A, A'))$. Choose an arbitrary τ from $\mathsf{if_B}(\gamma(A), \gamma(A'))$, and let σ be $\mathsf{first}(\tau)$. Then, $[\![B]\!](\mathsf{first}(\sigma)) = \mathsf{true}$, or $[\![B]\!](\mathsf{first}(\sigma)) = \mathsf{false}$. In the proof, we will consider the former case only, since the latter can be proved similarly. Suppose that $[\![B]\!](\mathsf{first}(\sigma)) = \mathsf{true}$. Then, for all $s \in (\mathsf{Vars} \to \mathsf{Rationals})$,

$$(`\mathsf{first}(\sigma), s) \ \models\ B_1. \tag{5.32}$$

Furthermore, by the definition of $\mathsf{if_B}$ and the assumption that $[\![B]\!](\mathsf{first}(\sigma)) = \mathsf{true}$, the trace τ should be in $\gamma(A)$, which means that

$$\begin{aligned}
&(\tau \in \mathsf{tagStates}^+ \implies (`\mathsf{first}(\mathsf{first}(\tau)), \mathsf{first}(\mathsf{last}(\tau))) \models \varphi_1) \quad \text{and} \\
&(\forall \sigma \in \tau.\, \mathsf{iscall}(\sigma) \implies (`\mathsf{first}(\mathsf{first}(\tau)), \mathsf{first}(\sigma)) \models \varphi_2) \quad \text{and} \\
&(\forall \sigma_1, \sigma_2.\, \mathsf{open}(\sigma_1, \sigma_2, \tau) \implies (`\mathsf{first}(\sigma_1), \mathsf{first}(\sigma_2)) \models \varphi_3).
\end{aligned} \tag{5.33}$$

From (5.32) and (5.33), the below property follows:

$$\begin{aligned}
&(\tau \in \mathsf{tagStates}^+ \implies (`\mathsf{first}(\mathsf{first}(\tau)), \mathsf{first}(\mathsf{last}(\tau))) \models (B_1 \wedge \varphi_1) \vee (B_2 \wedge \psi_1)) \quad \text{and} \\
&(\forall \sigma \in \tau.\, \mathsf{iscall}(\sigma) \implies (`\mathsf{first}(\mathsf{first}(\tau)), \mathsf{first}(\sigma)) \models (B_1 \wedge \varphi_2) \vee (B_2 \wedge \psi_2)) \quad \text{and} \\
&(\forall \sigma_1, \sigma_2.\, \mathsf{open}(\sigma_1, \sigma_2, \tau) \implies (`\mathsf{first}(\sigma_1), \mathsf{first}(\sigma_2)) \models \varphi_3 \vee \psi_3).
\end{aligned}$$

Thus, τ is in $\gamma(\mathsf{if}^\sharp_\mathsf{B}(A, A'))$, as desired. □

5.3.2.4 Function $\mathsf{procrun}^\sharp$ for Abstract Procedure Invocations

For a subset $X \subseteq \mathsf{Vars}$, let eq_X be the equality on the variables in X and the corresponding pre-primed ones:

$$eq_X \quad = \quad \bigwedge_{x \in X} (`x = x).$$

We define an abstract wrapper function for procedures as follows:

$$\mathsf{procrun}^\sharp_\mathsf{f}(\ \varphi_1,\ \varphi_2,\ \varphi_3\ ) \quad = \quad (\ \varphi_1,\ eq_{\mathsf{Vars}} \vee \varphi_2,\ \varphi_2 \vee \varphi_3\ ).$$

Lemma 5.37. *If $(\varphi_1, \varphi_2, \varphi_3)$ is in $\mathcal{A}_t$, so is $\mathsf{procrun}^\sharp_\mathsf{f}(\varphi_1, \varphi_2, \varphi_3)$.*

Proof. Suppose that $(\varphi_1, \varphi_2, \varphi_3)$ is in $\mathcal{A}_t$. This means that φ_1 is total, i.e., it belongs to $\mathsf{TLinForm}$. Since the first component of $\mathsf{procrun}^\sharp_\mathsf{f}(\varphi_1, \varphi_2, \varphi_3)$ is again φ_1, the totality of φ_1 implies that $\mathsf{procrun}^\sharp_\mathsf{f}(\varphi_1, \varphi_2, \varphi_3)$ is in $\mathcal{A}_t$, as desired. □

Lemma 5.38. *For all $f \in \mathsf{PNames}$ and all $(\varphi_1, \varphi_2, \varphi_3)$ in $\mathcal{A}_t$, we have that*

$$\mathsf{procrun}_f\big(\,\gamma(\varphi_1, \varphi_2, \varphi_3)\,\big) \ \subseteq\ \gamma\big(\,\mathsf{procrun}^\sharp_\mathsf{f}(\varphi_1, \varphi_2, \varphi_3)\,\big).$$

Proof. Pick $(\varphi_1, \varphi_2, \varphi_3) \in \mathcal{A}_t$. Consider a trace τ in $\mathsf{procrun}_f(\gamma(\varphi_1, \varphi_2, \varphi_3))$. We need to prove that

$$\tau \ \in\ \gamma\big(\,\mathsf{procrun}^\sharp_\mathsf{f}(\varphi_1, \varphi_2, \varphi_3)\,\big) \quad = \quad \gamma(\varphi_1,\ eq_{\mathsf{Vars}} \vee \varphi_2,\ \varphi_2 \vee \varphi_3).$$

Recall that $\mathsf{procrun}_f$ is defined to be the disjunction of three cases. In all three cases, there is some trace $\tau_1 \in \gamma(\varphi_1, \varphi_2, \varphi_3)$ such that

1. $\mathsf{first}(\tau_1) = \mathsf{first}(\tau)$, and
2. if τ is finite, so is τ_1 and $\mathsf{last}(\tau) = \mathsf{last}(\tau_1)$.

Furthermore, τ_1 satisfies the first requirement in the definition of $\gamma(\varphi_1, \varphi_2, \varphi_3)$, which is given by φ_1. Hence, τ also satisfies the first requirement of $\gamma(\mathsf{procrun}^\sharp_\mathsf{f}(\varphi_1, \varphi_2, \varphi_3))$, which is again given by φ_1.

In the rest of the proof, we prove the remaining two requirements for $\gamma(\mathsf{procrun}^\sharp_\mathsf{f}(\varphi_1, \varphi_2, \varphi_3))$. Let $T = \gamma(\varphi_1, \varphi_2, \varphi_3)$. Our proof will treat the three cases in the definition of $\mathsf{procrun}_f$ separately.

1. The first case is that $\tau = \sigma\sigma^{(f,call)}\sigma^{(f,ret)}\sigma$ for some $\sigma \in T$. In this case, the second requirement in the definition of γ is:

$$(\text{`}\mathsf{first}(\sigma), \mathsf{first}(\sigma^{(f,call)})) \models eq_{\mathsf{Vars}} \vee \varphi_2,$$

which holds because of eq_{Vars} on the RHS. For the third requirement, we note that there are no σ_1 and σ_2 in τ satisfying $\mathsf{open}(\sigma_1, \sigma_2, \tau)$. Hence, the requirement holds vacuously.

2. The second case is that $\tau = \sigma\sigma^{(f,call)}\tau_2\sigma_2^{(f,ret)}\sigma_2$ for some $\sigma\tau_2\sigma_2 \in (T \cap \mathsf{tagStates}^+)$. To prove the second requirement, consider $\sigma_3 \in \tau$ such that $\mathsf{iscall}(\sigma_3)$. If σ_3 is the second element in τ, it is $\sigma^{(f,call)}$, so

$$(\text{`}\mathsf{first}(\sigma), \mathsf{first}(\sigma_3)) \models eq_{\mathsf{Vars}}. \tag{5.34}$$

Otherwise, $\sigma_3 \in \sigma\tau_2\sigma_2$. Since $\sigma\tau_2\sigma_2$ is in $T = \gamma(\varphi_1, \varphi_2, \varphi_3)$, it satisfies the requirement given by φ_2. This implies that

$$(\text{`}\mathsf{first}(\sigma), \mathsf{first}(\sigma_3)) \models \varphi_2. \tag{5.35}$$

The satisfaction relationships (5.34) and (5.35) imply the desired property:

$$(\text{`}\mathsf{first}(\sigma), \mathsf{first}(\sigma_3)) \models eq_{\mathsf{Vars}} \vee \varphi_2.$$

For the third requirement, pick σ_3, σ_4 such that $\mathsf{open}(\sigma_3, \sigma_4, \tau)$. If σ_3 is not the second element of τ, we have that

$$\mathsf{open}(\sigma_3,\ \sigma_4,\ \sigma\tau_2\sigma_2).$$

Thus, $(\text{`}\mathsf{first}(\sigma_3), \mathsf{first}(\sigma_4)) \models \varphi_3$, and the desired third requirement follows from this. Otherwise, i.e., σ_3 is the second element of τ, it is $\sigma^{(f,call)}$. Thus, $\mathsf{first}(\sigma_3) = \mathsf{first}(\sigma)$. Since σ_4 is a call and it appears in $\sigma\tau_2\sigma_2 \in T$, it has to satisfy

$$(\text{`}\mathsf{first}(\sigma), \mathsf{first}(\sigma_4)) \models \varphi_2.$$

Now this satisfaction relationship and $\mathsf{first}(\sigma_3) = \mathsf{first}(\sigma)$ imply that the desired third requirement holds.

3. The last case is that $\tau \in \sigma\sigma^{(f,call)}\tau_2$ for $\sigma\tau_2 \in (T \cap \mathsf{tagStates}^\infty)$. The proof of this case

is almost identical to the one for the second. Simply replacing $\sigma\tau_2\sigma_2$ there by $\sigma\tau_2$ gives the proof of this third case.

□

5.3.2.5 Interpretation of Atomic Commands

The abstract semantics of atomic commands $x := E$ and $x := *$ is given as follows:

$$\begin{aligned} \mathsf{trans}^\sharp(x := E) &= (eq_{\mathsf{Vars}-\{x\}} \wedge (E[`Y/Y] = x),\ \mathsf{false},\ \mathsf{false}), \\ \mathsf{trans}^\sharp(x := *) &= (eq_{\mathsf{Vars}-\{x\}},\ \mathsf{false},\ \mathsf{false}), \end{aligned}$$

where Y is the set of all free variables in E. The first components of $\mathsf{trans}^\sharp(x := E)$ and $\mathsf{trans}^\sharp(x := *)$ express the relational meaning of both atomic commands, and the second and third components of them are false, because the atomic commands $x := E$ and $x := *$ do not involve procedure calls.

Lemma 5.39. *Both* $\mathsf{trans}^\sharp(x := E)$ *and* $\mathsf{trans}^\sharp(x := *)$ *are in* $\mathcal{A}_t$*. Furthermore, they satisfy the following soundness requirements:*

$$\mathsf{trans}(x := E) \subseteq \gamma(\mathsf{trans}^\sharp(x := E)), \qquad \mathsf{trans}(x{:=}*) \subseteq \gamma(\mathsf{trans}^\sharp(x{:=}*)).$$

Proof. The first claim of the lemma about the membership to $\mathcal{A}_t$ holds, because formulas $eq_{\mathsf{Vars}-\{x\}}$ and $eq_{\mathsf{Vars}-\{x\}} \wedge (x = E[`Y/Y])$ are in $\mathsf{TLinForm}$. Next, we show that

$$\mathsf{trans}(x := E) \subseteq \gamma(\mathsf{trans}^\sharp(x := E)).$$

The other inclusion can be proved similarly. Pick τ from $\mathsf{trans}(x := E)$. By the definition of $\mathsf{trans}(x := E)$, the trace τ should be of the form $\sigma_1\sigma_2$ such that

1. σ_1 and σ_2 are $none$-tagged states and
2. $\mathsf{first}(\sigma_2) = \mathsf{first}(\sigma_1)[x \mapsto [\![E]\!]\mathsf{first}(\sigma_1)]$.

This characterization of τ implies that τ does not include any function calls, and also that τ's starting and ending states σ_1 and σ_2 satisfy

$$(`\mathsf{first}(\sigma_1),\ \mathsf{first}(\sigma_2)) \models eq_{\mathsf{Vars}-\{x\}} \wedge (x = E[`Y/Y]).$$

From these, it follows that τ is in $\gamma(\mathsf{trans}^\sharp(x := E))$, as desired. □

5.3.2.6 Widening Operator

Before describing our widening operator, we note one simple fact on the requirement of our framework on that operator. The fact is described by the lemma below:

Lemma 5.40. *A binary operator $\triangledown : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$ is a widening operator, if it turns every sequence in $\mathcal{A}$ into one with a stable element and it satisfies the condition below:*

$$\big((\varphi_1, \varphi_2, \varphi_3) \triangledown (\psi_1, \psi_2, \psi_3) = (\delta_1, \delta_2, \delta_3)\big) \implies \forall i \in \{1, 2, 3\}.\ \big(\varphi_i \vee \psi_i \models \delta_i\big).$$

Proof. We need to prove two properties of $\triangledown$. Firstly, it can be restricted to a map from $\mathcal{A}_t \times \mathcal{A}_t$ to $\mathcal{A}_t$. Secondly, it computes an upper bound of its arguments:

$$\forall A, A' \in \mathcal{A}_t. \quad \gamma(A) \subseteq \gamma(A \triangledown A') \quad \text{and} \quad \gamma(A') \subseteq \gamma(A \triangledown A').$$

Pick $A = (\varphi_1, \varphi_2, \varphi_3)$ and $A' = (\psi_1, \psi_2, \psi_3)$ from $\mathcal{A}_t$. Let $(\delta_1, \delta_2, \delta_3)$ be $A \triangledown A'$. Since A, A' are in $\mathcal{A}_t$, their first components φ_1 and ψ_1 define total relations (i.e., $\varphi_1, \psi_1 \in \mathsf{TLinForm}$). Note that δ_1 is weaker than φ_1 and ψ_1 by assumption. Hence, δ_1 also defines a total relation, so $(\delta_1, \delta_2, \delta_3)$ is in $\mathcal{A}_t$, as desired by the first property. Now, we move on to the second property. For this, we show that $\gamma(A) \subseteq \gamma(A \triangledown A')$, since the other inclusion can be proved similarly. Pick a trace τ from $\gamma(A)$. By the definition of γ, we have that

$$\begin{aligned}
&(\tau \in \mathsf{tagStates}^+ \implies (\text{`}\mathsf{first}(\mathsf{first}(\tau)), \mathsf{first}(\mathsf{last}(\tau))) \models \varphi_1) \quad \text{and} \\
&(\forall \sigma \in \tau.\, \mathsf{iscall}(\sigma) \implies (\text{`}\mathsf{first}(\mathsf{first}(\tau)), \mathsf{first}(\sigma)) \models \varphi_2) \quad \text{and} \\
&(\forall \sigma_1, \sigma_2.\, \mathsf{open}(\sigma_1, \sigma_2, \tau) \implies (\text{`}\mathsf{first}(\sigma_1), \mathsf{first}(\sigma_2)) \models \varphi_3).
\end{aligned}$$

By assumption, $\varphi_i \models \psi_i$ for all $i \in \{1, 2, 3\}$. Thus, the three conjuncts above imply

$$\begin{aligned}
&(\tau \in \mathsf{tagStates}^+ \implies (\text{`}\mathsf{first}(\mathsf{first}(\tau)), \mathsf{first}(\mathsf{last}(\tau))) \models \delta_1) \quad \text{and} \\
&(\forall \sigma \in \tau.\, \mathsf{iscall}(\sigma) \implies (\text{`}\mathsf{first}(\mathsf{first}(\tau)), \mathsf{first}(\sigma)) \models \delta_2) \quad \text{and} \\
&(\forall \sigma_1, \sigma_2.\, \mathsf{open}(\sigma_1, \sigma_2, \tau) \implies (\text{`}\mathsf{first}(\sigma_1), \mathsf{first}(\sigma_2)) \models \delta_3).
\end{aligned}$$

That is, τ belongs to $\gamma(A \triangledown A')$, as desired. □

Our widening operator is parameterized by three elements. The first is a positive integer k, and it bounds the number of outermost disjuncts in formulas δ_i appearing in the results of widening. We will write $\triangledown_k$ to make this parameterization explicit. The second is a function lower that overapproximates a formula φ in LinForm by the conjunction of lower bounds on some pre-primed variables (i.e., the conjunction of constraints of the form $r \leq {}`x$ for some *pre-primed* variable ${}`x$ and rational number r):

$$\mathsf{lower}(\varphi) \quad = \quad (r_1 \leq {}`x_1 \ \wedge \ r_2 \leq {}`x_2 \ \wedge \ \ldots r_n \leq {}`x_n)$$

such that $\varphi \models \mathsf{lower}(\varphi)$. The third is the dual of the second function. It is a function upper that overapproximates a formula φ in LinForm by the conjunction of constraints of the form ${}`x \leq r$.

The widening operator uses three subroutines. The first is toDNF. It transforms a formula $\varphi \in \mathsf{LinForm}$ to a disjunctive normal form where all existential quantifications are placed right before each conjunct. For instance,

$$\begin{aligned}\mathsf{toDNF}\big(\ \exists x'.\ 10 < x' \ \wedge \ (x' < x \ \vee \ x < 2)\ \big) \quad = \\ (\exists x'.\ 10 < x' \ \wedge \ x' < x) \ \vee \ (\exists x'.\ 10 < x' \ \wedge \ x < 2).\end{aligned}$$

The second is the function bound_k for bounding the number of outermost disjuncts to k:

$$\begin{aligned}\mathsf{bound}_k \quad &: \quad \mathsf{LinForm} \to \mathsf{LinForm} \\ \mathsf{bound}_k(\varphi) \quad &= \quad \begin{cases} \varphi & \text{if there are at most } k \text{ outermost disjuncts in } \varphi \\ \mathsf{true} & \text{otherwise.} \end{cases}\end{aligned}$$

The third function is the ranking function synthesis engine RFS from Section 4.4. We remind the reader of two properties of RFS:

1. RFS takes a disjunction-free formula $\varphi \in \mathsf{LinForm}$ and returns the singleton set of a disjunction-free formula $\psi \in \mathsf{LinForm}$, or the empty set, or $\top$.
2. If $\mathsf{RFS}(\varphi) = \{\psi\}$, the formula ψ in the set is a ranking relation and it overapproximates the input formula φ.

Using these three parameters and three subroutines, we define the widening operator:

$$
\begin{array}{rcl}
(\varphi_1, \varphi_2, \varphi_3) \mathbin{\triangledown_k} (\psi_1, \psi_2, \psi_3) & = & \\
\quad \textbf{let} \;\; (\bigvee_{j \in J_i} \kappa^i_j) & = & \mathsf{toDNF}(\psi_i) \quad (\text{where } \kappa^i_j \text{ is disjunction-free}) \\
\qquad\qquad \chi^i_j & = & \bigwedge \{ \text{`}x = x \mid x \in \mathsf{Vars} \;\text{ and }\; \kappa^i_j \vdash \text{`}x = x \} \\
\qquad\qquad \xi^i_j & = & \textbf{if} \;\big(\; \mathsf{RFS}(\kappa^i_j) = \{\zeta^i_j\} \;\text{ for some } \zeta^i_j \;\big) \\
& & \quad \textbf{then} \;\big(\; \zeta^i_j \wedge \mathsf{lower}(\kappa^i_j) \wedge \mathsf{upper}(\kappa^i_j) \wedge \chi^i_j \;\big) \\
& & \quad \textbf{else} \;\big(\; \mathsf{lower}(\kappa^i_j) \wedge \mathsf{upper}(\kappa^i_j) \wedge \chi^i_j \;\big) \\
\qquad\qquad \delta_i & = & \mathsf{bound}_k(\varphi_i \vee \bigvee_{j \in J_i} \{\xi^i_j \mid \kappa^i_j \not\models \varphi_i\}) \\
\quad \textbf{in} \; (\delta_1, \delta_2, \delta_3). &&
\end{array}
$$

Lemma 5.41. *The operator $\triangledown_k : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$ is a widening operator.*

Proof. All the subroutines and parameters used in the definition of $\triangledown_k$ overapproximate their input formulas. From this and the definition of $\triangledown_k$ above, it follows that

$$\big((\varphi_1, \varphi_2, \varphi_3) \mathbin{\triangledown_k} (\psi_1, \psi_2, \psi_3) = (\delta_1, \delta_2, \delta_3)\big) \implies \forall i \in \{1,2,3\}.\, \big(\varphi_i \vee \psi_i \models \delta_i\big).$$

Thus, by Lemma 5.40, to prove this lemma, we just need to show that $\triangledown_k$ turns every sequence into one with a stable element. Note that the formula δ_i in the result of the widening is in the range of bound_k, so it cannot have more than k outermost disjuncts. Furthermore, δ_i in the result of the widening is true, or it has one more disjunct than φ_i, or it is the same as φ_i. These imply that for every sequence $\{A_n = (\delta^n_1, \delta^n_2, \delta^n_3)\}_{n \in \omega}$ in $\mathcal{A}$, if we construct the widened sequence $\{A'_n = ((\delta')^n_1, (\delta')^n_2, (\delta')^n_3)\}_{n \in \omega}$ by

$$A'_1 = A_1 \quad \text{and} \quad A'_{n+1} = A'_n \triangledown_k A_{n+1},$$

then for all $i \in \{1,2,3\}$, every disjunct in $(\delta')^n_i$ is included in $(\delta')^{n+1}_i$, unless $(\delta')^{n+1}_i$ is true. Thus, the sequence $\{(\delta')^n_i\}_{n \in \omega}$ goes over the bound k and remains true forever, or it hits a limit element before reaching the bound k. This implies that $\{A'_n\}_{n \in \omega}$ has a stable point. □

5.3.2.7 Abstract Liveness Predicate

Our abstract semantics uses the following predicate $\text{SATISFYLIV}^\sharp$ on $\mathcal{A}_t$ in order to check whether an analysis result implies the desired liveness property:

$$\begin{array}{lcl} \text{SATISFYLIV}^\sharp(\varphi_1, \varphi_2, \varphi_3) & = & \textbf{let } (\bigvee_{i \in I} \delta_i) = \mathsf{toDNF}(\varphi_3) \\ & & \textbf{in } \big(\textbf{if } (\mathsf{RFS}(\delta_i) \neq \top \text{ for all } i \in I) \textbf{ then } \mathsf{true} \textbf{ else } \mathsf{false}\big). \end{array}$$

The predicate $\text{SATISFYLIV}^\sharp$ first transforms φ_3 to a disjunctive normal form. Then, it checks whether each disjunct δ_i is well-founded using the function RFS from Section 4.4. Hence, if the predicate returns true, it means that φ_3 is disjunctively well-founded.

Lemma 5.42. *For all $A \in \mathcal{A}_t$, if $\text{SATISFYLIV}^\sharp(A) = \mathsf{true}$, we have that $\gamma(A) \in \text{LIVPROPERTY}$.*

Proof. Consider $A \in \mathcal{A}_t$ such that $\text{SATISFYLIV}^\sharp(A) = \mathsf{true}$. Pick a trace $\tau \in \gamma(A)$. For the sake of contradiction, suppose that τ includes an infinite subsequence of open calls. That is, there exists $\{\tau_i \sigma_i\}_{i \in \omega}$ such that

$$(\tau \;=\; \tau_1\sigma_1\tau_2\sigma_2 \ldots) \quad \wedge \quad (\forall i, j \in \omega.\, i < j \implies \mathsf{open}(\sigma_i, \sigma_j, \tau)).$$

By the definition of γ, we should have that

$$\forall i \in \omega.\, (\text{`}\mathsf{first}(\sigma_i), \mathsf{first}(\sigma_j)) \models \varphi_3.$$

Furthermore, the formula φ_3 is disjunctively well-founded, since $\text{SATISFYLIV}^\sharp(A) = \mathsf{true}$. Hence, the result of Podelski and Rybalchenko (Lemma 4.5) implies that the sequence $\sigma_1\sigma_2 \ldots$ is finite. But, this is impossible, since $\sigma_1\sigma_2 \ldots$ is an infinite sequence. We have just derived the desired contradiction. □

5.3.2.8 An Example

We illustrate our abstract interpreter with the command below:

$$C \;\equiv\; \texttt{fix}\ f.\, \Big(\ \texttt{if}\ \ (x \leq 0)\ \ (x{:=}x)\ \ (x{:=}x{-}1;\ f();\ x{:=}x{+}1)\ \Big)$$

Note that this command is not tail recursive, but it always terminates.

To simplify presentation, we will assume that x is the only program variable. We also assume that lower returns lower bounds of the form $`x \geq 0$ only. Similarly, we assume that upper computes upper bounds of the form $`x \leq 0$ only.

Our abstract interpreter calculates the abstract semantics of C by an iterative fixpoint computation. The first iteration of this computation works as follows. It picks the environment η_0 defined by:

$$A_0 \quad = \quad (\mathsf{false}, \mathsf{false}, \mathsf{false}), \qquad\qquad \eta_0 \quad = \quad [f \mapsto A_0].$$

Then, the abstract interpreter analyzes the true and false branches of the conditional statement in f:

$$\begin{aligned} [\![x{:=}x]\!]^\sharp \eta_0 \quad &= \quad (`x{=}x,\ \mathsf{false},\ \mathsf{false}), \\ [\![x{:=}x{-}1; f(); x{:=}x{+}1]\!]^\sharp \eta_0 \quad &= \quad (\mathsf{false},\ \ \mathsf{false},\ \ \mathsf{false}). \end{aligned}$$

Finally, it computes the abstract meaning of the body of f:

$$\begin{aligned} A_1 \quad &= \quad A_0 \triangledown \left(\mathsf{procrun}^\sharp_{\mathsf{f}}([\![\texttt{if}\ (x \leq 0)\ (x{:=}x)\ (x{:=}x{-}1; f(); x{:=}x{+}1)]\!]^\sharp \eta_0)\right) \\ &= \quad (\mathsf{false}, \mathsf{false}, \mathsf{false}) \triangledown \mathsf{procrun}^\sharp_{\mathsf{f}}(`x \leq 0 \wedge `x{=}x,\ \ \mathsf{false},\ \ \mathsf{false}) \\ &= \quad (\mathsf{false}, \mathsf{false}, \mathsf{false}) \triangledown (`x \leq 0 \wedge `x{=}x,\ \ `x{=}x,\ \ \mathsf{false}) \\ &= \quad (`x \leq 0 \wedge `x{=}x,\ \ `x{=}x,\ \ \mathsf{false}). \end{aligned}$$

The second fixpoint iteration proceeds similarly. It picks the environment η_1:

$$\eta_1 \quad = \quad [f \mapsto A_1].$$

Then, it computes the abstract semantics of the true and false branches:

$$\begin{aligned} [\![x{:=}x]\!]^\sharp \eta_1 \quad &= \quad (`x{=}x,\ \mathsf{false},\ \mathsf{false}), \\ [\![x{:=}x{-}1; f(); x{:=}x{+}1]\!]^\sharp \eta_1 \quad &= \quad \big(\ (\exists a'b'.\ `x{-}1{=}a' \wedge a' \leq 0 \wedge a'{=}b' \wedge b'{+}1{=}x), \\ &\qquad\quad (\exists a'.\ `x{-}1{=}a' \wedge a'{=}x), \\ &\qquad\quad \mathsf{false}\ \big). \end{aligned}$$

Finally, the abstract interpreter combines the above two abstract values, and finishes the second iteration:

$$
\begin{aligned}
A_2 &= A_1 \triangledown \mathsf{procrun}^\sharp_{\mathsf{f}}([\![\mathtt{if}\ (x \le 0)\ (x{:=}x)\ (x{:=}x{-}1; f(); x{:=}x{+}1)]\!]^\sharp \eta_1) \\
&= A_1 \triangledown \big((\text{`}x \le 0 \wedge \text{`}x{=}x) \vee (\text{`}x > 0 \wedge \exists a'b'.\, \text{`}x{-}1{=}a' \wedge a' \le 0 \wedge a'{=}b' \wedge b'{+}1{=}x), \\
&\qquad\qquad (\text{`}x{=}x) \vee (\text{`}x > 0 \wedge \exists a'.\, \text{`}x{-}1{=}a' \wedge a'{=}x), \\
&\qquad\qquad (\text{`}x > 0 \wedge \exists a'.\, \text{`}x{-}1{=}a' \wedge a'{=}x) \big) \\
&= \big((\text{`}x \le 0 \wedge \text{`}x{=}x) \vee (\text{`}x \ge 0 \wedge \text{`}x{=}x), \\
&\quad (\text{`}x{=}x) \vee (\text{`}x \ge 0 \wedge \text{`}x{-}1 \ge x), \\
&\quad (\text{`}x \ge 0 \wedge \text{`}x{-}1 \ge x) \big).
\end{aligned}
$$

The computed A_2 is the fixpoint, and becomes the result of analyzing the command C.

After the fixpoint computation, the abstract interpreter checks whether SATISFYLIV$^\sharp$ holds for A_2. In this case, we have that SATISFYLIV$^\sharp(A_2) = \mathsf{true}$, because the third component of A_2 is a well-founded relation. Hence, by Lemma 5.42, the concrete meaning of C belongs to LIVPROPERTY, i.e., $[\![C]\!]$ does not contain an infinite subsequence of open calls. In still other words, C terminates.

5.4 Conclusion

In this chapter, we have presented a framework for designing a sound abstract interpreter for liveness properties. The key feature of the framework is the use of metric space in the concrete semantics of programs. By using the metric-space semantics, our framework allows us to overcome the main challenge in showing the soundness of an abstract interpreter for liveness, namely to relate the meaning of recursion in the concrete semantics with that in the abstract semantics, even when procedures in a program are recursive over the unit type.

In this chapter, we also described an instance of the framework, where the abstract semantics uses ranking functions and the result of Podelski and Rybalchenko, as in the previous chapter. However, the abstract semantics in this chapter went beyond abstract interpreters in the previous one, in that it can prove the termination of programs with recursion over the unit type.

Chapter 6

Conclusion and Related Work

6.1 Synopsis

In this thesis we have defined a number of abstract interpreters to prove program termination. All of these have made use of the Podelski-Rybalchenko result, which provides a proof rule to prove the well-foundedness of a relation which is suited to automatic program verification.

In Chapter 2 we gave a brief outline of a algorithm to convert abstract interpreters for safety properties into a termination analyser. The approach was very promising but had its limitations. The algorithm reused operators from the safety abstract domain. These operators were not designed with proving termination, a liveness property. As a result the termination analysers produced were fast but limited in comparison to existing approaches.

To address this issue, in Chapter 4 we turned our attention to designing an abstract domain specifically for termination. We presented a framework for defining such abstract domains, and presented an instance of this framework based on linear ranking functions. The abstract domain only keeps information relevant to termination by using a rank function synthesis engine. In essence the abstract domain is built using ranking functions. Since termination is a path sensitive property, the abstract domain defined is disjunctive. The instance was implemented and matched the termination provers we produced in the first chapter and was more precise. But one shortcoming of this approach is that it can only analyse programs with iteration. This problem was due to the fact the soundness proof had to relate greatest fixpoints in the concrete semantics with least-fixed points in the abstract semantics. This limitation was a symptom of the proof technique relating the fixpoints.

To get a cleaner soundness result we turned our attention to metric spaces in Chapter 5. The theory of metric spaces has a well-known theorem: Banach's Fixpoint Theorem which states that

a contractive function on a complete metric space has a unique fixpoint. We built a framework around this result which allowed us to relate unique fixpoints in the concrete semantics with the pre-fixpoints in the abstract semantics. However the use of metric spaces means that we have to take care to ensure that we are working in a complete metric space allowing us to use Banach's Fixpoint Theorem. Like Chapter 4 the concrete semantics was based on sets of traces, but since the powerset of traces is not a complete metric space we had to have certain restrictions to ensure our concrete semantics was a complete metric space. This in turn affected the design of the abstract semantics: we had to ensure that the concretization of abstract elements produced non-empty closed sets of traces. The framework built in Chapter 5 defines a general framework for designing abstract interpreters for liveness properties. We showed one instance, again based on linear ranking relations. However we have not implemented this instance and this is left as future work.

6.2 Related Work

In this thesis we have focused on proving termination for imperative programs. There has been much work on proving termination in the functional [41], logic [17, 30, 46] and term rewriting [18, 32] community.

The first work to utilise the Podelski-Rybalchenko result was TERMINATOR [19, 20]. TERMINATOR uses abstraction refinement in order to construct a finite collection of well-founded relations. TERMINATOR transforms the program being analysed to add an error location which corresponds to a case in which the program does not terminate. Now the search for a termination argument has been reduced to a reachability question: can this error location be reached from some initial state? TERMINATOR makes use of SLAM[5, 6] in order to find some path from the initial state to the error location. If such a path exists and is not spurious, TERMINATOR checks to see if this path is well-founded using a rank function synthesis engine to construct well-founded relations and then tries to prove that these relations overapproximate the relational meaning of the program, i.e., checking the subset inclusion holds in the Podelski-Rybalchenko result. This subset inclusion check is the main bottleneck in TERMINATOR. By directly designing abstract domains for termination we are avoiding the need to check the subset inclusion since we know the analysis overapproximates the program's concrete semantics.

The abstract domain in Chapter 4 is related to the abstraction used in size-change termination [41]. In both approaches program fragments are abstracted in terms of measures decreased or preserved by the fragments. The major difference is that our domain contains only abstract elements relevant for termination (unless the elements are $\top$ in the case we can't prove termination), whilst size-change termination analyses can have an (non-$\top$) abstract element that denotes a diverging program. As a result, size-change termination analyses have to check whether the

concretization of an abstracted program terminates, whereas our analysis can skip this rather expensive check.

Bradley et al.[10, 12] define a method to perform rank function synthesis over polynomials. They use a technique based on finite difference equations over transition systems to produce a termination argument. The result is complementary to the work in this thesis: we could use this in place of RANKFINDER in the analysis presented in this thesis.

In [3, 58] CEGAR for safety properties is combined with ranking abstractions for liveness properties. Ranking functions are represented as progress monitors, which are syntactically added to a program. They make use of the CEGAR loop to construct a set of ranking functions which overapproximate the program. However the ranking function synthesis heuristics used in their paper is not for linear expressions. In comparison our use of abstract domains built with ranking functions is much more efficient than the SCAR approach used in this work.

The use of metric spaces in programming language semantics has been extensively studied [13, 29, 57]. The book [29] defines metric space semantics for a number of languages. We have used many of the results from [29] in defining the concrete instance in Chapter 5, but have had to make additions to the semantics in order to produce a useful analyser, such as distinguishing normal and function call/return states.

The relationship between metric spaces and domain theory has also been studied extensively [2]. Previous work is orthogonal to the work we have done in this thesis. We have shown conditions for when an order-theoretic semantics overapproximates a metric space semantics. In [2] the authors show how a metric semantics can be derived from a partial order semantics.

In [21] the authors define a method for proving the termination of recursive procedures. The technique works by transforming the program to remove the recursive calls. These are replaced by a non-deterministic choice between entering a procedure body (in the case that the procedure does not terminate) or the application of a summary of the procedure (in the case that the procedure does terminate). The work we have done in Chapter 5 could be seen as a step towards a semantic understanding of the technique in [21]. The exact relationship between the two is left as future work.

The work by Podelski et al. [51] defines a framework and a proof system for showing liveness properties for while programs with recursion. The work could be used as a basis for abstract interpreters for liveness properties. The exact relationship between this work and ours is left as future work.

Cousot [24] defines an abstract interpretation framework for overapproximating both least and greatest fixpoints. Using a given abstract domain they lift this to another abstract domain, whose

least fixpoint contains the least and greatest fixpoints of functions in the original domain. Essentially the authors are using both induction and co-induction to produce an abstract domain which accounts for both finite and infinite behaviours.

Recently there has been progress in automatically proving a complexity bound on programs [34, 35]. Here the authors define a program analysis to find bounds on the number of iterations a loop can take. To calculate the bound, the authors augment the program with counters and attempt to extract symbolic bounds using numerical abstract domains. A bound implies program termination, so they are in fact going further than termination analysis. However in order to get useful results the bound analysis very often requires an initial safety analysis which computes global invariants which are then used in the computation of the bounds.

6.3 Conclusions

In this thesis we have defined novel techniques for automatically proving program termination. We have defined two frameworks for designing abstract interpreters to prove program termination. By designing abstract domains specifically for termination we have addressed problems with previous approaches based on abstraction refinement and lifting safety analysers to termination analysers. We believe that future progress in practical termination analyses will employ specially designed abstract domains for termination (and liveness), as advanced in this thesis. The ideas in this thesis could be used with other techniques, particularly counter example guided abstraction refinement.

Appendix A

Basic Notions of Metric Spaces

The material below is standard and taken from [29, 57].

Definition A.1 (Metric Space). A metric space consists of a pair (X, d_X) such that

1. X is a non-empty set, and
2. d_X is a function of type $X \times X \to [0, \infty]$, called metric, and it satisfies the following three conditions:
   - $\forall x, y \in X.\ \ d_X(x, y) = 0 \iff x = y.$
   - $\forall x, y \in X.\ \ d_X(x, y) = d_X(y, x).$
   - $\forall x, y, z \in X.\ \ d_X(x, z) \leq d_X(x, y) + d_X(y, z).$

Definition A.2 (Convergence). For a metric space (X, d_X):

1. A sequence $(x_n)_n$ in X is convergent if and only if

$$\forall \epsilon > 0.\ \exists N \in \mathbb{N}.\ \forall n \geq N.\ d_X(x_n, x) \leq \epsilon$$

   for some $x \in X$.

2. A sequence $(x_n)_n$ in X is Cauchy if

$$\forall \epsilon > 0.\ \exists N \in \mathbb{N}.\ \forall m, n \geq N.\ d_X(x_m, x_n) \leq \epsilon.$$

Proposition A.3. *Every convergent sequence is Cauchy.*

Definition A.4 (**Completeness**). A metric space is complete if and only if every Cauchy sequence in the metric space is convergent.

Definition A.5 (**Non-expansiveness and α-Contractiveness**). Let X and Y be metric spaces. Let $f : X \to Y$ be a function.

1. The function f is continuous if and only if for every $(x_n)_n$ converging to x we have the sequence $(f(x_n))_n$ converges to $f(x)$.
2. Let $\alpha \geq 0$. The function f is α-Lipschitz if and only if for all $x_1, x_2 \in X$,

$$d_Y(f(x_1), f(x_2)) \leq \alpha \times d_X(x_1, x_2).$$

3. A function is non-expansive if and only if it is 1-Lipschitz.
4. A function is α-contractive if and only if it is α-Lipschitz and $0 \leq \alpha < 1$.

Theorem A.6 (**Banach's Fixed Point Theorem**). *Let X be a metric space and let $f : X \to X$ be a α-contractive function for some $0 \leq \alpha < 1$.*

1. *If $f(x) = x$ and $f(y) = y$, then $x = y$*
2. *Suppose that X is complete. If $\{x_n\}_{n\in\omega}$ is a sequence defined by $x_{n+1} = f(x_n)$, the sequence is Cauchy and its limit x is the fixpoint of f, that is, $f(x) = x$.*

We denote the unique fixpoint of f by $\mathsf{ufix}(f)$.

In this thesis, we work with traces and sets of traces. Thus, we need suitable metrics for them. Suppose we have a set A. Let $A^{*,\infty}$ be the set of all nonempty finite or infinite sequences constructed from A.

Definition A.7 (**Baire Metric**). The Baire-metric $d_B : A^{*,\infty} \times A^{*,\infty} \to [0, \infty]$ is given by:

$$d_B(v, w) = \begin{cases} 0 & \text{if } v = w \\ 2^{-\max\{k \mid v[k]=w[k]\}} & \text{if } v \neq w \end{cases}$$

Here the notation $x[n]$ means the n-th prefix of trace x. A useful lemma linking the Baire metric and properties of traces is next.

Lemma A.8. $d_B(x, y) \leq 2^{-n} \iff x[n] = y[n]$.

Metric on Subsets of a Metric Space

We will now recall how to lift a metric from a set to its powerset. The Hausdorff metric is the standard way to do such lifting.

Definition A.9 (Hausdorff Metric). Let (M, d) be a metric space. The Hausdorff metric is defined by:

$$d_{\mathcal{P}(M)}(X, Y) = \inf\{\alpha > 0 \mid (\forall x \in X.\ \exists y \in Y.\ d(x, y) < \alpha) \ \wedge \ (\forall y \in Y.\ \exists x \in X.\ d(x, y) < \alpha)\}.$$

Definition A.10 (Closed Sets). Let (M, d) be a metric space. A subset X of M is closed if and only if each convergent sequence $(x_n)_n$ with $x_n \in X, n = 0, 1, ...$ has its limit x in X.

Theorem A.11. *If (M, d) is a metric space, $(\mathcal{P}_{cl}(M), D_H)$ is a metric space, where $\mathcal{P}_{cl}(M)$ is the collection of all closed subsets of M.*

The theorem below establishes that the closed powerdomain operator preserves completeness.

Theorem A.12 (Hahn). *Let (M, d) be a complete metric space. Then $(\mathcal{P}_{cl}(M), d_H)$ is a complete metric space.*

The definition below is more useful and relevant for our needs than the definition of d_H above.

Definition A.13 (Lifted Baire Metric). Consider the metric space $(A^{*,\infty}, d_B)$ with nonempty finite or infinite traces. Let $X, Y \subseteq A^{*,\infty}$. The lifted Baire metric d_B^+ is defined as follows:

$$d_B^+(X, Y) = \begin{cases} 0 & \text{if } X = Y \\ 2^{-\max\{k \mid X[k]=Y[k]\}} & \text{otherwise} \end{cases}$$

where $X[n]$ denotes $\{\ \tau[n] \mid \tau \in X\ \}$.

The following lemma proves that the above definition corresponds with the general definition:

Lemma A.14. *For each X,$Y \in \mathcal{P}_{cl}(A^{*,\infty})$, we have that $d_H(X, Y) = d_B^+(X, Y)$.*

A.1 Farkas Lemma

In this section we will mention Farkas Lemma [53], which is used later in this thesis. Farkas Lemma gives a sound and complete method for reasoning about systems of linear inequalities.

We use it in this thesis in order to find linear ranking functions. Note that Farkas Lemma is usually presented over real numbers. However the proof is equally valid over the rationals: Farkas Lemma does not depend on any properties which hold for the reals but not for rational numbers [28, 55].

Theorem A.15 (Farkas Lemma)**.** *Consider the following system of linear inequalities over real variables* $V = \{x_1, ..., x_m\}$

$$S : \begin{bmatrix} A_{1,1}x_1 & + & \dots & + & A_{1,m}x_m & + & A_{1,m+1} \geq 0 \\ \vdots & & & & \vdots & & \vdots \\ A_{n,1}x_1 & + & \dots & + & A_{n,m}x_m & + & A_{n,m+1} \geq 0 \end{bmatrix}$$

If S *is satisfiable, it entails a linear inequality* $c_1x_1 + ... + c_mx_m + c_{m+1} \geq 0$ *iff there exist real numbers* $\lambda_1, ..., \lambda_n \geq 0$ *such that:*

$$c_1 = \Sigma_{i=1}^{n} \lambda_i.A_{i,1} \ ... \ c_m = \Sigma_{i=1}^{n} \lambda_i.A_{i,m} \ \ c_{m+1} \geq \Sigma_{i=1}^{n} \lambda_i.A_{i,m+1}$$

Furthermore, S *is unsatisfiable iff* S *entails* $-1 \geq 0$*.*

A.2 Equations to Support Adequacy of Metric Semantics

In this section we will show some example programs which give evidence that loop unrolling is sound.

Example A.1. *Let* $T = [\![\Gamma \vdash \texttt{fix}\ f.x := 1; f()]\!]\eta$

$$\begin{array}{lcl} [\![\Gamma \vdash \texttt{fix}\ f.x := 1; f()]\!]\eta & = & \mathsf{ufix}\lambda k.\mathsf{procrun}_f([\![x := 1; f()]\!]\eta[f \mapsto k] \\ & = & \mathsf{ufix}\lambda k.\mathsf{procrun}_F(\mathsf{seq}([\![x := 1]\!]\eta[f \mapsto k], [\![f()]\!]\eta[f \mapsto k])) \\ & = & \mathsf{ufix}\lambda k.\mathsf{procrun}(\mathsf{seq}([\![x := 1]\!]\eta[f \mapsto k], k)) \\ \textit{Since } T \textit{ is a fixpoint} & = & \mathsf{procrun}_f(\mathsf{seq}([\![x := 1]\!]\eta[f \mapsto k], T) \\ \textit{By definition of } T & = & \mathsf{procrun}_f(\mathsf{seq}([\![x := 1]\!]\eta, [\![fix f.x := 1; f()]\!]\eta)) \\ \textit{By definition of ;} & = & \mathsf{procrun}_f([\![x := 1; fix f.x := 1; f()]\!]\eta) \\ & = & \mathsf{procrun}_f((x := 1; f())[fix f.x := 1; f()/f]) \end{array}$$

where $C[\texttt{fix}\ f.C/f]$ *denotes the substition of* $\texttt{fix}\ f.C$ *for* f *in* C*.*

Example A.2. *Let* $T = [\![\Gamma \vdash \texttt{fix}\ f.f(); x := 1; f()]\!]\eta$

$$
\begin{array}{lcl}
[\![fix f.f(); x = 1; f()]\!]\eta & = & \mathsf{ufix}\ \lambda\, k.(\mathsf{procrun}_f([\![f(); x := 1; f()]\!])\eta[f \mapsto k] \\
\textit{By definition of ;} & = & \mathsf{ufix}\ \lambda\, k.(\mathsf{procrun}_f(\mathsf{seq}([\![f()]\!]\eta[f \mapsto k], [\![x := 1; f()]\!]\eta[f \mapsto k]))) \\
\textit{By definition of ;} & = & \mathsf{ufix}\ \lambda\, k.(\mathsf{procrun}_f(\mathsf{seq}([\![f()]\!]\eta[f \mapsto k], \mathsf{seq}([\![x := 1]\!]\eta[f \mapsto k], [\![f()]\!]\eta[f \mapsto k])))) \\
\textit{By definition of } f & = & \mathsf{ufix}\ \lambda\, k.(\mathsf{procrun}_f(\mathsf{seq}(k, \mathsf{seq}([\![x := 1]\!]\eta[f \mapsto k], k)))) \\
\textit{Since } T \textit{ is a fixpoint} & = & \mathsf{procrun}_f(\mathsf{seq}(T, \mathsf{seq}([\![x := 1]\!]\eta[f \mapsto k], T))) \\
\textit{By definition of seq and ;} & = & \mathsf{procrun}_f(\mathsf{seq}([\![\texttt{fix}\ f.f(); x := 1; f()]\!]\eta, \mathsf{seq}([\![x := 1]\!]\eta, [\![\texttt{fix}\ f.f(); x := 1; f()]\!]\eta))) \\
 & = & \mathsf{procrun}_f([\![(f(); x := 1; f())[(\texttt{fix}\ f.x := 1; f())/f]]\!]\eta
\end{array}
$$

Example A.3.

$$
\begin{array}{lcl}
[\![\texttt{fix}\ f.x := 1]\!]\eta & = & \mathsf{ufix}\lambda\ k.(\mathsf{procrun}_f([\![x := e]\!]\eta[f \mapsto k])) \\
\textit{By definition of assign} & = & (\mathsf{procrun}_f(\{ss' \mid s' = s[x \mapsto [\![e]\!]]\})) \\
& = & \mathsf{procrun}_f([\![x := e[(\texttt{fix}\ f.x := 1)/f]]\!]\eta
\end{array}
$$

Bibliography

[1] ALPERN, B., ALPERA, B., SCHNEIDER, F. B., AND SCHNEIDER, F. B. Recognizing safety and liveness. *Distributed Computing 2* (1986), 117–126.

[2] BAIER, C., AND MAJSTER-CEDERBAUM, M. E. Metric semantics from partial order semantics. *Acta Inf. 34*, 9 (1997), 701–735.

[3] BALABAN, I., PNUELI, A., AND ZUCK, L. Ranking abstraction as companion to predicate abstraction. In *FORTE'05* (2005).

[4] BALL, T., COOK, B., DAS, S., AND RAJAMANI, S. K. Refining approximations in software predicate abstraction. In *TACAS'04: Tools and Algorithms for Construction and Analysis of Systems* (2004), vol. 2988 of *LNCS*.

[5] BALL, T., COOK, B., LEVIN, V., AND RAJAMANI, S. K. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In *IFM'04: Fourth International Conference on Integrated Formal Methods* (2004), vol. 2999 of *LNCS*.

[6] BALL, T., MAJUMDAR, R., MILLSTEIN, T., AND RAJAMANI, S. Automatic predicate abstraction of C programs. In *PLDI'01: Programming Language Design and Implementation* (2001), vol. 36 of *ACM SIGPLAN Notices*.

[7] BERDINE, J., CHAWDHARY, A., COOK, B., DISTEFANO, D., AND O'HEARN, P. Variance analyses from invariance analyses. In *POPL'07* (2007).

[8] BLANCHET, B., COUSOT, P., COUSOT, R., FERET, J., MAUBORGNE, L., MINÉ, A., MONNIAUX, D., AND RIVAL, X. A static analyzer for large safety-critical software. In *PLDI'03: Programming Language Design and Implementation* (2003).

[9] BRADLEY, A. Personal communication. Aaron Bradley's suggested script that iteratively applies the tools described in [12] and [10] with increasingly expensive options, June 2006.

[10] BRADLEY, A., MANNA, Z., AND SIPMA, H. Termination of polynomial programs. In *VMCAI'05: Verification, Model Checking, and Abstract Interpretation* (2005).

[11] BRADLEY, A. R., MANNA, Z., AND SIPMA, H. B. Linear ranking with reachability. In *CAV* (2005), pp. 491–504.

[12] BRADLEY, A. R., MANNA, Z., AND SIPMA, H. B. The polyranking principle. In *ICALP'05: International Colloquium on Automata, Languages and Programming* (2005).

[13] BREUGEL, F. V. *Comparative Metric Semantics of Programming*. Birkhauser Boston, 1997.

[14] CHAWDHARY, A., COOK, B., GULWANI, S., SAGIV, M., AND YANG, H. Ranking abstractions. In *ESOP* (2008), pp. 148–162.

[15] CLARKE, E., GRUMBERG, O., JHA, S., LU, Y., AND VEITH, H. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM 50*, 5 (September 2003), 752–794.

[16] CLARKE, E., GRUMBERG, O., AND PELED, D. *Model checking*. MIT Press, December 1999.

[17] CODISH, M., AND TABOCH, C. A semantic basis for the termination analysis of logic programs. *The Journal of Logic Programming 41*, 1 (1999).

[18] CONTEJEAN, E., MARCHÉ, C., MONATE, B., AND URBAIN, X. Proving Termination of Rewriting with C*i*ME. In *Extended Abstracts of the 6th International Workshop on Termination, WST'03* (June 2003).

[19] COOK, B., PODELSKI, A., AND RYBALCHENKO, A. Termination proofs for systems code. In *PLDI'06: Programming Language Design and Implementation* (2006).

[20] COOK, B., PODELSKI, A., AND RYBALCHENKO, A. Terminator: Beyond safety. In *CAV'06: International Conference on Computer Aided Verification* (2006).

[21] COOK, B., PODELSKI, A., AND RYBALCHENKO, A. Summarization for termination: no return! *Formal Methods in System Design 35*, 3 (2009), 369–387.

[22] COUSOT, P. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoretical Comput. Sci. 277*, 1–2 (2002), 47–103.

[23] COUSOT, P., AND COUSOT, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77: Principles of Programming Languages* (1977).

[24] COUSOT, P., AND COUSOT, R. Bi-inductive structural semantics. *Information and Computation 207*, 2 (2009), 258–283.

[25] COUSOT, P., COUSOT, R., FERET, J., MAUBORGNE, L., MINÉ, A., MONNIAUX, D., AND RIVAL, X. The ASTRÉE analyzer. In *ESOP'05: European Symposium on Programming* (2005).

[26] COUSOT, P., AND HALBWACHS, N. Automatic discovery of linear restraints among variables of a program. In *POPL'78: Principles of Programming Languages* (1978).

[27] DAMS, D., GERTH, R., AND GRUMBERG, O. A heuristic for the automatic generation of ranking functions. In *Workshop on Advances in Verification* (2000), pp. 1–8.

[28] DAX, A. Classroom note: An elementary proof of farkas' lemma. *SIAM Rev. 39*, 3 (1997), 503–507.

[29] DE BAKKER, J., AND DE VINK, E. *Control flow semantics*. MIT Press, Cambridge, MA, USA, 1996.

[30] DERSHOWITZ, N., LINDENSTRAUSS, N., SAGIV, Y., AND SEREBRENIK, A. A general framework for automatic termination analysis of logic programs. *Appl. Algebra Eng. Commun. Comput. 12*, 1/2 (2001).

[31] DETLEFS, D., NELSON, G., AND SAXE, J. Simplify: A theorem prover for program checking, 2003.

[32] GIESL, J., THIEMANN, R., SCHNEIDER-KAMP, P., AND FALKE, S. Automated termination proofs with AProVE. In *RTA'04: Rewriting Techniques and Applications* (2004).

[33] GULAVANII, B. S., AND RAJAMANI, S. K. Counterexample driven refinement for abstract interpretation. In *TACAS'06: Tools and Algorithms for the Construction and Analysis of Systems* (2006).

[34] GULWANI, S. Speed: Symbolic complexity bound analysis. In *CAV* (2009), pp. 51–62.

[35] GULWANI, S., JAIN, S., AND KOSKINEN, E. Control-flow refinement and progress invariants for bound analysis. In *PLDI* (2009), pp. 375–385.

[36] GUPTA, A., HENZINGER, T. A., MAJUMDAR, R., RYBALCHENKO, A., AND XU, R.-G. Proving non-termination. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2008), ACM, pp. 147–158.

[37] HENZINGER, T., JHALA, R., MAJUMDAR, R., AND SUTRE, G. Lazy abstraction. In *POPL'02: Principles of Programming Languages* (2002).

[38] HENZINGER, T. A., JHALA, R., MAJUMDAR, R., AND MCMILLAN, K. L. Abstractions from proofs. In *POPL'04: Principles of Programming Languages* (2004).

[39] JEANNET, B. NewPolka polyhedra library. http://pop-art.inrialpes.fr/ people/bjeannet/newpolka/index.html.

[40] LAMPORT, L. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng. 3*, 2 (1977), 125–143.

[41] LEE, C. S., JONES, N. D., AND BEN-AMRAM, A. M. The size-change principle for program termination. In *POPL'01: Principles of Programming Languages* (2001).

[42] MAUBORGNE, L., AND RIVAL, X. Trace partitioning in abstract interpretation based static analyzers. In *ESOP'05: European Symposium on Programming* (2005).

[43] MINÉ, A. The octagon abstract domain. In *AST 2001 in WCRE 2001* (October 2001), IEEE. http://www.di.ens.fr/~mine/publi/article-mine-ast01.pdf.

[44] MINÉ, A. The Octagon abstract domain. *Higher-Order and Symbolic Computation* (2006). (to appear).

[45] NECULA, G., MCPEAK, S., RAHUL, S., AND WEIMER, W. CIL:intermediate language and tools for analysis and transformation of C programs. In *CC'02* (2002).

[46] OHLEBUSCH, E., CLAVES, C., AND MARCHÉ, C. The talp tool for termination analysis of logic programs. In *Extended Abstracts of the 6th International Workshop on Termination, WST'03* (June 2003).

[47] PODELSKI, A., AND RYBALCHENKO, A. A complete method for the synthesis of linear ranking functions. In *VMCAI'04* (2004).

[48] PODELSKI, A., AND RYBALCHENKO, A. A complete method for the synthesis of linear ranking functions. In *VMCAI'04: Verification, Model Checking, and Abstract Interpretation* (2004).

[49] PODELSKI, A., AND RYBALCHENKO, A. Transition invariants. In *LICS'04* (2004).

[50] PODELSKI, A., AND RYBALCHENKO, A. Transition invariants. In *LICS'04: Logic in Computer Science* (2004).

[51] PODELSKI, A., SCHAEFER, I., AND WAGNER, S. Summaries for while programs with recursion. In *ESOP'05: European Symposium on Programming* (2005), S. Sagiv, Ed., LNCS. To appear.

[52] SANKARANARAYANAN, S., IVANCIC, F., SHLYAKHTER, I., AND GUPTA, A. Static analysis in disjunctive numerical domains. In *SAS'06: Static Analysis Symposium* (2006).

[53] SCHRIJVER, A. *Theory of linear and integer programming*. John Wiley & Sons, Inc., New York, NY, USA, 1986.

[54] SEARCOID, M. *Metric Spaces*. Springer, 2007.

[55] SVANBERG, K. Farkas lemma derived by elementary linear algebra.

[56] TURING, A. On computable numbers, with an application to the Entscheidungsproblem. *London Mathematical Society 42*, 2 (1936).

[57] VAN BREUGEL, F. An introduction to metric semantics: operational and denotational models for programming and specification languages. *Theoretical Computer Science 258*, 1-2 (2001), 1 – 98.

[58] YAHAV, E., REPS, T., SAGIV, M., AND WILHELM, R. Verifying temporal heap properties specified via evolution logic. *Logic Journal of IGPL* (Sept. 2006).