University of Padova

DEPARTMENT OF MATHEMATICS "TULLIO LEVI-CIVITA"

MASTER DEGREE IN COMPUTER SCIENCE

Titolo della tesi

Supervisor
Prof. Prof 1

Co. Supervisor
Prof. Prof 2

Candidate
Luca Zaninotto

ACADEMIC YEAR 2023-2024

Abstract

Abstract

Acknowledgments

?

Contents

Chapter 1

Background

1.1 Assumptions

Assumption 1.1 (Gödel Numbering)**.** There is a fixed Gödel numbering for programs, which means there exists an *effective bijection* $\gamma : \text{Imp} \to \mathbb{N}$ mapping each program to a natural number.

Observation 1.1 (denumerability of programs)**.** Because of assumption 1.1, the set of programs is denumerable

Assumption 1.2 (Data type)**.** There exists a single data type: $\mathbb{N}$.

The limitation above is not restrictive, different data types can be encoded and decoded into and out of $\mathbb{N}$.

1.2 Recursion Theory

Following [CGR18], we'll provide the general terminology and nottion fr computable funcitons in recursion theory, as in [Cut80; Odi92; Rog87].

Definition 1.1 (Total functions)**.** Let X, Y be two sets. Then

$$X \to Y$$

is the set of all total functions from X to Y.

Definition 1.2 (Partial functions)**.** Let X, Y be two sets. Then

$$X \hookrightarrow Y$$

is the set of all partial functions from X to Y.

Definition 1.3 (Domain of partial functions)**.** Let $f : X \hookrightarrow Y$. $f(x)\downarrow$ means that f is defined on x, $f(x)\uparrow$ means that f is undefined on x. Hence

$$\mathit{dom}(f) = \{x \in X \mid f(x)\downarrow\}$$

Notation 1.1 (Implication overloading)**.** If $S \subseteq Y$ then by $f(x) \in S$ we mean $f(x)\downarrow \Rightarrow f(x) \in S$

Notation 1.2 (Function Equality)**.** If $f, g : X \hookrightarrow Y$ then by $f = g$ we mean that $\mathit{dom}(f) = \mathit{dom}(g)$ and for any $x \in \mathit{dom}(f) = \mathit{dom}(g)$, $f(x) = g(x)$.

Notation 1.3 (Set of partial recursive functions)**.** By $\mathbb{N} \overset{r}{\hookrightarrow} \mathbb{N}$ we denote the set of partial recursive functions on natural numbers

Notation 1.4 (Set of total recursive functions)**.** By $\mathbb{N} \xrightarrow{r} \mathbb{N}$ we denote the set of partial recursive functions on natural numbers

Definition 1.4 (Recursively enumerable set)**.** $A \subseteq \mathbb{N}$ is *recursively enumerable* (r.e. or semidecidable) if $A = dom(f)$ for some $f \in \mathbb{N} \overset{r}{\hookrightarrow} \mathbb{N}$

Definition 1.5 (Recursive set)**.** $A \subseteq \mathbb{N}$ is a recursive set if both A and its complement $\overline{A} = \mathbb{N} \setminus A$ are semidecidable, i.e., there exists some $f \in \mathbb{N} \xrightarrow{r} \mathbb{N}$ s.t.

$$f = \lambda n.(n \in A)?1 : 0$$

Lemma 1.1 (Computable function over a recursive set)**.** *Given* $f : A \xrightarrow{r} B$*, let the domain* A *to be recursive.* B *is at least r. e.*

Proof. $f : A \xrightarrow{r} B$ total recursive function over a recusrive set A. We can write the function

$$\mathcal{X}_B = \lambda x.sg(\mu z.|f(z) - x|)$$

which is computable as it is composition of computable functions. In oder terms, this is function

$$\mathcal{X}_B(x) = \begin{cases} 1 & x \in B \\ \uparrow & \text{otherwise} \end{cases}$$

which is the semi-decision function for B $\square$

Observation 1.2. In general, B is not recursive, as it would means that both $A \leqslant_m B$ and $B \leqslant_m A$, which is not always the case, but it is r.e., as we could always write the inverse function as in lemma 1.1 and derive a semi-decision function for the image of the function.

1.3 Order theory

Within Theoretical Computer Science, especially in the field of semantics, partial orders hold significant importance. They are extensively employed in Abstract Interpretation, as highlighted in [Min18], serving different levels of the theory to model core notions. These notions include the idea of approximation, where certain analysis results may be less precise than others, creating a partial order where some results are incomparable. Moreover, partial orders are fundamental in conveying the concept of soundness: an analysis is deemed sound if its result is more general than the actual behavior. These mathematical notions, essential for discussions surrounding the Abstract Interpretation formalism, primarily involve order and lattice theory.

Definition 1.6 (Partiall ordered set)**.** Let X be a non-empty set, $\sqsubseteq \subseteq X \times X$ be a reflexive, antisymmetric and transitive relation on that set, i.e., $\forall x, y, z \in X$:

1. $x \sqsubseteq x$ (reflexivity)
2. $x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$ (antisymmetry)
3. $x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$ (transitivity)

Then the tuple $\langle X, \sqsubseteq \rangle$ is a *partially ordered set* (POSet).

Definition 1.7 (Least upper bound)**.** Let $\langle X, \sqsubseteq \rangle$ be a POSet and let $Z \subseteq X$. We say that $\overline{z}$ is an *upper bound* on Z if $\forall z \in Z\ z \sqsubseteq \overline{z}$. It is the *least upper bound* of Z (denoted as $\cup_X Z$) if $\forall z' \in Z$ upper bounds on Z, $\overline{z} \sqsubseteq z'$.

Definition 1.8 (Greatest lower bound)**.** Let $\langle X, \sqsubseteq \rangle$ be a POSet and let $Z \subseteq X$. We say that $\overline{z}$ is a *lower bound* on Z if $\forall z \in Z\ \overline{z} \sqsubseteq z$. It is the *greatest lower bound* of Z (denoted as $\cap_X Z$) if $\forall z' \in Z$ upper bounds on Z, $z' \sqsubseteq \overline{z}$.

Usually then we're talking about least and greatest lower bound the bigger set is often implicit, and we therefore simply write $\cup Z$ and $\cap Z$.

In abstract interpretation we often rely on special kinds of POSets, where the existance of the greatest lower bound and the least upper bound is ensured for each subset of the original POSet. These sets are called complete lattices

Definition 1.9 (Complete lattice)**.** A POSet $\langle X, \sqsubseteq \rangle$ is called a *complete lattice* if

$$\forall Y \subseteq X \quad \exists \cup Y \wedge \exists \cap Y$$

We're also interested in join morphisms

Definition 1.10 (Join Morphism)**.** Let $\langle X, \sqsubseteq_X \rangle, \langle Y, \sqsubseteq_Y \rangle$ be two lattices. Let $f : X \to Y$ be a mappging from X to Y. F is a *join morphism* if $\forall x_1, x_2 \in X \; f(x_1 \vee x_2) = f(x_1) \vee f(x_2)$. In this case we can also say that f *preserves the joins.*

1.4 Program semantics

In the next lines we'll refer to a generic deterministic programming language Prog, which is just bounded to be Turing complete.

Definition 1.11 (Denotational semantics)**.** Given a programming language Prog a denotational-style semantics is a function

$$[\![\cdot]\!] : \text{Prog} \to D \hookrightarrow D$$

which maps each and every program to a partial function on a set of I/O values.

In pair with that we usually define a small step semantics which allows us to reason inductively on program executions:

Definition 1.12 (Small-step semantics)**.** A small step transition relation

$$\Rightarrow \subseteq (\text{Prog} \times D) \times ((\text{Prog} \times D) \cup D)$$

is a small step semantics functionally equivalent a denotational semantics (1.11):

$$\langle P, i \rangle \Rightarrow^* o \iff [\![P]\!] i = o$$

These two definitions allow us to reason on abstract terms on the partial functions a program computes, once its semantics is well defined trough the semantic function, effectively defining an interpreter for the language.

Definition 1.13 (Turing completeness)**.** A programming language is said to be Turing complete when for every partial recursive function $f : \mathbb{N} \overset{r}{\hookrightarrow} \mathbb{N}$ there exists a program $P \in \text{Prog}$ s.t.

$$[\![P]\!] \cong f$$

Observation 1.3.

$$\{[\![P]\!] : \mathbb{N} \overset{r}{\hookrightarrow} \mathbb{N} \mid P \in \text{Prog}\} = \mathbb{N} \overset{r}{\hookrightarrow} \mathbb{N}$$

Notation 1.5 (I/O semantics)**.** We'll overload the notation of $[\![P]\!]$ to also identify a program by the partial recursive function it computes. i.e.,

$$[\![P]\!] : \mathbb{N} \overset{r}{\hookrightarrow} \mathbb{N}$$

1.4.1 Collecting semantics

Collecting semantics is a special kind of semantics, which operates on sets of memory states:

Definition 1.14 (Memory states)**.** A memory state $\rho \in \text{Env}$ is a function mapping each variable to its value

$$\text{Env} \triangleq \{\rho \mid \rho : \text{Var} \to \mathbb{Z}\}$$

Memory states are used to infer a complete lattice, called the *concrete domain*:

Definition 1.15 (Concrete domain)**.** The concrete domain of a collecting semantics is the complete lattice $\wp(\text{Env})$, paired with the inclusion relation as partial order, the empty set as bottom element and the set Env as top element, union and intersection defined as usual on sets.

$$\mathbb{C} \triangleq \langle \wp(\text{Env}), \subseteq, \cup, \cap, \varnothing, \text{Env} \rangle$$

The concrete domain is used in the concrete collecting semantics as before ($D \cong \mathbb{C}$):

Definition 1.16 (Collecting semantics)**.** Let Prog be a non-deterministic turing complete language. That operates on numeric values (in pair with assumption 1.2). The function

$$\langle \cdot \rangle : \text{Prog} \to \mathbb{C} \to \mathbb{C}$$

is the collecting semantics of the language.

Notice how the collecting semantics is similar to the general program semantics $[\![\cdot]\!] : \text{Prog} \to D \to D$; in fact, we can say that

$$\forall P \in \text{Prog} \quad \langle P \rangle \{\rho\} = \varnothing \iff [\![P]\!]\rho \uparrow$$

Lemma 1.2 (Collecting semantics undecidability)**.** *In general the statement*

$$Q(P, X) = "\langle P \rangle X \downarrow"$$

is undecidable.

Proof. To prove that, we can just observe that for a single state ρ deciding weather $[\![P]\!]\rho \downarrow$ is undecidable, as it would mean that the set

$$B_n = \{x \mid n \in dom(\varphi_x)\}$$

is recursive, which is not, which is a well known fact coming from the input problem [Cut80, p. 104]. Since given a single state ρ deciding weather $\langle P \rangle \{\rho\} \downarrow$ is undecidable, the statement $Q(P, X)$ is in general undecidable. □

We're however interested in computer rapresentable elements of the concrete domain $\mathbb{C}$. Because of observation 1.2 if we have a recursive set of environments $X \in 2^{\text{Env}}$, their image $\langle C \rangle X = Y$ will in general be r.e.

1.5 Abstract Interpretation

For the following section we're following Mine's notes on abstract interpretation [Min18].

Definition 1.17 (Galois connection)**.** A *Galois connection* is a tuple (α, C, A, γ) where

1. A, C are complete lattices
2. $\alpha : C \to A$, $\gamma : A \to C$ are a monotone *abstraction* and *concretization* maps.
3. $\forall a \in A, c \in C$ we have that $c \sqsubseteq_C \gamma(a) \iff a \sqsubseteq_A \alpha(c)$

$$\begin{tikzcd}
A \arrow[r, bend right, "\gamma"'] & C \arrow[l, bend right, "\alpha"']
\end{tikzcd}$$

Figure 1.1: Galois connection between an abstract domain A and a concrete domain C

We're biulding a mapping between an *abstract domain* A and a *concrete domain* C, where the concretization and abstraction maps α, γ are sound, a visual rapresentation is in figure 1.1.

Definition 1.18 (Galois Insertion)**.** A Galois connection (α, C, A, γ) is a *Galois Insertion* if one of the (equivalent) following conditions hold:

1. α is surjective;
2. γ is injective;
3. $\forall a \in A \quad \alpha(\gamma(a)) = a$

The intuition is that α is a bijection, whose inverse function is γ, this way the abstract and the concrete domain have the same cardinality, and therefore there are no "useless" abstract values.

Definition 1.19 (Abstract domain)**.** An *abstract domain* is a tuple $(D^\sharp, \sqsubseteq^\sharp, \gamma)$:

- a denumerable set $D^\sharp$ of abstract values;
- a partial order $\sqsubseteq^\sharp$ on $D^\sharp$;
- a monotone concretization map $\gamma : D^\sharp \to D$.

We denote with $\perp^\sharp \in D^\sharp$ the smallest element of the abstract domain. Similarly, with $\top^\sharp \in D^\sharp$ we rapresent the biggest element of the abstract domain. The concretization function sould express the map $\gamma(\perp^\sharp) = \varnothing$ and $\gamma(\top^\sharp) = \text{Env}$.

Chapter 2

Framework

2.1 The Imp language

We'll denote by $\mathbb{Z}$ the set of integers with the usual order plus two bonus elements $-\infty$ and $+\infty$, s.t. $-\infty \leqslant z \leqslant +\infty \quad \forall z \in \mathbb{Z}$. We also extend addition and subtraction by letting, for $z \in \mathbb{Z} \quad +\infty + z = +\infty - z = +\infty$ and $-\infty + z = -\infty - z = -\infty$.

We'll focus on the following non-deterministic language.

$$\begin{aligned}
\text{Exp} \ni e ::= \quad & x \in S \mid x \in [a, b] \mid x \leqslant k \mid x > k \mid \texttt{true} \mid \texttt{false} \mid \\
& x := k \mid x := y + k \mid x := y - k \\
\text{Imp} \ni C ::= \quad & e \mid C + C \mid C; C \mid C*
\end{aligned}$$

where $x, y \in \text{Var}$ a finite set of variables of interest, i.e., the variables appearing in the considered program, $S \subseteq \mathbb{N}$ is (possibly empty) subset of numbers, $a \in \mathbb{Z} \cup \{-\infty\}, b \in \mathbb{Z} \cup \{+\infty\}, a \leqslant b, k \in \mathbb{Z}$ is any finite integer constant.

2.2 Semantics

In the following section we'll provide the semantics of the language we're working on and we'll make some observations; we'll also prove some properties of the language we'll use in the next sections.

The first building block is that of environments. We'll use environments to model the most precise invariant our semantic can describe for a program.

Definition 2.1 (Environments)**.** Environments are (total) maps from variables to (numerical) values

$$\text{Env} = \{\rho \mid \rho : \text{Var} \to \mathbb{Z}\}$$

Note: the set of notable variables is assumed to be finite.

An environment is therefore a map from a finite set of interesting variables (Var) to a set of possible values the variables can assume (because of assumption 1.2 we're restraining ourselves to integer values, without loss of generality as more complex datatypes can be encoded into $\mathbb{N}$ values, as long as we provide an enconding function $\xi : D \to \mathbb{N}$ for some data type D.

The next building block is the semantics of basic expressions, which encodes the most important operations we can do on variables: tests and assignments. Tests are used in the semantics to filter out some states, namely those state that do not respect some condition, while assignments are used as a way of updating the state in which our machine finds itself during the execution of a program.

Definition 2.2 (Semantics of Basic Expressions)**.** For basic expressions $e \in \text{Exp}$ the concrete semantics $(\!|\cdot|\!) : \text{Exp} \to \text{Env} \to \text{Env} \cup \{\bot\}$ is recursively defined as follows:

$$
\begin{aligned}
(\!|x \in S|\!)\rho &\triangleq \begin{cases} \rho & \rho(x) \in S \\ \bot & \text{otherwise} \end{cases} \\
(\!|x \in [a,b]|\!)\rho &\triangleq \begin{cases} \rho & \rho(x) \in [a,b] \\ \bot & \text{otherwise} \end{cases} \\
(\!|x \leqslant k|\!)\rho &\triangleq \begin{cases} \rho & \rho(x) \leqslant k \\ \bot & \text{otherwise} \end{cases} \\
(\!|x > k|\!)\rho &\triangleq \begin{cases} \rho & \rho(x) > k \\ \bot & \text{otherwise} \end{cases} \\
(\!|\texttt{true}|\!)\rho &\triangleq \rho \\
(\!|\texttt{false}|\!)\rho &\triangleq \bot \\
(\!|x := k|\!) &\triangleq \rho[x \mapsto k] \\
(\!|x := y + k|\!) &\triangleq \rho[x \mapsto \rho(y) + k] \\
(\!|x := y - k|\!) &\triangleq \rho[x \mapsto \rho(y) - k]
\end{aligned}
$$

The next building block is the concrete collecting semantics for the language, maps each program in Imp to a function on the $\mathbb{C}$ complete lattice.

Definition 2.3 (Concrete collecting domain)**.** The concrete collecting domain for the Imp language concrete collecting semantics is the complete lattice

$$\mathbb{C} \triangleq \langle 2^{\text{Env}}, \subseteq \rangle$$

We can therefore define the concrete collecting semantics for our language:

Definition 2.4 (Concrete collecting semantics)**.** The concrete collecting semantics for Imp is given by the total mapping

$$\langle \cdot \rangle : \text{Imp} \to \mathbb{C} \to \mathbb{C}$$

which maps each program $C \in \text{Imp}$ to its total mapping

$$\langle C \rangle : \mathbb{C} \to \mathbb{C}$$

on the complete lattice $\mathbb{C}$. The semantics is recursively defined as follows: given $X \in 2^{\text{Env}}$

$$
\begin{aligned}
\langle e \rangle X &\triangleq \{(\!|e|\!)\rho \mid \rho \in X, (\!|e|\!)\rho \neq \bot\} \\
\langle C_1 + C_2 \rangle X &\triangleq \langle C_1 \rangle X \cup \langle C_2 \rangle X \\
\langle C_1; C_2 \rangle X &\triangleq \langle C_2 \rangle (\langle C_1 \rangle X) \\
\langle C^* \rangle X &\triangleq \bigcup_{i \in \mathbb{N}} \langle C \rangle^i X
\end{aligned}
$$

Definition 2.5 (Program length)**.** Given a program $C \in \text{Imp}$ its length is recursively defined as follows:

$$
\begin{aligned}
len(e) &\triangleq 1 \\
len(C_1 + C_2) &\triangleq len(C_1) + len(C_2) + 1 \\
len(C_1; C_2) &\triangleq len(C_1) + len(C_2) + 1 \\
len(C^*) &\triangleq len(C) + 1
\end{aligned}
$$

Along with the collecting semantics we're also defining a one step transition relation. This will be useful to prove that finiteness is not decidable, as it helps to reason about program execution in a recursive way. The relation is defined on program states:

Definition 2.6 (Program State)**.** Program states are tuples of programs and program environments:

$$\text{State} \triangleq \text{Imp} \times \text{Env}$$

Definition 2.7 (Small step semantics)**.** The small step transition relation $\rightarrow$: $\text{State} \times (\text{State} \cup \text{Env})$ is a small step semantics for the Imp language. It is defined based on the following rules

$$\frac{(\!|e|\!)\rho \neq \bot}{\langle e, \rho \rangle \rightarrow (\!|e|\!)\rho}\ \text{expr}$$

$$\frac{}{\langle C_1 + C_2, \rho \rangle \rightarrow \langle C_1, \rho \rangle}\ \text{sum}_1 \qquad \frac{}{\langle C_1 + C_2, \rho \rangle \rightarrow \langle C_2, \rho \rangle}\ \text{sum}_2$$

$$\frac{\langle C_1, \rho \rangle \rightarrow \langle C_1', \rho' \rangle}{\langle C_1; C_2, \rho \rangle \rightarrow \langle C_1'; C_2, \rho' \rangle}\ \text{comp}_1 \qquad \frac{\langle C_1, \rho \rangle \rightarrow \rho'}{\langle C_1; C_2, \rho \rangle \rightarrow \langle C_2, \rho' \rangle}\ \text{comp}_2$$

$$\frac{}{\langle C^*, \rho \rangle \rightarrow \langle C; C^*, \rho \rangle}\ \text{star} \qquad \frac{\langle C, \rho \rangle \rightarrow^* \rho}{\langle C^*, \rho \rangle \rightarrow \rho}\ \text{star}_{\text{fix}}$$

As we can see non-determinism can produce multiple traces stargin from the same program and initial state. This does not affect the expressivity of the language, but makes it harder to reason about program execution, insted of single traces we should instead speak about execution graphs.

Definition 2.8 (Execution graphs)**.** An execution graph is a transition system where

$$\text{Exe} \triangleq (\Gamma, T, \Rightarrow)$$

where $\Gamma \subseteq \text{State} \cup \text{Env}$ is the subset of states we're considering, $T = \text{Env}$ is the set of final states of the transition system and $\Rightarrow \subseteq (\text{State} \times \Gamma)$ s.t. $(s,t) \in E \Rightarrow s \rightarrow t$ is the transitions that occur in the system.

Usually we're interested in a particular execution graph for some program C, starting from a state ρ:

Definition 2.9 (Execution graph)**.** Given a program $C \in \text{Imp}$ and an initial state $\rho \in \text{Env}$ its execution graphs are defined as

$$\text{Exe}(C, \rho) \triangleq \{(\Gamma, T, \Rightarrow) \in \text{Exe} \mid \langle C, \rho \rangle \in \Gamma \wedge \forall s \in \Gamma, t \in \text{State}. s \rightarrow t \Rightarrow t \in \Gamma \wedge (s,t) \in \Rightarrow\}$$

Example 2.1. Consider the program $C = x := 1; (((x \leqslant 5; x := x + 1)*; x > 5) + (x := 6))$ and the initial environment $\varnothing$. Figure 2.1 shows a part of the execution graph of program C.

Observation 2.1 (Basic graph constructions)**.** We can observe that the semantics of a program identifies some building blocks of the execution graph:

- $\langle e, \rho \rangle$ identifies a node with on single child which is a terminal environment $(\!|e|\!)\rho$ (as in figure 2.2)
- $\langle C_1 + C_2, \rho \rangle$ In this case the node has 2 children, one labelled by the execution of the first branch, one by the execution of the other, as shown in figure 2.4
- $\langle C_1; C_2, \rho \rangle$ In this case we should label the intermediate executions $\langle C_1 \rangle X = Y$ so that we cna rapresent the execution graph, shown in figure **??**
- $\langle C*, \rho \rangle$ in this case 2 things can happen. Either we produce an infinite path, a cycle, or at some point we can apply star_{fix}

$$\begin{tikzcd}
& \langle x := 1; (((x \leqslant 5; x := x+1)*; x > 5) + (x := 6)), \varnothing\rangle \arrow[d] & \\
& \langle x := 1; (((x \leqslant 5; x := x+1)*; x > 5) + (x := 6)), \varnothing[x \mapsto 1]\rangle \arrow[dl] \arrow[dr] & \\
\langle ((x \leqslant 5; x := x+1)*; x > 5), \varnothing[x \mapsto 1]\rangle \arrow[d] & & \langle x := 6, \varnothing[x \mapsto 1]\rangle \arrow[dl] \\
\cdots \arrow[r] & \varnothing[x \mapsto 6] &
\end{tikzcd}$$

Figure 2.1: Partial execution graph of C

$$\begin{tikzcd}
\cdots \arrow[r] & \langle e, \rho\rangle \arrow[r] & (\!|e|\!)\rho(\neq \bot)
\end{tikzcd}$$

Figure 2.2: single expression execution semantics

$$\begin{tikzcd}
& \cdots \arrow[d] & \\
& \langle C_1 + C_2, \rho\rangle \arrow[dl] \arrow[dr] & \\
\langle C_1, \rho\rangle & & \langle C_2, \rho\rangle
\end{tikzcd}$$

Figure 2.3: Execution of a + instruction

$$\begin{tikzcd}
\cdots \arrow[r] & \langle C_1; C_2, \rho\rangle \arrow[r] & \langle C_1'; C_2, \rho'\rangle \arrow[r] & \cdots \arrow[r] & \langle C_2, \rho''\rangle \arrow[r] & \cdots
\end{tikzcd}$$

Figure 2.4: Execution of concatenated instructions

Lemma 2.1 (finite sets on finite traces)**.** *Let $X \in 2^{Env}$ be a finite set of environments, $C \in Imp$ a program in the Imp language for which we know goes trough finite amount of states in its execution graph. Therefore*

$$\langle C\rangle X \textit{ is finite}$$

Proof. We'll prove it by structural induction on the length of the program C:

Base case:
$C \equiv e$, therefore $\langle e\rangle X = \{(\!|e|\!)\rho \mid \rho \in X, (\!|e|\!)\rho \neq \bot\}$ has at most as much elements as X, which is a finite quantity.

Inductive cases:

- $C \equiv C_1 + C_2$, threfore $\langle C_1 + C_2\rangle X = \langle C_1\rangle X \cup \langle C_2\rangle X$ by induction, $\langle C_1\rangle X$ and $\langle C_2\rangle X$ are finite (as C_1, C_2 are programs of length strictly smaller than C), therefore their union $\langle C_1\rangle X \cup \langle C_2\rangle X = \langle C_1 + C_2\rangle X$ is finite;
- $C \equiv C_1; C_2$, therefore $\langle C_1; C_2\rangle X = \langle C_1\rangle(\langle C_1\rangle X)$. By induction since C_1 is strictly smaller than C, $\langle C_1\rangle X = Y$ is finite, therefore, again by induction $\langle C_2\rangle Y$ is finite;
- $C \equiv C^*$, therefore $\langle C^*\rangle X = \bigcup_{i\in\mathbb{N}}\langle C\rangle^i X$. By hypothesis it terminates, therefore $\exists i \in \mathbb{N} \mid \langle C\rangle^i X = \langle C\rangle^{i+k} X \forall k \in \mathbb{N}$, which is the least upper bound on the lattice. We have to use induction again on the number $j \in [i; k]$ of applications of $\langle C\rangle$:

base case: one step, C has length strictly smaller than C^* and X is finite, therefore $\langle C\rangle X = X_1$ is finite.

recursive case: k steps: X_k is again finite by inductive hypothesis. Since C is stricly smaller than C^* we can use the other inductive hypothesis and say that $\langle C\rangle X_k = X_{k+1}$ is again, finite.

□

Observation 2.2. The latter theorem states that given a finite set of initial states with the collecting semantics we defined and knowing that the program terminates, then the invariant on the final step of the computation will be finite. This implies that from a finite set of initial environments an *infinite* set of environments can be produced only by a program which has a branch of its execution graph which is infinite.

Lemma 2.2. *Let $X \in 2^{Env}, C \in Imp$ be respectively a finite collection of environments and a finite program. If $\langle C\rangle X$ is infinite then the execution graph contains an infinite branch.*

Proof. We can use the same intuition as before an reason inductively on the length of the program C.

Base case:
$C \equiv e$, but $\langle e\rangle X$ is by definition finite, so the statement holds vacuously.

Recursive cases:

- $C \equiv C_1 + C_2$, $\langle C_1 + C_2\rangle X = \langle C_1\rangle X \cup \langle C_2\rangle X$. $\langle C_1 + C_2\rangle X$ is infinite, therefore since C_1, C_2 have a length strictly smaller than C by inductive hypothesis either one of $\langle C_1\rangle X$ $\langle C_2\rangle X)$ is infinite, and contains an infinite branch. Either case, in the end $\langle C\rangle X$ if infinite, contains an infinite branch.
- $C \equiv C_1; C_2$, $\langle C_1; C_2\rangle X = \langle C_2\rangle(\langle C_1\rangle X)$. We have 2 cases: either $\langle C_1\rangle X$ is infinite, and sice C_1 is striclty smaller than C by inductive hypothesis it contains an infinite branch in the execution graph or $\langle C_1\rangle X = Y$ is finite, but in this case $\langle C_2\rangle Y$ if infinite we can apply the inductive hypothesis and say that it contains an infinite branch in the execution graph.

$$\begin{tikzcd}
& \langle (x := 1; (x \leqslant 1; x := 1)^*), \varnothing \rangle \arrow[d] \\
& \langle (x \leqslant 1; x := 1)^*, \varnothing[x \mapsto 1] \rangle \arrow[dl] \\
\langle (x \leqslant 1; x := 1); (\ldots)^*, \varnothing[x \mapsto 1] \rangle \arrow[d] & \\
\langle x := 1; (\ldots)^*, \varnothing[x \mapsto 1] \rangle \arrow[ruu, bend right] &
\end{tikzcd}$$

Figure 2.5: Execution graph of $(x := 1; (x \leqslant 1; x := 1)^*)$

- $C \equiv C_1*$ $\langle C_1*\rangle X = \cup_{i \in \mathbb{N}} \langle C_1 \rangle^i X$. In this case the invariant is infinite only if there's no $j \in \mathbb{N} \mid \langle C \rangle^j X_{j-1} = \langle C \rangle^{j+1} X_j$, which means that we're creating an infinite chain of $X_i \forall i \in \mathbb{N} \mid \forall j, k \in \mathbb{N} X_j \neq X_k$, and therefore there is an infinte path

  $$\forall j \in \mathbb{N} \quad \langle C_1^*, X_1 \rangle \Rightarrow^* \langle C_1^*, X_2 \rangle \Rightarrow \ldots \Rightarrow \langle C_1^*, X_j \rangle \Rightarrow \ldots$$

  which is in the execution graph for the original program C_1^*.

□

Example 2.2. This lemma provides an insights that links non-termination to an infinite path in the execution graph, but it is not the only way a program might not terminate. In fact, considering the program $C = (x := 1; (x \leqslant 1; x := 1)^*)$ which of course does not halt on the empty environment $\varnothing$ and has the execution graph in figure 2.5, which is composed of a finite amount of states, but since it contains a cycle, the program does not terminate.

The next step is to prove that a given a finite set of initial states and a program, if we know that the final invariant is finite, we cound decide termination:

Lemma 2.3 (Finite invariant on finite program)**.** *Given a finite set of initial states $X \in 2^{Env}$, a program $C \in Imp$, "$\langle C \rangle X = Y$ is finite" is undecidable.*

Proof. Suppose we can decide weather $\langle C \rangle X$ (the most precise invariant for the program C) is finite. Based on the result we have 2 options:

- **The invariant is infinite**: in this case because of observation 2.2, we can say that the original program does not halt, as it would not terminate in a finite amount of steps. Additionally, because of lemma 2.2 the execution graph of the program contains an infinite branch, which is the one responsible for non-termination.

- **The invariant is finite**: This does not give direct informations about the termination of the program, because like in example 2.2 the program can have a finite invariant but it might not terminate due to the presence of a cycle in the termination graph. However it means that there's no infinite branch in the execution graph of C.

  The initial set of environments is finite, therefore consider $\forall \rho \in X$, the execution graph $\text{Exe}(C, \rho)$: It is composed of a finite set of environments, therefore by running any cycle-finding algorithm (bfs should do, as in [Cor+22]) we have 2 kinds of output:

  – there is a cycle, in this case the execution graphs has the shape as in figure 2.6, therefore there's a possible execution that never terminates.

□

$$\begin{tikzcd}
\cdot \arrow[r] & \cdot \arrow[r] \arrow[d, bend left] & \cdot \\
 & \cdots \arrow[u, bend left] &
\end{tikzcd}$$

Figure 2.6: example of an execution graph that contains a cycle

Chapter 3

Intervals

Interval semantics and analysis are among the most well known abstract interpretation standard abstract domains. Are generally studied as simple non-relational domains, as intervals are not able to capture the relation between variables occurring in the program

The following chapter aims to prove the fact that interval analysis is decidable without a widening operator, i.e., infinite ascending chains can be decided.

3.1 Interval Analysis

The following chapter aims to introduce what we mean by interval analysis, giving some background and definition and finally proving the decidability of the analysis without the use of a widening operator. Notice that this is not in contrast to the more general results on concrete semantics, as interval analysis, falling under the broader umbrella of static analysis aims to provide a sound but maybe not complete analysis results.

Definition 3.1 (Interval domain)**.** We call by

$$Int \triangleq \{[a, b] \mid a \in \mathbb{Z} \cup \{-\infty\} \wedge b \in \mathbb{Z} \cup \{+\infty\} \wedge a \leqslant b\}$$

the interval domain.

Definition 3.2 (Concretization map)**.** The concretization map $\gamma : Int \to 2^{\mathbb{Z}}$ is the following

$$\gamma([a, b]) = \{x \in \mathbb{Z} \mid a \leqslant x \leqslant b\}; \quad \gamma(\bot) = \varnothing$$

Definition 3.3 (Abstract domain)**.** The We'll call by $\mathbb{A}$ the abstract domain

$$\mathbb{A} \triangleq (Var \to int_*) \cup \{\bot\}$$

Definition 3.4 (Abstract concretization)**.** The abstract domain $\mathbb{A}$ concretization map is

$$\gamma(\bot) \triangleq \varnothing \tag{3.1}$$

$$\forall \eta \neq \bot, \quad \gamma(\eta) \triangleq \{\rho \in \text{Env} \mid \forall x \in \text{Var}.\rho(x) \in \gamma(\eta(x))\} \tag{3.2}$$

Chapter 4

Non relational collecting

Bibliography

[CGR18] Patrick Cousot, Roberto Giacobazzi, and Francesco Ranzato. “Program Analysis Is Harder Than Verification: A Computability Perspective”. In: *Computer Aided Verification*. Ed. by Hana Chockler and Georg Weissenbacher. Cham: Springer International Publishing, 2018, pp. 75–95. ISBN: 978-3-319-96142-2.

[Cor+22] Thomas H Cormen et al. *Introduction to algorithms*. MIT press, 2022.

[Cut80] Nigel Cutland. *Computability: An introduction to recursive function theory*. Cambridge university press, 1980.

[Min18] Antonie Miné. *Static Inference of Numeric Invariants by Abstract Interpretation*. Université Pierre et Marie Curie, Paris, France, 2018.

[Odi92] Piergiorgio Odifreddi. *Classical recursion theory: The theory of functions and sets of natural numbers*. Elsevier, 1992.

[Rog87] Hartley Rogers Jr. *Theory of recursive functions and effective computability*. MIT press, 1987.