# Analyses statiques pour sémantiques squelettiques

Vincent Rébiscoul

COLLEGE DOCTORAL BRETAGNE
MATHS, TELECOMS
INFORMATIQUE, SIGNAL
SYSTEMES, ELECTRONIQUE

Université de Rennes

# THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES

ÉCOLE DOCTORALE N° 601
*Mathématiques, Télécommunications, Informatique, Signal, Systèmes, Électronique*
Spécialité : *Informatique*

Par

**Vincent RÉBISCOUL**

## Static Analysis for Skeletal Semantics

Analyse Statique pour Sémantique Squelettique

**Thèse présentée et soutenue à Centre Inria de l'Université de Rennes, le 14 Mai 2024**
**Unité de recherche : Irisa**

### Rapporteurs avant soutenance :

Antoine MINÉ — Professeur, Sorbonne Université
Tom HIRSCHOWITZ — Directeur de Recherche, LAMA

### Composition du Jury :

| | | |
|---|---|---|
| Examinateurs : | Ilaria CASTELLANI | Chargée de Recherche, INRIA Sophia-Antipolis |
| | Tom HIRSCHOWITZ | Directeur de Recherche, LAMA |
| | Antoine MINÉ | Professeur, Sorbonne Université |
| | Anne-Cécile ORGERIE | Directrice de Recherche, CNRS/IRISA |
| Dir. de thèse : | Thomas JENSEN | Directeur de Recherche, INRIA Rennes |
| Co-dir. de thèse : | Alan SCHMITT | Directeur de Recherche, INRIA Rennes |

# Remerciements

Avant de rentrer dans le vif du sujet, je tiens à remercier tous ceux qui m'ont aidé de près ou de loin durant ces trois années de thèse.

Je suis particulièrement reconnaissant envers Thomas et Alan, qui m'ont rigoureusement encadré. Vous avez toujours été disponible, et chacun m'a apporté son expertise. Alan, tu m'as énormément aidé à comprendre les sémantiques squelettiques. Ton expertise technique m'a bien souvent permis d'avancer. Thomas, tu m'as rappelé (plusieurs fois) que la recherche se plaçait dans un contexte, et que l'on devait se comparer, s'inspirer, et se différencier du travail réalisé par d'autres. Aussi, vous avez été très compréhensif lors de moments difficiles.

Je salue aussi l'ensemble de l'équipe Épicure. Les discussions autour d'un café et les séminaires d'équipe ont été de très bons moments à partager. Cela a grandement participé à mon envie de venir au laboratoire tous les jours, et au plaisir que j'ai pris à travailler.

Je remercie les collègues du bureau F213 avec qui j'ai beaucoup ri (et qui m'ont parfois distrait de mon travail). Pierre, merci d'avoir mi à disposition ton appartement pour les soirées rugby ou dégustation de gin. Victoire, merci pour tes analyses sociétales et ton aide sur les sémantiques squelettiques.

Merci à Sarah de m'avoir accueilli dans sa colocation, tu es toujours disponible et prête à aider. Merci à Théo et Jean-Loup, dont le bureau a été un refuge quand les journées devenaient longues. Merci à Lucas et Calvin d'être venu à Rennes quand j'en ai eu grandement besoin.

Merci à toute ma fratrie : Nathan et Flora de s'être déplacés pour ma soutenance, Xavier d'avoir été un soutien moral pendant une année compliquée, Laurence d'avoir partagé ma douleur lorsque l'on devait chacun rédiger un manuscrit. Je remercie ma mère, Chantal, pour avoir préparé un excellent pot de thèse. Merci a Nadja pour tes pastillas.

Je dédie cette thèse à mon père, qui aurait probablement été très fier (et un peu jaloux).

# Résumé en Français

Nous interagissons de plus en plus avec des objets pilotés par des programmes. Souvent, une erreur d'exécution d'un programme ne provoque pas de situation dangereuse. Un *plantage* de téléphone peut être embêtant, mais est souvent sans conséquences graves. Néanmoins, il existe des programmes gérant des tâches critiques, et les échecs de ceux-ci peuvent avoir des conséquences graves. Les pays membres de l'Agence Spatiale Européenne gardent en mémoire le vol 501 d'Ariane, le premier vol de la fusée Ariane 5. Celui-ci s'est terminé prématurément après 30 secondes, lorsque le pilote automatique a guidé la fusée hors de la trajectoire prévue, et celle-ci a été détruite par sécurité [25]. Cet événement est la conséquence d'une erreur de programmation, et a été à la date du vol en 1995 « le bug le plus cher de l'histoire ». Bien que les conséquences matérielles ont été grandes, il n'y eut pas de victimes. En 2016, un programme de financement participatif automatisé (un *smart contract*) s'exécutant sur la blockchain Ethereum s'est révélé vulnérable. Le rôle de ce programme était de recevoir des cryptomonnaies d'utilisateurs, dans l'objectif de financer des projets. Or, celui-ci présentait une faille de sécurité, et un utilisateur malicieux a réussi à extraire une part importante de l'argent possédée par le programme [12]. Ce piratage a pu avoir lieu, car les développeurs du programme avaient une mauvaise compréhension du langage qu'ils utilisaient. Il y avait un écart entre le comportement attendu et le comportement réel du programme. Le vol a été chiffré en dizaines de millions de dollars.

S'assurer d'avoir des programmes sûrs, et qui remplissent bien les fonctions qui leur sont attribuées est naturellement un enjeu important. Dans cette thèse, nous allons nous intéresser à la définition d'analyses statiques correctes pour langages de programmation. Une analyse statique analyse du code sans l'exécuter. Elle peut garantir certaines propriétés d'un programme, par exemple, montrer qu'il n'y a pas de division par zéro. Nous souhaitons définir des analyses correctes, c'est-à-dire avec une preuve formelle, mathématique, que les analyses calculent effectivement les propriétés attendues.

Pour atteindre ces objectifs, notre approche est de partir d'une description formelle d'un langage de programmation, et d'en dériver une interprétation abstraite. L'interprétation abstraite [9] est une méthode de calcul d'approximations des comportements des programmes, et d'en déduire des propriétés sur ces programmes. Nous avons choisi les Sé-

mantiques Squelettiques comme cadre pour formaliser des langages de programmation, car elles sont simples mais très expressives. Une sémantique squelettique est une description partielle d'un langage. En complétant cette description, on peut obtenir une *sémantique* du langage, c'est-à-dire une description mathématique de l'exécution des programmes de ce langage. Notre objectif est d'obtenir une sémantique et une interprétation abstraite à partir de la sémantique squelettique d'un langage, et de montrer que l'interprétation abstraite est une bonne approximation de la sémantique du langage. Nous présentons ici un résumé des différents chapitres de cette thèse.

**Contexte** Le Chapitre 1 présente différents formats sémantiques tel que les sémantiques opérationnelles. Nous présentons aussi des sémantiques « mécanisées », c'est à dire représentable et manipulable par une machine. Nous introduisons l'interprétation abstraite, une méthode pour calculer des approximations de sémantiques en temps fini. Nous en arrivons à la conclusion qu'il est intéressant de générer des analyses statiques, sous forme d'interprétation abstraite à partir de sémantiques mécanisées. En effet, cela nous permet de générer une interprétation abstraite pour chaque langage dont on a une sémantique mécanisée, et une preuve de correction de cette interprétation abstraite. Notre objectif est de gagner en temps : l'interprétation abstraite est en partie générée, et de gagner en sûreté car une preuve de correction est aussi produite.

**Sémantique Squelettiques** Le Chapitre 2 présente les sémantiques squelettiques : un cadre sémantique pour formaliser des langages. Une sémantique squelettique est une description formelle et partielle d'un langage. Nous présentons la sémantique squelettique d'un petit langage impératif que nous appelons While. De plus, nous montrons comment l'on peut obtenir une sémantique pour un langage dont on a une sémantique squelettique : nous l'appellerons ici *sémantique concrète*. Cette sémantique concrète décrit le comportement des programmes du langage. La définition de la sémantique est simplifiée et elle est représentable par une machine.

**Points de Programme** Le Chapitre 3 est notre première contribution. Nous présentons les « points de programme » : pour un programme donné, un point de programme permet de désigner avec précision une sous partie du programme. Ce travail est très important, car les points de programme sont indispensables pour faire des analyses de programme. Dans ce chapitre, nous présentons notre méthode pour intégrer les points de programmes

aux sémantiques obtenues à partir de sémantiques squelettiques. Nous montrons qu'il y a une équivalence entre la sémantique sans, et avec, points de programmes.

**L'Interprétation Abstraite** Le Chapitre 4 est le cœur de cette thèse. Il montre comment définir une interprétation abstraite d'un langage à partir de sa sémantique squelettique. Pour cela, nous présentons une méthode inspirée de la définition de sémantique concrète à partir d'une sémantique squelettique. La définition d'une interprétation abstraite est plus compliquée, car l'on doit y intégrer des concepts issus de la théorie de l'interprétation abstraite, comme les unions et les comparaisons abstraites. De plus, on ajoute des mécanismes pour s'assurer que l'interprétation abstraite termine. Enfin, le théorème central de la thèse est présenté : étant donné la sémantique squelettique d'un langage, la sémantique concrète obtenue par la méthode du Chapitre 2 est correctement approximée par l'interprétation abstraite obtenue dans ce chapitre. Nous donnons un exemple d'interprétation abstraite obtenue à partir de la sémantique squelettique de While et nous montrons que celle-ci est correcte : elle approxime bien la sémantique de While.

**Le $\lambda$-calcul et Analyse de Flot de Contrôle** Le Chapitre 5 donne un nouvel exemple de définition d'interprétation abstraite pour un autre langage, le $\lambda$-calcul, à partir de sa sémantique squelettique. Avec la même recette, nous obtenons une analyse très différente pour ce langage fonctionnel. Nous définissons cette fois une *Analyse de Flot de Contrôle*. En effet, les langages fonctionnels comme le $\lambda$-calcul ont un flot de contrôle qui n'est pas connu avant l'exécution du programme. Il existe des analyses pour en calculer une approximation, et nous en définissons une en utilisant notre méthodologie.

**Taiga** Le Chapitre 6 présente notre générateur d'interpréteur abstrait exécutable. Taiga est un programme OCaml qui à partir d'une sémantique squelettique peut générer un interpréteur abstrait. Nous présentons l'implémentation de Taiga qui nous permet d'obtenir des interpréteurs abstraits à partir de sémantiques squelettiques, et nous montrons comment il peut être utilisé sur notre petit langage While. Il suffit de fournir des abstractions pour certaines parties du langage, et Taiga est capable de générer un interpréteur abstrait pouvant analyser de vrais programmes du langage.

Nous concluons cette thèse en discutant de ce qui a été accompli, ainsi que des travaux à mener dans le futur. Nous réussissons à obtenir des interprétations abstraites correctes à partir de descriptions sémantiques de langages. Nous avons testé notre approche sur deux

petits langages et nous avons défini deux interprétations abstraites pour ces langages différents : While, un langage impératif, et le $\lambda$-calcul, un langage fonctionnel. Les interprétations abstraites sont très simples à obtenir, car pour une sémantique squelettique donnée, il suffit de définir les abstractions à utiliser pour les parties les plus spécifiques au langage. Nous prouvons simplement que les analyses obtenues sont correctes en prouvant de petits lemmes sur les parties dépendantes du langage, et en utilisant le théorème de correction de l'interprétation abstraite présenté au chapitre 4. Notre générateur d'interpréteur abstrait nous permet même d'obtenir un interpréteur abstrait exécutable pouvant analyser de vrai programmes à partir d'une sémantique squelettique. Néanmoins, ils demeurent de nombreuses pistes d'améliorations. Nous n'avons pas prouvé la terminaison de nos interpréteurs abstraits, qui est une propriété essentielle pour ces programmes. Nos analyses manquent de précisions, car notre approche est indépendante du langage, et nous discutons des pistes pour l'améliorer. Nous n'avons qu'une preuve papier de la correction de nos interpréteurs, or nous souhaitons l'écrire en Coq.

# Table of Contents

Chapter 1

# Context

## 1.1 Introduction

### 1.1.1 Static Analyses and Their Correctness

Many objects we interact with everyday when paying, driving, cooking, and more, involve some code and programs. Programs run many applications, some where the stakes are low like the code running your high-tech coffee machine: a program selects the quantity of beans to grind, and the quantity of water to pump. The worst that can happen in case of a malfunction is to get terrible coffee. However, critical tasks have also been delegated to software where the consequences of a malfunction can be very serious, and dangerous. Some bugs are not life-threatening but are very costly, like the DAO attack [3], where an attacker was able to steal millions of dollars in Ethereum currency. There was a vulnerability in a program managing a crowdfunding platform on the Ethereum blockchain that was exploited to withdraw money. Other bugs may have physical impacts, like the crash of Ariane 5 that stemmed from an overflow error in the autopilot of the rocket that led to the loss of the satellites that were on board. It was considered one of the most expensive bugs in history at the time. Hence, ensuring that a program is indeed doing what it is meant to do became paramount for code running critical applications.

To ensure that software satisfies a property $P$, many techniques have been developed and are used nowadays. The most ubiquitous one is testing: programs are subject to intensive testing to make sure that they work properly. However, tests can be tedious to write and maintain, and they are not exhaustive: a specific bug not covered by tests may never be detected. Tests may seem to be an easy solution but they require lots of work to be useful: they should cover executions that are representative of what may happen in the real world, and the expected result of a test must be correct and therefore written with the utmost care, otherwise the test is useless. Tools have been developed using formal methods to tackle some of the issues of tests. Formal methods is the set of every approach that

use mathematics to ensure that a program respects some properties. Interactive Theorem Provers (ITP) like Coq [7] or Isabelle [47] can be used to prove that a program meets a property $P$ mathematically. Indeed, an ITP is a tool to write mathematical definitions, lemmas and theorems, and proofs that are verified by the ITP. A property that can be expressed and proved mathematically can be proved with a theorem prover, which checks that the proof provided by the user is correct. However, writing proofs is tedious with an ITP and hard to make it automatically. That is an issue Static Analyses try to resolve. A Static Analysis can analyse programs automatically without executing them. It is usually automated and therefore potentially scales on an extensive quantity of code. An automated static analysis can be *safe*: if a program may go bad, the analysis will catch it, but the analysis might not be able to conclude that every correct program is indeed safe. Or, the analysis can be *complete*: every program that satisfies the desired property is accepted by the analyser, but it might flag programs that do not meet $P$ as safe. By Rice Theorem, it is impossible to get safety and completeness for a non-trivial properties with automatic analysis. With formal methods, static analyses can be proved mathematically correct: we can mathematically prove that some analyses have the safety property for instance. This should increase the confidence in these analyses. Abstract Interpretation is a particular technique to define mathematically correct static analyses for languages, as the analysis can be proved *safe*, and therefore will always reject all problematic programs. However, the more expressive an analysis is (meaning it rejects few safe programs), the harder it is to prove the analysis correct. The objective of this thesis is to derive correct safe static analyses from a formal description of languages. We hope that we can reduce the work necessary to design a static analysis by providing a language-agnostic method to get a working, correct abstract interpreter.

### 1.1.2 Static Analyses from Machine Representable Semantics

Given a property $P$ and a program, does the program meet the property? To express property $P$ and to prove that it holds, we need a mathematical definition of the behaviour of the program. This mathematical definition is called the semantics of the program. There are many types of semantics, but the goal is to generate a static analysis **from** the semantics of the language. Hence, we needed to use *mechanised semantics*. We call the *mechanisation* of an object a machine representation of this object. A mechanised semantics is a machine representation of the semantics. Mechanised semantics can offer interesting features, one example is to prove properties with an interactive prover. For

instance, CompCert is a certified C compiler written in Coq, that transforms C programs into assembly code. Given a valid C source program, it is mathematically proved in Coq that the assembly code produced by CompCert has a behaviour that is valid according to the semantics of the source program. To prove this theorem, a mechanisation of the semantics of C and assembly has been written in Coq. The theorem of correctness of the compilation has been written and proved in Coq. Similarly, Verasco, a verified C analyser written in Coq performs static analyses using abstract interpretation. The abstract interpretation is proved safe in Coq: the abstract interpretation is a safe approximation of the semantics of C. These two works are major undertakings and it required many experts and are therefore hard to reproduce. However, Verasco that it was possible to get a correct abstract interpreter proved correct using mechanised semantics. The objective of this thesis is to give a methodology to derive static analyses from a formal description of a language, to get a safe, proven correct static analysis with as little pain as possible. Our approach is to define the semantics of a language and its abstract interpretation from the same object: a skeletal semantics [4]. Skeletal semantics is a framework for writing machine representable semantics of languages.

## 1.2 Semantics of Programs

Our objective in this thesis is to make the development of static analysers, that ensure that programs are *safe*, easier. To formally reason about programs, we need a mathematical description of the behaviour of a program, this description is called *semantics*. This way, a formally defined analysis of a language can be proved correct relative to the semantics of the language. There are many types of semantic formats each one comes with its benefits and drawbacks. We try to give an overview of some semantic formats, and give an intuition of what they are good and bad for.

### 1.2.1 What is a Semantics?

When one writes code in some language, *syntax* is the set of rules of how to represent a *program*: what characters to type to form an expression, a function etc... But what meaning is given to the program? The programmer has an intuition of what its code should do, as they may read the documentation of the language, study some code examples, and make comparisons with other languages. However, all of these resources are usually

```
expr ::= n ∈ lit
       |  x ∈ ident
       |  expr + expr
       |  expr <= expr
stmt ::= skip
       |  stmt ; stmt
       |  x := expr
       |  if expr then stmt else stmt
       |  while expr do stmt
```

Figure 1.1: Grammar of the While Language

imprecise: a documentation is written in natural language which is often ambiguous. Code examples can be misinterpreted or hide subtleties and different languages can differ on syntactically similar programs.

A *semantics* is a mathematical definition of the meaning of a program, removing all ambiguities coming from the natural language. There are many semantic formats, and choosing the relevant one depends on the context. A compiler developer may prefer an *intensional* semantics: a semantics exposing the internal circuitry of the language. A programmer may prefer *extensional* semantics, which only exposes externally observable behaviours. There is not one semantic format that is better in the absolute: it is often necessary to define several semantics for a given language because they have different uses, for different audiences. A brief presentation of some semantic formats is given in this chapter, along with an intuition of what they can be used for.

### 1.2.2 Operational Semantics

The first semantics we present are *operational semantics*. An operational semantics gives meaning to programs by constructing trees from logical statements that describe the executions of every construct of the language. There are several operational semantics and we present the two most notorious ones: the *Structural Operational Semantics*, also known as "small-step semantics", and the *Natural Semantics*, also known as "big-step semantics".

We define a toy imperative language called "While" which will be used to illustrate the different semantic formats. The grammar of While is given on Figure 1.1 and uses four syntactic categories. Literals (`lit`) that we assume to be the set of relative integers.

$$\Downarrow_e \in \mathcal{P}\left((\texttt{store} \times \texttt{expr}) \times \mathbb{Z}\right)$$
$$\Downarrow_s \in \mathcal{P}\left((\texttt{store} \times \texttt{stmt}) \times \texttt{store}\right)$$

$$\frac{}{\sigma, \texttt{n} \Downarrow_e n} \qquad \frac{\sigma(\texttt{x}) = n}{\sigma, \texttt{x} \Downarrow_e n} \qquad \frac{\sigma, \texttt{e1} \Downarrow_e n_1 \qquad \sigma, \texttt{e2} \Downarrow_e n_2}{\sigma, \texttt{e1 + e2} \Downarrow_e n_1 + n_2}$$

$$\frac{\sigma, \texttt{e1} \Downarrow_e n_1 \qquad \sigma, \texttt{e2} \Downarrow_e n_2 \qquad n_1 \leq n_2}{\sigma, \texttt{e1 <= e2} \Downarrow_e 1} \qquad \frac{\sigma, \texttt{e1} \Downarrow_e n_1 \qquad \sigma, \texttt{e2} \Downarrow_e n_2 \qquad n_1 > n_2}{\sigma, \texttt{e1 <= e2} \Downarrow_e 0}$$

$$\frac{}{\sigma, \texttt{skip} \Downarrow_s \sigma} \qquad \frac{\sigma, \texttt{s1} \Downarrow_s \sigma' \qquad \sigma', \texttt{s2} \Downarrow_s \sigma''}{\sigma, \texttt{s1 ; s2} \Downarrow_s \sigma''} \qquad \frac{\sigma, \texttt{e} \Downarrow_e n}{\sigma, \texttt{x := e} \Downarrow_s \sigma[\texttt{x} \leftarrow n]}$$

$$\frac{\sigma, \texttt{e} \Downarrow_e n \qquad n \neq 0 \qquad \sigma, \texttt{s1} \Downarrow_s \sigma'}{\sigma, \texttt{if e then s1 else s2} \Downarrow_s \sigma'} \qquad \frac{\sigma, \texttt{e} \Downarrow_e n \qquad n = 0 \qquad \sigma, \texttt{s2} \Downarrow_s \sigma'}{\sigma, \texttt{if e then s1 else s2} \Downarrow_s \sigma'}$$

$$\frac{\sigma, \texttt{e} \Downarrow_e n \qquad n = 0}{\sigma, \texttt{while e do c} \Downarrow_s \sigma} \qquad \frac{\sigma, \texttt{e} \Downarrow_e n \qquad n \neq 0 \qquad \sigma, \texttt{c} \Downarrow_s \sigma' \qquad \sigma', \texttt{while e do c} \Downarrow_s \sigma''}{\sigma, \texttt{while e do c} \Downarrow_s \sigma''}$$

Figure 1.2: Inference Rules of the Natural Semantics of While

Identifiers (`ident`) that we assume to be a countable set of variables. Expressions (`expr`) that are defined by a BNF grammar to be either a literal, an identifier, an addition of two sub-expressions or a comparison of two sub-expressions. Statements (`stmt`) are defined to be either a skip instruction, an assignment, a conditional branching or a while loop. Moreover, we introduce a set of stores, which are partial functions from identifiers to integers. We use the notation $f : A \hookrightarrow B$ for partial functions from $A$ to $B$.

$$\texttt{store} \equiv \texttt{ident} \hookrightarrow \mathbb{Z}$$

### 1.2.3 Natural Semantics

A Natural Semantics is an operational semantics defined as an inductive relation from a program and an appropriate context to the result of the evaluation of the program. It was first defined by Kahn [19] in 1987 to specify a semantics for mini-ML, a small functional language. The Natural Semantics of While is presented on Figure 1.2 as a set of inference rules, each rule describes how a construct is computed. The relation $\Downarrow_e$ is the evaluation of

expressions and maps a store and an expression to an integer. The relation $\Downarrow_s$ associates a store and a statement to a new store. It is said big-step as it is an input-output relation. We write $\sigma, \texttt{s} \Downarrow_s \sigma'$ when there exists a valid derivation tree with this conclusion. We call a derivation tree a combination of inference rules where the top parts of the tree are all *axioms*: logical statements that do not depend on the relations $\Downarrow_e$ or $\Downarrow_s$. Natural Semantics are convenient because they have the structure of a recursive interpreter, and thus can be helpful for developers to implement an interpreter for the language. They are called "natural" because they describe the evaluation of each construct as one would say in natural language. As an example, to compute an addition, the rule says that the left and right operands must be evaluated, and the output is the sum of the results.

We present each inference rule of Figure 1.2, from left to right, top to bottom. We start with the evaluation of expressions, which given a store and an expression map-to an integer. The first rule is the evaluation of a constant. Note that we use different fonts for the constant: `n` is syntax, and $n$ is an integer and the result of the evaluation of the constant. The evaluation of a variable is equal to the value it is bound to in the store. The evaluation of an addition is the integer addition of the evaluation of both sub-expressions. Here, `+` is syntax, and $+$ is the mathematical operation. There are two rules for the comparison of expressions: they all start by evaluating both sub-expressions. For a comparison, if the left sub-expression is smaller than the right sub-expression, then the condition is true, and 1 is returned: the truth value. Otherwise, it is 0 that is returned, the false value. This concludes the evaluation of expressions, there remains the evaluation of statements, which given a store and a statement, returns a new store. The evaluation of a skip statement returns the store unchanged. The evaluation of a sequence is obtained by evaluating the left statement and using the new store to evaluate the right statement and get the result. There are two rules for if-statements: both start by evaluating the expression, which represents a boolean. If the expression is true (the evaluation of the expression is not 0), the first branch is evaluated, otherwise, the condition is false and the second branch is evaluated. Finally, there are two rules to evaluate a loop, both start with the evaluation of the expression. If the expression evaluates to false, then the same store is returned as the body of the loop is never evaluated. Otherwise, the body of the loop is evaluated one time and then re-evaluated with the new store.

We give a simple example with the program `while x <= 0 do x := x + 1`. We write $\{x_1 \mapsto v_1, .., x_n \mapsto v_n\}$ for the partial function that maps $x_i$ to $v_i$. Let $\sigma_i = \{x \mapsto i\}$, we show that $\sigma_i, \texttt{x := x + 1} \Downarrow_s \sigma_{i+1}$ and we show the evaluation of two expressions on

$$\dfrac{\dfrac{\sigma_i(\texttt{x}) = i}{\sigma_i, \texttt{x} \Downarrow_e i} \qquad \dfrac{}{\sigma_i, \texttt{1} \Downarrow_e 1}}{\dfrac{\sigma_i, \texttt{x + 1} \Downarrow_e i + 1}{\sigma_i, \texttt{x := x + 1} \Downarrow_s \sigma_{i+1}}}$$

$$\dfrac{\dfrac{\sigma(\texttt{x}) = 0}{\sigma_0, \texttt{x} \Downarrow_e 0} \qquad \dfrac{}{\sigma_0, \texttt{1} \Downarrow_e 1} \qquad 0 \leq 0}{\sigma_0, \texttt{x <= 0} \Downarrow_e 1}$$

$$\dfrac{\dfrac{\sigma_1(\texttt{x}) = 1}{\sigma_1, \texttt{x} \Downarrow_e 1} \qquad \dfrac{}{\sigma_1, \texttt{0} \Downarrow_e 0} \qquad 1 > 0}{\sigma_1, \texttt{x <= 0} \Downarrow_e 0}$$

Figure 1.3: Semantics of Fragments of a Program

Figure 1.3 as preliminary results, using the inference rules of Figure 1.2.

Using these results, the derivation of our program is:

$$\dfrac{\dfrac{\dots}{\sigma_0, \texttt{x <= 0} \Downarrow_e 1} \qquad \dfrac{\dots}{\sigma_0, \texttt{x := x + 1} \Downarrow_s \sigma_1} \qquad \dfrac{\dfrac{\dots}{\sigma_1, \texttt{x <= 0} \Downarrow_e 0}}{\sigma_1, \texttt{while x <= 0 do x := x + 1} \Downarrow_s \sigma_1}}{\sigma_0, \texttt{while x <= 0 do x := x + 1} \Downarrow_s \sigma_1}$$

When evaluating the while loop from store $\sigma_0$, the condition `x <= 0` is evaluated as true. The body of the loop `x := x + 1` is evaluated and gives a new store $\sigma_1$: the value of variable `x` has been incremented. The loop is then re-evaluated with the new store $\sigma_1$, but the condition is now false, thus $\sigma_1$ is returned and is the final result of our program.

There are several drawbacks to natural semantics. They make it hard to reason about non-terminating programs. The trees are defined by induction, and have finite height. Therefore, a non-terminating program does not have a derivation. In particular, it is impossible to find a store $\sigma$ such that:

$$\sigma_0, \texttt{while 1 do skip} \Downarrow_s \sigma$$

One consequence is that it is not possible to reason about non-terminating programs with big-step semantics defined by induction.

$$\rightarrow_e \in \mathcal{P}\left((\texttt{store} \times \texttt{expr}) \times (\texttt{store} \times \texttt{expr})\right)$$
$$\rightarrow_s \in \mathcal{P}\left((\texttt{store} \times \texttt{stmt}) \times ((\texttt{store} \times \texttt{stmt}) \cup \texttt{store})\right)$$

$$\frac{}{\langle \sigma, \texttt{n} \rangle \rightarrow_e \langle \sigma, n \rangle} \qquad \frac{}{\langle \sigma, \texttt{x} \rangle \rightarrow_e \langle \sigma, \sigma(\texttt{x}) \rangle} \qquad \frac{\langle \sigma, \texttt{e1} \rangle \rightarrow_e \langle \sigma, \texttt{e1'} \rangle \qquad \texttt{binop} \in \{\texttt{<=}, \texttt{+}\}}{\langle \sigma, \texttt{e1 binop e2} \rangle \rightarrow_e \langle \sigma, \texttt{e1' binop e2} \rangle}$$

$$\frac{\langle \sigma, \texttt{e2} \rangle \rightarrow_e \langle \sigma, \texttt{e2'} \rangle \qquad \texttt{binop} \in \{\texttt{<=}, \texttt{+}\}}{\langle \sigma, n \ \texttt{binop e2} \rangle \rightarrow_e \langle \sigma, n \ \texttt{binop e2'} \rangle} \qquad \frac{n_1 \leq n_2}{\langle \sigma, n_1 \ \texttt{<=} \ n_2 \rangle \rightarrow_e \langle \sigma, 1 \rangle}$$

$$\frac{n_1 > n_2}{\langle \sigma, n_1 \ \texttt{<=} \ n_2 \rangle \rightarrow_e \langle \sigma, 0 \rangle} \qquad \frac{}{\langle \sigma, n_1 \ \texttt{+} \ n_2 \rangle \rightarrow_e \langle \sigma, n_1 + n_2 \rangle}$$

$$\frac{}{\langle \sigma, \texttt{skip} \rangle \rightarrow_s \sigma} \qquad \frac{\langle \sigma, \texttt{s1} \rangle \rightarrow_s \langle \sigma', \texttt{s1'} \rangle}{\langle \sigma, \texttt{s1 ; s2} \rangle \rightarrow_s \langle \sigma', \texttt{s1' ; s2} \rangle} \qquad \frac{\langle \sigma, \texttt{s1} \rangle \rightarrow_s \sigma'}{\langle \sigma, \texttt{s1 ; s2} \rangle \rightarrow_s \langle \sigma', \texttt{s2} \rangle}$$

$$\frac{\langle \sigma, \texttt{e} \rangle \rightarrow_e \texttt{e'}}{\langle \sigma, \texttt{x := e} \rangle \rightarrow_s \langle \sigma', \texttt{x := e'} \rangle} \qquad \frac{}{\langle \sigma, \texttt{x := } n \rangle \rightarrow_s \sigma[\texttt{x} \leftarrow n]}$$

$$\frac{\langle \sigma, \texttt{e} \rangle \rightarrow_e \texttt{e'}}{\langle \sigma, \texttt{if e then s1 else s2} \rangle \rightarrow_s \langle \sigma, \texttt{if e' then s1 else s2} \rangle}$$

$$\frac{n \neq 0}{\langle \sigma, \texttt{if } n \texttt{ then s1 else s2} \rangle \rightarrow_s \langle \sigma, \texttt{s1} \rangle} \qquad \frac{}{\langle \sigma, \texttt{if } 0 \texttt{ then s1 else s2} \rangle \rightarrow_s \langle \sigma, \texttt{s2} \rangle}$$

$$\frac{}{\langle \sigma, \texttt{while e do c} \rangle \rightarrow_s \langle \sigma, \texttt{if e then (c ; while e do c) else skip} \rangle}$$

Figure 1.4: Structural Operational Semantics of While

### 1.2.4 Structural Operational Semantics

A Structural Operational Semantics (SOS) is an operational semantics where the logical statements describe a transition system. A transition system is a relation $\rightarrow$ and a set of states. It said that there is a transition between $p$ and $q$ when $(p, q) \in \rightarrow$. It was first described by Plotkin [40] in 1981. A SOS gives meaning to a program by describing how to evaluate its parts and operates on the syntax of the program: a transition modifies the program. SOS are *intensional* as they describe the internal computations to get a result, rather than an input-output relation. A Structural Operational Semantics is sometimes called "small-step". The small-step semantics of While is presented in Figure 1.4. It is formally defined as two relations, one for expressions, $\rightarrow_e$, and one for statements, $\rightarrow_s$. We call *intermediate states* a pair of a store and an expression, or a pair of a store and a statement. The relation $\rightarrow_e$ maps an intermediate state to a new intermediate state. The relation $\rightarrow_s$ maps an intermediate state to a new intermediate state or a store. Evaluating a literal or a variable is done in one step of computation as the result is immediate. Addition and comparison have similar rules. The first rule of both binary expressions performs one step in the left operand, and the second rule reduces the right operand, supposing the left operand is already an integer. We suppose that an expression can be an integer $n$ (which is different than `n`), therefore the grammar of expressions has slightly changed compared to the definition given earlier. For comparisons, when both operands are reduced to integers if the left operand is smaller than the second operand, the result is the integer 1 (the truth value), and 0 otherwise (the false value). Addition is done when both operands are reduced to integers and is the sum of both integers. Evaluating a *skip* program is immediate as the result is the current store. Evaluating a sequence is done by reducing the left statement. There are two transitions for assignments: the first rule reduces the expression, and the second rule applies when the expression is an integer and reduces to a new store where the variable is mapped to the integer. A step of computation for an if-statement is a step of computation of the condition. If the condition is an integer, then a step of computation is evaluating the first or second branch. We suppose that any non-zero value is the truth value. A while loop is transformed in an equivalent conditional branching.

Given a relation $\mathcal{R} \in \mathcal{P}(S \times S)$, $\mathcal{R}^*$ is its reflexive and transitive closure. Let `loop` = `while x <= 1 do x := x + 1`. Let $\sigma_i = \{\mathtt{x} \mapsto i\}$. One can show that (details on Figure 1.5):

$$\langle \sigma_0, \mathtt{loop} \rangle \rightarrow_s^* \{\mathtt{x} \mapsto 2\}$$

$\langle \sigma_0, \texttt{while x <= 1 do x := 1} \rangle \rightarrow_s$
$\langle \sigma_0, \texttt{if x <= 1 then x := x + 1 ; while x <= 1 do x := x + 1 else skip} \rangle \rightarrow_s$
$\langle \sigma_0, \texttt{if 0 <= 1 then x := x + 1 ; while x <= 1 do x := x + 1 else skip} \rangle \rightarrow_s$
$\langle \sigma_0, \texttt{if 1 then x := x + 1 ; while x <= 1 do x := x + 1 else skip} \rangle \rightarrow_s$
$\langle \sigma_0, \texttt{x := x + 1 ; while x <= 1 do x := x + 1} \rangle \rightarrow_s$
$\langle \sigma_0, \texttt{x := 0 + 1 ; while x <= 1 do x := x + 1} \rangle \rightarrow_s$
$\langle \sigma_0, \texttt{x := 1 ; while x <= 1 do x := x + 1} \rangle \rightarrow_s$
$\langle \sigma_1, \texttt{while x <= 1 do x := x + 1} \rangle \rightarrow_s \cdots \rightarrow_s$
$\langle \sigma_2, \texttt{while x <= 1 do x := x + 1} \rangle \rightarrow_s$
$\langle \sigma_2, \texttt{if x <= 1 then x := x + 1 ; while x <= 1 do x := x + 1 else skip} \rangle \rightarrow_s$
$\langle \sigma_2, \texttt{if 2 <= 1 then x := x + 1 ; while x <= 1 do x := x + 1 else skip} \rangle \rightarrow_s$
$\langle \sigma_2, \texttt{if 0 then x := x + 1 ; while x <= 1 do x := x + 1 else skip} \rangle \rightarrow_s$
$\langle \sigma_2, \texttt{skip} \rangle \rightarrow_s$
$\sigma_2$

Figure 1.5: The Small-Step Semantics of a Simple While Loop

One can reason about non-terminating program with a small-step semantics as it is possible to have an infinite chain of intermediate states. Take the following example where the condition of the loop is always true and the body of the loop does nothing.

$$\langle \sigma, \texttt{while 1 do skip} \rangle \rightarrow_s$$
$$\langle \sigma, \texttt{if 1 then skip ; while 1 do skip else skip} \rangle \rightarrow_s$$
$$\langle \sigma, \texttt{skip ; while 1 do skip} \rangle \rightarrow_s$$
$$\langle \sigma, \texttt{while 1 do skip} \rangle$$

With this program, for any store $\sigma$, from the intermediate state $\langle \sigma, \texttt{while 1 do skip} \rangle$ the same state is reached. Here, it is easy to see that the program is non-terminating because from any intermediate state, the same intermediate state is reached. However, with program `x := 0 ; while 1 do x := x + 1`, from any store the program will never terminate, and never with repeating intermediate states.

A SOS semantics has some drawbacks. First, a SOS derivation is flat and does not follow the structure of the inductive term representing the program. Because a SOS deriva-

tion is a sequence of rewritings of the term, proofs by induction must be done on the size of the chain, which is less convenient. Moreover, in the While language, loops have to be transformed in a conditional branching which is not natural. Moreover, in a SOS derivation, syntax and values are mixed. The term is progressively rewritten until the result is obtained, but the distinction between syntax and semantics is unclear.

### 1.2.5 Control Flow Graph

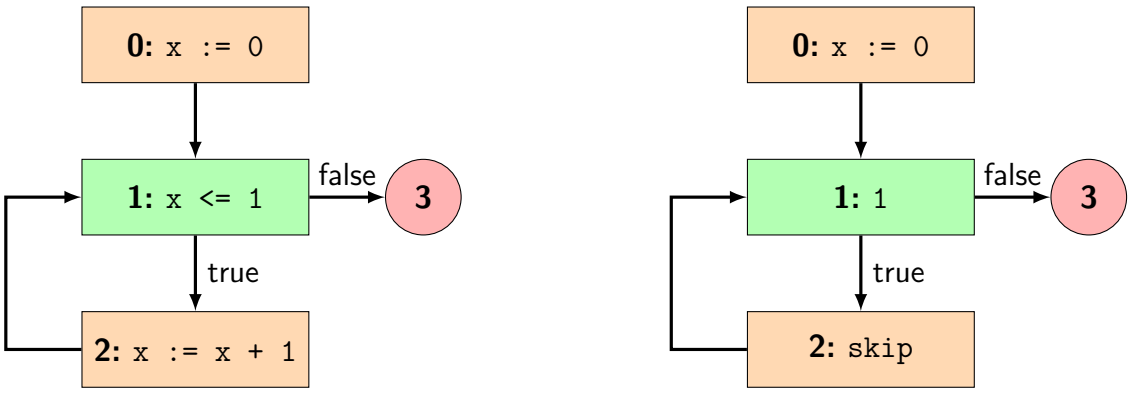
A Control Flow Graph (CFG) is a representation of a program as a directed graph. A graph is a set of nodes and a set of edges: an edge connects two nodes. A directed graph is a graph where edges have a direction, they go from one node to another. A node in a CFG is a pair of a *program point* and an atomic instruction. A program point is a unique identifier, its purpose is to identify precisely a fragment of a program. Program points will be explained in more depth later in this thesis. Here, they are used to uniquely identify each node. We call $\mathbb{P}$ the set of program points. The edges between nodes give the control flow of the program.

**Definition 1** $\mathbb{I}$ *is the set of atomic instructions. It contains assignments and tests, and is formally defined as:*

$$\mathbb{I} = \{\, \texttt{x := e} \mid \texttt{x} \in \texttt{var} \wedge \texttt{e} \in \texttt{expr} \,\} \cup \{\, \texttt{e1 binop e2} \mid \texttt{e1}, \texttt{e2} \in \texttt{expr} \wedge \texttt{binop} \in \{\texttt{+}, \texttt{<=}\} \,\}$$

The Control Flow Graph of `x := 0; while (x <= 1) do x := x + 1` is presented on Figure 1.6a. Each node contains an atomic instruction and is labelled by a unique program point (in bold). The node labelled `3` is special as it is the end of the program and does not contain an atomic instruction. The start of the program is at program point 0, where the variable `x` is initialised. Then, the condition `x <= 1` is tested. If it is true, the next atomic instruction is the incrementation of `x` at program point 2, and then the condition is tested again. If the condition is false, the program exits at program point 3.

Formally, the graph is a tuple $(\text{pp}_{init}, S, A, \text{pp}_{out})$ where $\text{pp}_{init}$ and $\text{pp}_{out}$ are the entry and the exit program points respectively. The set of nodes $S \in \mathcal{P}(\mathbb{P} \times \mathbb{I})$ is a set of pairs of program points and atomic instructions. The set of edges $A \in \mathcal{P}(\mathbb{P} \times \mathbb{P} \times \{\texttt{true}, \texttt{false}, \_\})$ contains the edges, and are labelled. An atomic instruction is either an assignment `x := expr`, or a test `expr1 binop expr2` where `binop` is either a comparison or an addition. For every program point pp, there is at most one atomic instruction $i$ such that $(\text{pp}, i) \in S$. Moreover, if node $(\text{pp}, i) \in S$, and $i$ is a test, then $(\text{pp}, i)$ has two successors: $\text{pp}_t$ and




(a) Incrementation of `x` (b) Infinite loop

Figure 1.6: CFG of a While program

$$\frac{(\text{pp}, \texttt{x := e}) \in S \qquad \sigma, e \Downarrow_e v \qquad \sigma' = \sigma[\texttt{x} \mapsto v] \qquad \text{pp}' = \text{succ}(\text{pp})}{(\text{pp}, \sigma) \rightarrow_{cfg} (\text{pp}', \sigma')} \textsc{Assignment}$$

$$\frac{(\text{pp}, \texttt{e1 binop e2}) \in S \qquad \sigma, \texttt{e1 binop e2} \Downarrow_e v \qquad v \neq 0 \qquad \text{pp}' = \text{succ-true}(\text{pp})}{(\text{pp}, \sigma) \rightarrow_{cfg} (\text{pp}', \sigma)} \textsc{Test-True}$$

$$\frac{(\text{pp}, \texttt{e1 binop e2}) \in S \qquad \sigma, \texttt{e1 binop e2} \Downarrow_e v \qquad v = 0 \qquad \text{pp}' = \text{succ-false}(\text{pp})}{(\text{pp}, \sigma) \rightarrow_{cfg} (\text{pp}', \sigma)} \textsc{Test-False}$$

Figure 1.7: Small-Step Semantics of Control Flow Graph

$\text{pp}_f$ such that $(\text{pp}, \text{pp}_t, \texttt{true}) \in A$ and $(\text{pp}, \text{pp}_f, \texttt{false}) \in A$. We define the notations $\text{succ-true}(\text{pp}, i) \equiv \text{pp}_t$ and $\text{succ-false}(\text{pp}, i) \equiv \text{pp}_f$. If $(\text{pp}, i) \in S$ and $i$ is an assignment, then pp has one successor written $\text{succ}(\text{pp}, i) \equiv \text{pp}'$ such that $(\text{pp}, \text{pp}', \_) \in A$.

We define a semantics for CFGs as a set of transitions. Let $\texttt{state} = \mathbb{P} \times \texttt{store}$. The semantics $\rightarrow_{cfg} \in \texttt{state} \times \texttt{state}$ is defined on Figure 1.7. Because we need a semantics for expressions, we re-used the big-step semantics of expressions. There is one transition for assignments, and two for tests, depending on the result of the expression in the test. Because of the simplicity of atomic instructions, the semantics of CFG is very compact and simple.

**Definition 2** *The set of reachable states for a given CFG* $(\text{pp}_{init}, S, A, \text{pp}_{out})$ *is written*

$[\![(\mathrm{pp}_{init}, S, A, \mathrm{pp}_{out})]\!]_{cfg}$ *and is defined as:*

$$[\![(\mathrm{pp}_{init}, S, A, \mathrm{pp}_{out})]\!]_{cfg} = \left\{ (\mathrm{pp}, \sigma) \;\middle|\; (\mathrm{pp}_{init}, \{\,\}) \rightarrow^*_{cfg} (\mathrm{pp}, \sigma) \right\}$$

*If* `prog` *is a statement, and* $(\mathrm{pp}_{init}, S, A, \mathrm{pp}_{out})$ *its CFG, we write* $[\![\texttt{prog}]\!]_{cfg} \equiv [\![(\mathrm{pp}_{init}, S, A, \mathrm{pp}_{out})]\!]$

We take back the example of Figure 1.6, where
**prog** $\equiv$ `x := 0 ; while x <= 1 do x := x + 1`, and we re-use the notation where $\sigma_i = \{x \mapsto i\}$, the set of reachable states for **prog** is:

$$[\![\mathbf{prog}]\!]_{cfg} = \{(0, \{\}),\ (1, \sigma_0),\ (1, \sigma_1),\ (2, \sigma_1),\ (2, \sigma_2)\,(3, \sigma_2)\}$$

Similarly, the set of reachable states for a non-terminating program is well-defined. Take **prog** $\equiv$ `x := 0 ; while 1 do skip`. The set of reachable states is:

$$[\![\mathbf{prog}]\!]_{cfg} = \{(0, \{\}),\ (1, \sigma_0),\ (2, \sigma_0)\}$$

Because program points 3 is unreachable, there are not reachable states with program point 3.

Writing programs as CFG is not natural, using languages is almost always preferred. However, due to its simplicity, yet powerful expressiveness, CFG are usually parts of the compilation chain of a language compiler. Indeed, CFGs are suited to perform analysis, optimisations and to be compiled to machine code. Moreover, almost every language can be transformed into a CFG, making it a universal representation of programs.

### 1.2.6 Equivalence of the Semantics

The semantic formats seen so far serve different purposes. The natural semantics is easy to write and read and can be used as a reference for an implementation as a recursive interpreter. The small-step semantics can reason about non-terminating programs. The CFG is often used as an intermediate representation in compilers. Because all these semantics can be used for the same language, it is useful to make sure they are equivalent. The semantics we have presented for the While language are all equivalent for terminating programs.

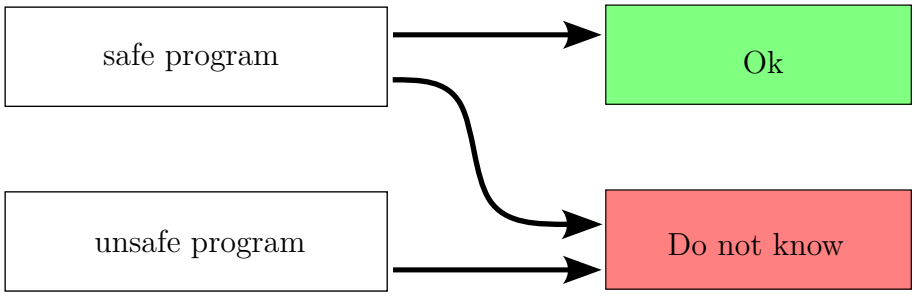



Figure 1.8: Output of a Safe Analysis

**Theorem 1** *Let* $\texttt{prog} \in \texttt{stmt}$*, and* $\sigma \in \texttt{store}$ *and* $(\mathrm{pp}_{init}, S, A, \mathrm{pp}_{out})$ *the CFG of* $\texttt{prog}$*.*

$$\langle \sigma, \texttt{prog} \rangle \rightarrow_s^* \sigma' \iff \sigma, \texttt{prog} \Downarrow_s \sigma' \iff (\mathrm{pp}_{init}, \sigma) \rightarrow_{cfg}^* (\mathrm{pp}_{out}, \sigma')$$

## 1.3 Semantics for Static Analyses

We have presented semantics used to describe the real behaviour of programs. However, these semantics may not be computable or finite: the CFG semantics of the program `while 1 do x := x + 1` is infinite for instance. Some semantics have been designed to analyse programs: the semantics must be computable and finite to be useful. In this thesis, we will focus on Abstract Interpretation, a method to define correct over-approximation of concrete semantics

A Static Analysis of a program is the computation of properties about the program without executing it by some (at least partially) automated tool. Static Analyses can be applied to solve many problems such as detecting some bugs (like division by zero, dereference of null pointers etc...). They can be used to prove that a program does what it is meant to do, for example, that the implementation of a sorting algorithm does indeed sort. Static analyses are usually designed to be fast as one objective of an automated tool is to be able to verify large quantities of code.

In this thesis, we will focus on *safe* static analysis. A safe analysis tries to verify that a property holds in a program and outputs *Ok* when the analysis was able to prove that the property holds, or *Do not know* when the analysis was unable to determine if the property holds. As presented on Figure 1.8, a static analysis may output that a safe program is indeed safe, but if the analysis is not precise enough, it can fail to do so. An unsafe program should systematically be rejected.

### 1.3.1 A pinch of Order Theory

We present some fundamentals of order theory, that will be used to define abstract interpretation in the next sections.

**Definition 3** *A **partial order** $\leq \in \mathcal{P}(L \times L)$ on a set $L$ is a relation that is*

- reflexive*:* $\forall l \in L,\ a \leq a$
- transitive*:* $\forall a, b, c \in L,\ a \leq b \wedge b \leq c \implies a \leq c$
- antisymmetric*:* $\forall a, b \in L,\ a \leq b \wedge b \leq a \implies a = b$

**Definition 4** *A **partially ordered set** (poset) is a pair $(L, \leq)$ of a set $L$ and a partial order $\leq \in \mathcal{P}(L \times L)$.*

**Definition 5** *Let $(L, \leq)$ be a poset. We call an **upper bound** of $a$ and $b$, elements of $L$, an element $c \in L$ such that $a \leq c$ and $b \leq c$. Similarly, a **lower bound** of $a$ and $b$ is an element $c \in L$ such that $c \leq a$ and $c \leq b$.*

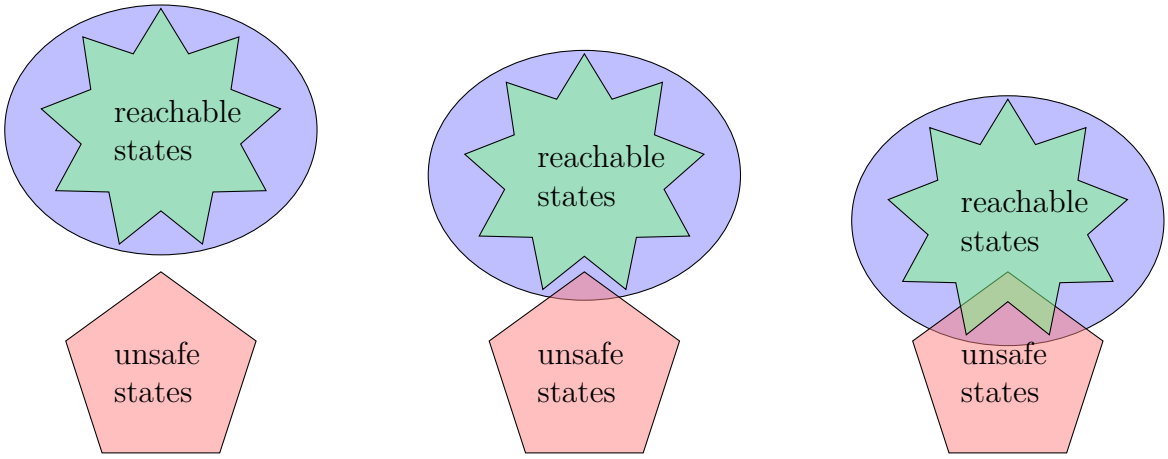
**Definition 6** *The **least upper bound** (lub) of $a$ and $b$ is the least element of $L$ written $a \vee b$ such that $a \leq a \vee b$ and $b \leq a \vee b$. Therefore, if $c$ is an upper bound of $a$ and $b$, then $a \vee b \leq c$.*

*The **greatest lower bound** (glb) of $a$ and $b$ is the greatest element of $L$ written $a \wedge b$ such that $a \wedge b \leq a$ and $a \wedge b \leq b$. Therefore, if $c$ is a lower bound of $a$ and $b$, then $c \leq a \wedge b$.*

**Definition 7** *Let $(L, \leq)$ be a poset. We call $\bot \in L$ and $\top \in L$ the least and greatest elements of the poset if $\forall a \in L,\ \bot \leq a \wedge a \leq \top$.*

*A poset does not necessarily contain a least or greatest element.*

**Definition 8** *A **lattice** $(L, \leq, \wedge, \vee, \bot, \top)$ is a poset with a glb $\wedge$ and a lub $\vee$ for every pairs, a least element $\bot \in L$, and a greatest element $\top \in L$.*




(a) (b) (c)

Figure 1.9: Abstract Interpretation: an Illustration

**Lemma 1** *Let* $(L, \leq, \wedge, \vee, \bot, \top)$ *be a lattice, and* $S$ *a set. Then,* $(S \to L, \leq_\to, \wedge_\to, \vee_\to, \bot_\to, \top_\to)$ *where*

$$
\begin{aligned}
x \leq_\to y &\iff \forall s \in S,\ x(s) \leq y(s) \\
x \vee_\to y &\iff \forall s \in S,\ (x \vee_\to y)(s) = x(s) \vee y(s) \\
x \wedge_\to y &\iff \forall s \in S,\ (x \wedge_\to y)(s) = x(s) \wedge y(s) \\
&\forall s \in S,\ \bot_\to(s) = \bot \\
&\forall s \in S,\ \top_\to(s) = \top
\end{aligned}
$$

*is also a lattice, obtained by point-wise lifting.*

### 1.3.2 Abstract Interpretation

Abstract interpretation [9] is a method of over-approximation of the semantics of programs. An abstract interpretation computes an over-approximation of the *concrete* executions of a program. Figure 1.9 shows the three outcomes of an abstract interpretation. The possible real executions of the program, in green, are complexes, possibly infinite, and therefore not computable. An abstract interpretation computes an over-approximation of the executions, in blue. This over-approximation is simpler, computable, and therefore can be compared to the unsafe states. The latter ones are the states where the desired

property is not verified. In the first example, the set of unsafe states does not intersect the over-approximation. Therefore, one can conclude that the program cannot go in an unsafe state. The examples on Figure 1.9b and Figure 1.9c show when the analysis cannot conclude. In both examples, the over-approximation intersects the set of unsafe states. However, in the second example, the program is safe and cannot go in a state where the property is not verified, while in the third example the program can go bad. Therefore, when the over-approximation intersects the set of unsafe states, the analysis cannot conclude as the program may or may not go bad. An abstract interpretation covers all executions of a program by abstracting the concrete domains. Formally, in our While language, there are two domains: `int` and `store`. We call the concrete domains the definitions of the domains for a concrete execution. We call the abstract domains the definitions of the domains for an abstract interpretation.

**Definition 9** *The concrete domains of While are defined as:*

$$\mathit{int} = \mathbb{Z}$$
$$\mathit{store} = \mathit{ident} \hookrightarrow \mathbb{Z}$$

**Definition 10** *The abstract domain of a domain* $\mathit{type}$ *is denoted with* $\mathit{type}^\sharp$*.*

**Definition 11** *The abstract domain of the* $\mathit{int}$ *type of While is defined as:*

$$\mathit{int}^\sharp = \mathbb{I}$$

*where* $\mathbb{I}$ *is the set of intervals.*

$$\mathbb{I} \equiv \{\, [a;b] \mid a, b \in \mathbb{Z} \cup \{+\infty, -\infty\},\ a \leq b \,\}$$

*An interval* $[a;b]$ *is a pair of two integers, or infinity, that represents the convex set of integers* $\{\, n \mid a \leq n \leq b \,\}$*. Thus, an arbitrarily large set of integers can be represented efficiently.*

We call a *concrete value* an element of a concrete domain, and *abstract value* an element of an abstract domain. An abstract interpretation requires comparisons, greatest lower

bounds and least upper bound for all abstract domains. In abstract interpretation, we call a greatest lower bound a *meet* and a least upper bound a *join* or an *abstract union*. We write $\sqsubseteq_{\texttt{type}}$, $\sqcup_{\texttt{type}}$ and $\sqcap_{\texttt{type}}$ for the comparison, the join and the meet for abstract domain $\texttt{type}^\sharp$. We drop the domain annotation when it is unambiguous. For each domain type, $(\texttt{type}^\sharp, \sqcup_{\texttt{type}}, \sqsubseteq_{\texttt{type}}, \top_{\texttt{type}}, \bot_{\texttt{type}})$ is a lattice.

**Definition 12** *The abstract comparison and abstract union for intervals are defined as:*

$$[i_1; i_2] \sqsubseteq_{int} [j_1; j_2] \iff i_1 \geq j_1 \wedge i_2 \leq j_2$$
$$[i_1; i_2] \sqcup_{int} [j_1; j_2] = [\min(i_1, j_1); \max(i_2, j_2)]$$

*The tuple* $(\boldsymbol{int}^\sharp, \sqsubseteq_{int}, \sqcup_{int}, \emptyset, [-\infty; +\infty])$ *is a lattice, if we suppose* $\emptyset$ *is the empty interval.*

*The tuple* $(\boldsymbol{store}^\sharp, \sqsubseteq_{store}, \sqcup_{store}, \bot_{store}, \top_{store})$ *with* $\boldsymbol{store}^\sharp = \mathcal{X} \to \boldsymbol{int}^\sharp$ *is obtained by point-wise lifting of the lattice of integers is therefore also a lattice by Lemma 1.*

**Definition 13** *A widening operator* $\nabla_{type}$ *is an upper-bound operator with the property that, given any sequence of abstract values* $x_i^\sharp \in \boldsymbol{type}^\sharp$*, and by defining the sequence:*

$$y_0^\sharp = x_0^\sharp$$
$$y_{n+1}^\sharp = x_{n+1}^\sharp \nabla_{type} y_n^\sharp$$

*then the sequence* $y^\sharp$ *converges in finite time:* $\exists n_0, \forall n \geq n_0, y_n^\sharp = y_{n_0}^\sharp$

The widening operator is paramount to ensure the termination of an abstract interpretation.

The concrete and abstract domains are linked by abstraction and concretisation functions. They are used to pass from an abstract domain to a concrete domain, and from an abstract domain to a concrete domain.

**Definition 14** *Let* **type** *by a domain. The abstraction function and concretisation function have type:*

$$\alpha_{type} : \mathcal{P}(\boldsymbol{type}) \to \boldsymbol{type}^\sharp$$
$$\gamma_{type} : \boldsymbol{type}^\sharp \to \mathcal{P}(\boldsymbol{type})$$

$\alpha_{\texttt{type}}$ is the canonic name for an abstraction function and $\gamma_{\texttt{type}}$ is the canonic name for a concretisation function. When it is not ambiguous, we will drop the domain annotation and write $\alpha$ and $\gamma$.

The abstraction function, given a set of concrete values, should give an abstract value that is an over-approximation of the set of concrete values. The concretisation function, given an abstract value, should give the set of concrete values such that the abstract value is a correct approximation of these values.

**Definition 15**

$$\gamma_{int}([i;j]) = \{\, n \mid i \leq n \leq j \,\}$$
$$\gamma_{store}(\sigma^\sharp) = \left\{\, \sigma \;\middle|\; \text{dom}\ \ \sigma = \text{dom}\ \ \sigma^\sharp \wedge \forall x \in \text{dom}\,\sigma,\, \sigma(x) \in \gamma_{int}(\sigma^\sharp(x)) \,\right\}$$

**Definition 16**

$$\alpha_{int}(S) = [\inf S; \sup S]$$
$$\alpha_{store}(\Sigma) = \left\{ x \in \bigcup_{\sigma\in\Sigma} \text{dom}\,\sigma \mapsto \bigsqcup_{\sigma\in\Sigma} \alpha_{int}(\sigma(x)) \right\}$$

We have defined concrete and abstract domains for the While language and functions to go from an abstract domain to a concrete domain and from a concrete domain to an abstract domain. There remains to define the abstract interpretation: that is how to compute a correct approximation of the semantics of a While program using the abstract domain we have defined.

### 1.3.3 Collecting Semantics

A Collecting Semantics defines the set of reachable stores at a given program point. We can easily define it from a CFG. Let `prog` be a while statement and $(\text{pp}_{init}, S, \text{pp}_{out})$ its CFG. The collecting semantics is defined as:

$$\text{collect}(\texttt{prog}) = \left\{\, (\text{pp}, \sigma) \;\middle|\; \sigma_0 \in \texttt{store} \wedge (\text{pp}_{init}, \sigma^0) \rightarrow^*_{cfg} (\text{pp}, \sigma) \,\right\}$$

$$\Downarrow_e^\sharp \in \mathcal{P}\left((\texttt{store}^\sharp \times \texttt{expr}) \times \texttt{int}^\sharp\right)$$
$$\Downarrow_s^\sharp \in \mathcal{P}\left((\texttt{store}^\sharp \times \texttt{stmt}) \times \texttt{store}^\sharp\right)$$

$$\frac{}{\sigma^\sharp, \texttt{n} \Downarrow_e^\sharp [n, n]} \qquad \frac{}{\sigma^\sharp, \texttt{x} \Downarrow_e^\sharp \sigma^\sharp(\texttt{x})} \qquad \frac{\sigma^\sharp, \texttt{e1} \Downarrow_e^\sharp i_1 \qquad \sigma^\sharp, \texttt{e2} \Downarrow_e^\sharp i_2 \qquad i = i_1 +_{\mathbb{I}} i_2}{\sigma^\sharp, \texttt{e1 + e2} \Downarrow_e^\sharp i}$$

$$\frac{}{\sigma^\sharp, \texttt{skip} \Downarrow_s^\sharp \sigma^\sharp} \qquad \frac{\sigma^\sharp, \texttt{s1} \Downarrow_s^\sharp \sigma'^\sharp \qquad \sigma'^\sharp \texttt{s2} \Downarrow_s^\sharp \sigma''^\sharp}{\sigma^\sharp, \texttt{s1 ; s2} \Downarrow_s^\sharp \sigma''^\sharp} \qquad \frac{\sigma^\sharp, \texttt{e} \Downarrow_e^\sharp i}{\sigma^\sharp, \texttt{x := e} \Downarrow_s^\sharp \sigma^\sharp[\texttt{x} \leftarrow i]}$$

$$\frac{\sigma^\sharp, \texttt{e} \Downarrow_e^\sharp i \qquad 0 \notin i \qquad \sigma^\sharp, \texttt{s1} \Downarrow_s^\sharp \sigma'^\sharp}{\sigma^\sharp, \texttt{if e then s1 else s2} \Downarrow_s^\sharp \sigma'^\sharp} \qquad \frac{\sigma^\sharp, \texttt{e} \Downarrow_e^\sharp i \qquad i = [0, 0] \qquad \sigma^\sharp, \texttt{s2} \Downarrow_s^\sharp \sigma'^\sharp}{\sigma^\sharp, \texttt{if e then s1 else s2} \Downarrow_s^\sharp \sigma'^\sharp}$$

$$\frac{\sigma^\sharp, \texttt{e} \Downarrow_e^\sharp i \qquad 0 \in i \qquad n \in i, n \neq 0 \qquad \sigma^\sharp, \texttt{s1} \Downarrow_s^\sharp \sigma'^\sharp \qquad \sigma^\sharp, \texttt{s2} \Downarrow_s^\sharp \sigma''^\sharp}{\sigma^\sharp, \texttt{if e then s1 else s2} \Downarrow_s^\sharp \sigma'^\sharp \sqcup \sigma''^\sharp}$$

$$\frac{\sigma^\sharp, \texttt{e} \Downarrow_e^\sharp i \qquad 0 \notin i \qquad \sigma^\sharp, \texttt{c} \Downarrow_s^\sharp \sigma'^\sharp \qquad \sigma'^\sharp, \texttt{while e do c} \Downarrow_s^\sharp \sigma''^\sharp}{\sigma^\sharp, \texttt{while e do c} \Downarrow_s^\sharp \sigma''^\sharp}$$

$$\frac{\sigma^\sharp, \texttt{e} \Downarrow_e^\sharp i \qquad i = [0, 0]}{\sigma^\sharp, \texttt{while e do c} \Downarrow_s^\sharp \sigma^\sharp}$$

$$\frac{\sigma^\sharp, \texttt{e} \Downarrow_e^\sharp i \qquad 0 \in i \qquad n \in i, n \neq 0 \qquad \sigma^\sharp, \texttt{c} \Downarrow_s^\sharp \sigma'^\sharp \qquad \sigma'^\sharp, \texttt{while e do c} \Downarrow_s^\sharp \sigma''^\sharp}{\sigma^\sharp, \texttt{while e do c} \Downarrow_s^\sharp \sigma''^\sharp \sqcup \sigma^\sharp}$$

Figure 1.10: Rules of Big-step Abstract Interpretation of While

### 1.3.4 Big-Step Abstract Interpretation

Abstract Interpretation was originally defined on structural operational semantics, then for denotational semantics. Schmidt [44] introduced an abstract interpretation from a big-step semantics for $\lambda$-calculus. We present a big-step abstract interpretation for While on Figure 1.10 inspired by the method of Schmidt. It is similar to the big-step semantics of While, but the result of an expression is an interval, and an abstract store is a partial function from variables to intervals. Moreover, there are three rules for if-statement (there are only two rules in the big-step semantics). Indeed, when the evaluation of the condition returns an ambiguous result, both branches must be evaluated, and their results are joined.

Similarly, for a while loop, when the evaluation of the condition gives a result that can be interpreted as true and false, then the result is the abstract union of the current store and the store obtained by evaluating the loop.

Schmidt proposes two mechanisms to prevent infinite computation. First, when there is a repeating node of the form: $\dfrac{\dots}{\sigma^\sharp, \mathtt{s} \Downarrow_s^\sharp ?}$, then we have reached a fixpoint, and the result is $\sigma^\sharp$. However, an infinite computation may happen without a repeating node. Schmidt forces repeating nodes but modifying the derivation tree such that it is not well-formed according to the rules of Figure 1.10 but that is still a correct over-approximation of the big-step semantics of While. Program points are added to statements to differentiate syntactically identical statements but that are different parts of the program. When the abstract interpretation of a fragment of the program depends on itself, the abstract store on the recursive call is widened with the previous store. The derivation $\dfrac{\dots}{\sigma^\sharp, \mathtt{s}^{\mathrm{pp}} \Downarrow_s^\sharp ?}$ becomes $\dfrac{\dots}{\sigma^\sharp, \mathtt{s}^{\mathrm{pp}} \Downarrow_s^\sharp ?}$. By the property of the widening operator, it eventually forces a repeating node and therefore ensures termination.

### 1.3.5 Abstract Interpretation from CFG

We present an abstract interpretation close to the one defined originally by Cousot and Cousot [9]. Our starting point is the CFG of the While program of Figure 1.6a. The idea is to define a set of equations from the CFG, each solution is an abstract store for each program point. The equations of the While program are:

$$
\begin{aligned}
X_0^\sharp &= \bot_{\mathtt{store}} \\
X_1^\sharp &= [\![\texttt{x := 0}]\!]_{cfg}^\sharp(X_0^\sharp) \sqcup_{\mathtt{store}} [\![\texttt{x := x + 1}]\!]_{cfg}^\sharp(X_2^\sharp) \\
X_2^\sharp &= [\![\texttt{x <= 1}]\!]_{cfg}^\sharp(X_1^\sharp) \\
X_3^\sharp &= [\![\texttt{x > 1}]\!]_{cfg}^\sharp(X_1^\sharp)
\end{aligned}
$$

Each $X_i^\sharp$ is an abstract store. The abstract store $X_0^\sharp$ corresponding to the entry point of the program is always $\bot_{\texttt{store}}$ as the store is undefined at this point. For other nodes $X_i^\sharp$, their equations are defined as functions of their predecessors. The equations derived from the CFG are not dependent on the abstraction of integers, but not the semantics of atomic instructions is:

$$\begin{aligned}
[\![\texttt{e}]\!]^\sharp_{cfg}(\sigma^\sharp) &= i && \sigma^\sharp, \texttt{e} \Downarrow^\sharp_e i \\
[\![\texttt{x := e}]\!]^\sharp_{cfg}(\sigma^\sharp) &= \sigma^\sharp[\texttt{x} \leftarrow [\![e]\!]^\sharp_{cfg}(\sigma^\sharp)] \\
[\![\texttt{x <= e}]\!]^\sharp_{cfg}(\sigma^\sharp) &= \sigma^\sharp[\texttt{x} \leftarrow [-\infty; \sup\ i_e] \sqcap^\sharp i_e] && i_e = [\![\texttt{e}]\!]^\sharp_{cfg}(\sigma^\sharp) \\
[\![\texttt{x > e}]\!]^\sharp_{cfg}(\sigma^\sharp) &= \sigma^\sharp[\texttt{x} \leftarrow [\inf\ i_2; +\infty] \sqcap^\sharp i_e] && i_e = [\![\texttt{e}]\!]^\sharp_{cfg}(\sigma^\sharp)
\end{aligned}$$

We re-use the big-step abstract interpretation of expressions to evaluate expressions not to redefine the semantics of expressions. The semantics of an assignment give a new abstract store where `x` maps-to the evaluation of the expression `e`. The semantics of tests is more complex, thus we restrict the tests to be of form `x <= e` or `x > e` here. The semantics of test `x <= e` with abstract store $\sigma^\sharp$ is a new abstract store similar to $\sigma^\sharp$ where the condition `x <= e` holds, thus where `x` is bound-to the intersection of $\sigma^\sharp(\texttt{x})$ and $[-\infty; \sup[\![e]\!]^\sharp_{cfg}(\sigma^\sharp)]$. The abstract interpretation of this program is the solution of the equations:

$$\begin{aligned}
X_0^\sharp &= \bot_{\texttt{store}} \\
X_1^\sharp &= \{x \mapsto [0; 2]\} \\
X_2^\sharp &= \{x \mapsto [0; 2]\} \\
X_3^\sharp &= \{x \mapsto [2; 2]\}
\end{aligned}$$

Each $X_i^\sharp$ is an abstract store that approximates concrete store at program points $i$ in the CFG. Supposing that the abstract semantics of expression is correct, it comes that

$$(\text{pp}, \sigma) \in \text{collect}(\texttt{prog}) \implies \sigma \in \gamma_{\texttt{store}}(X^\sharp_{\text{pp}})$$

## 1.4 Machine Representable Semantics

The semantics that have been presented here were all *pen-and-paper* semantics. However, the formalisation on paper is error-prone if not done carefully. To mitigate the issue, a

```
type expr =
  | Lit of int
  | Var of string
  | Plus of expr * expr
  | Leq of expr * expr

type stmt =
  | Skip
  | Seq of stmt * stmt
  | Assign of string * expr
  | IfThenElse of expr * stmt * stmt
  | While of expr * stmt

type store = (string * int) list

let rec eval_stmt (s: store) (c: stmt): store =
  match c with
  | While (e, c') ->
     let n = eval_expr s e in
     if n <> 0 then
       let s' = eval_stmt s c' in
       eval_stmt s' (While (e, c'))
     else
       s
```

...

Figure 1.11: Code snippets of an OCaml interpreter for While

semantics can be written with the help of a computer that ensures some properties on the semantics: for instance that there are no issues of "typechecking". We call a Mechanised Semantics a machine-representable semantics. This broad definition encompasses many types of semantics. In this section, we try to present diverse methods currently used to write mechanised semantics and show what are their limitations or benefits.

### 1.4.1 Interpreter in a Functional Language

We present the code of a recursive interpreter coded in a functional language that will serve as our first mechanized semantics. We chose the *OCaml* language as it is well-suited to write interpreters. A snippet of the code of the interpreter is presented on Figure 1.11. First, there are two type definitions: `expr` and `stmt` that are Algebraic Data Types (ADT)

defining expressions and statements. Their definitions are close to the grammar given previously of the While language. Then, the `store` type is defined as an association list between variables and integers. We chose this common definition to represent partial functions. Two recursive functions are defined to evaluate expressions and statements: `eval_expr` and `eval_stmt`. Only the start of `eval_stmt` is presented here and shows the evaluation of a loop. The functions are very close to the big-step semantics seen previously because both the interpreter and the big-step semantics are defined inductively. To evaluate a while loop, the expression is evaluated. Then if the result is not zero, the condition is true, we evaluate the body of the loop. It gives a new store, and the loop is re-entered with the new store. If the condition is false, the current store is returned.

The main benefit of this mechanisation is that it can be executed and is easy to write. However, programming languages are not designed to mechanise semantics and make it hard to work with this representation. First, one would need to re-use the OCaml parser to get a mechanised representation of the semantics to perform operations on it. For instance, it is unclear how one would prove properties of the language with this representation, especially because the OCaml abstract syntax tree is very large, thus OCaml representation of our semantics is probably very complex. Also, the semantics of OCaml is not formally defined, thus so is our mechanised semantics.

### 1.4.2 Coq

Coq [7] is a proof assistant that comes with a functional language, the ability to write propositions, and a tactic language to write proofs that are verified by Coq. The language of Coq is a functional language, it allows the definition of Algebraic Data Types, recursive functions, definitions of relations and more. It is possible to define the small-step, or big-step, semantics of previous sections. A fragment of a Coq definition of a big-step semantics is presented on Figure 1.12. However, the semantics is defined as a relation, or `Prop` in Coq, and there are no simple methods to extract an executable from a relation. Functions are executable, but Coq must be able to determine that they are terminating. When writing a recursive function, Coq must be able to guarantee that it ends. It is possible to write an interpreter in Coq like the interpreter for While seen previously. However, one must add guarantees that the interpretation function is terminating by specifying a decreasing quantity at each recursive call. Therefore in Coq, one can define an interpreter, and then state and prove properties about it. It can then be exported to OCaml code and compiled to get an executable interpreter. The expressiveness of Coq allows the definitions

```
Inductive expr : Type :=
| Lit: nat -> expr
| Ident: var -> expr
| Plus: expr -> expr -> expr
| Leq: expr -> expr -> expr.

Inductive stmt : Type :=
| Skip: stmt
| Seq: stmt -> stmt -> stmt
| Assign: var -> expr -> stmt
| IfThenElse
  : expr -> stmt -> stmt -> stmt
| While: expr -> stmt -> stmt.

Inductive eval_stmt : store -> stmt -> store -> Prop :=
| eval_stmt_while_true :
  forall (σ σ' σ'': store) (e: expr) (st: stmt) (n: nat),
    eval_expr σ e n ->
    n <> 0 ->
    eval_stmt σ st σ' ->
    eval_stmt σ' (While e st) σ'' ->
    eval_stmt σ (While e st) σ''
| eval_stmt_while_false:
  forall (σ: store) (e: expr) (st: stmt),
    eval_expr σ e 0 ->
    eval_stmt σ (While e st) σ
```

Figure 1.12: Big-step semantics in Coq

of semantics in many formats, like big-step semantics, small-step semantics, denotational semantics, and even co-inductive semantics. This complexity makes it hard to generically extract an executable analyser from a Coq description of the language.

## 1.5 Mechanised Semantics of Languages

### 1.5.1 The $\mathbb{K}$ Framework

The $\mathbb{K}$ framework [42] provides tools to write mechanized semantics that are executable. A mechanization in $\mathbb{K}$ consists of the definition of the syntax of the language, and rewriting rules on that syntax. We present a mechanisation of the While language in $\mathbb{K}$. The definition of the syntax of the language is done by specifying BNF grammar for expressions and statements.

```
syntax Expr  ::= Int | Id
               | Expr "+" Expr
               | Exp "<=" Expr
syntax Stmt  ::= Id ":=" Expr ";"
               | "if"  Expr
                 "then" Stmt "else" Stmt
               | "while"  Expr "do" Stmt
               | Stmt Stmt
```

`Int` and `Id` are primitive syntactic categories for integers and identifiers.

In $\mathbb{K}$, the state of the computation is called a *configuration*. Here we give the start configuration for our While language.

```
configuration <T>
                <k> $PGM:Stmt </k>
                <store> .Map </store>
              </T>
```

A configuration is an XML-like value. The outer tag `T` is mandatory and contains the whole configuration. The tag containing the program is often written `k`. The meta-variable `PGM` stands for the initial program passed to the interpreter or compiler, and its type is `Stmt`. The tag `store` contains a map ``.''. In $\mathbb{K}$, . stands for empty. Thus `.Map` is the empty map. A configuration is a pair of the current program and a store, similar to the

states in SOS 1.2.4. The start configuration contains the program and the empty store, represented as an empty map.

Finally, there remains to write rewriting rules to define the semantics of the While language.

```
// Expr
  rule <k> X:Id => I ...</k> <store>... X |-> I ...</store>
  rule I1 + I2 => I1 +Int I2
  rule I1 <= I2 => I1 <=Int I2
// Stmt
  rule <k> X := I:Int; => . ...</k> <store>... X |-> (_ => I) ...</store>
  rule S1:Stmt S2:Stmt => S1 ~> S2
  rule if (I) then  S else _ => S  requires I =/=Int 0
  rule if (I) then _ else S => S  require I ==Int 0
  rule while (E) S => if (E) {S while (E) S} else {}
```

There are three rules for expressions and five for statements. The `=>` arrow specify the rewriting. The left-hand side of the arrow matches what should be rewritten, and the right-hand side in what it is rewritten. The arrow is local, a rewriting rule applies to a configuration, but often only a small part of the actual configuration is modified. In the first rewriting rule of expressions, only the tag `k` is modified. This rule means that the content of the tag `k`, which is a variable, is replaced by `I`. Indeed, the store tag contains the value `... X |-> I ...` which is any store where `X` maps to `I`. The remaining rules of the evaluation of expressions also make use of the locality of the arrow, only expressions are rewritten, and the configuration is not explicitly mentioned.

$\mathbb{K}$ has been designed and used to formalise programming languages, like C, Java or functional languages. The $\mathbb{K}$ framework is designed to mechanise semantics and thus comes with a set of tools, like a compiler, an interpreter or a prover able to prove some properties about the semantics by writing *claims* in the semantics and by discharging the proofs to an SMT solver. However, the semantics using rewriting rules is close to the small-step semantics and suffers from the same issues: in particular the values and terms are mixed. Moreover, it is unclear how one would do fully automatic static analysis, like abstract interpretation from rewriting rules as is done with $\mathbb{K}$.

### 1.5.2 Lem

Lem [31] is a semantic definition language. It was originally built as a pure subset of OCaml. It supports recursive function definitions, recursive Algebraic Data Types, polymorphism, and a standard library. Moreover, there are added logical constructs, like universal and existential quantification, inductive relations, lemmas, sets and more We present a formalisation of our imperative language.

```
open import Pervasives
open import List

type expr =
  | Lit of int
  | Var of string
  | Plus of expr * expr
  | Leq of expr * expr

type stmt =
  | Skip
  | Seq of stmt * stmt
  | Assign of string * expr
  | IfThenElse of expr * stmt * stmt
  | While of expr * stmt

type store = list (string * int)

let rec eval_expr (s: store) (e: expr): int =
  match e with
  | Lit n -> n
  | Var x ->
    match List.lookup x s with
    | Nothing -> 0
    | Just n -> n
    end
  | Plus e1 e2 ->
```

```
    let n1 = eval_expr s e1 in
    let n2 = eval_expr s e2 in
    n1 + n2
  | Leq e1 e2 ->
    let n1 = eval_expr s e1 in
    let n2 = eval_expr s e2 in
    if n1 <= n2 then 1 else 0
  end

lemma expr_are_positive :
  (forall s e. List.all (fun (_, n) -> n >= 0) s --> eval_expr s e >= 0)

let rec eval_stmt (s: store) (c: stmt): store =
  match c with
  | While (e, c') ->
    let n = eval_expr s e in
    if n <> 0 then
      let s' = eval_stmt s c' in
      eval_stmt s' (While (e, c'))
    else
      s
  ...
  end
```

The Lem formalisation is very similar to the OCaml one. The most notable difference is the possibility of writing logical formulae. A lemma is stated at the end of the Lem formalisation that says if a store maps every variable to a non-negative integer, then the evaluation on any expression returns a non-negative integer. Lem is not a proof assistant, therefore the lemma can be stated but not proven in Lem. A Lem description can be exported to multiple formats: Coq, HOL/Isabelle, OCaml and Latex. The Lem developers have emphasized generating idiomatic code for each target. To do that, they chose to constrain the expressiveness of the Lem language. The objective is that for each targeted format, to get a human-readable file where the structure of the Lem description is preserved (comments, definitions etc...) and that produces idiomatic code for that targeted format.

Lem supports many features: inductive definitions, relations, functions etc... Also, it comes with native definitions of sets, integers and more. The language is therefore complex, and the definition of a new back-end producing an analyser requires a lot more work.

## 1.6 The Objectives of this Thesis

In this thesis, our goal is to derive correct static analyses from formal descriptions of languages. We see several advantages to this approach. First, it is *language agnostic*: once we choose a language description framework, our approach aims to derive a static analyser for any language description. Most of the work is done once, at the semantic framework level. Second, it should make the definition of analyses easier. Indeed, we start with a formal language description, from which the core infrastructure of the abstract analyser can be automatically built. Then, the remaining work is to provide the abstractions specific to the language that the analysis will use. Finally, we want a methodology that makes it easy to prove the correctness of the analysis. As the core of the analyser is derived from the description of the language, so should most of the proof of correctness.

To achieve these objectives, we use Skeletal Semantics [4] as our language description framework. Skeletal semantics are a powerful method to write semantic descriptions of languages. At its core, there is Skel, a small functional language, used to write *skeletal semantics*: a formal description of a language. Skel is expressive, making it possible to describe complex languages, like the work in progress JSkel [22], the skeletal semantics of JavaScript. The simplicity of Skel makes the derivation of semantics from a skeletal semantics easy. From the skeletal semantics of a language, one can already derive a big-step semantics. One only needs to provide some small language- specific definitions to get it working. By following a similar approach, we want to derive abstract interpretations from skeletal semantics of languages. Abstract Interpretation is a powerful method to define correct static analyses. Moreover, deriving a big-step semantics and an abstract interpretation from the same object, a skeletal semantics, makes the proof that the abstract interpretation is an over-approximation of the big-step semantics of the language easier.

The remainder of this document is structured as follows.

- Chapter 2 presents the Skeletal Semantics Framework: a method to formally describe a programming language semantics. We show how a big-step semantics for a language can be obtained from its skeletal semantics, and we illustrate it with an example on a small imperative language: While.

- Chapter 3 shows our first contribution. We added support for program points to the big-step semantics. Program points are essential to define abstract interpretations.
- Chapter 4 is the main contribution: a methodology to define abstract interpreters from skeletal semantics. We give a step-by-step explanation to get an abstract interpreter from the skeletal semantics of While. This method should work for almost any skeletal semantics. We also prove the abstract interpreter is correct by only proving small language-dependent lemmas.
- Chapter 5 is another example of deriving an abstract interpreter with the same methodology, but for a small functional language: $\lambda$-calculus.
- Chapter 6 is an overview of the implementation of the abstract interpreter generator. This software generates executable abstract interpreters, by following the methodology described in Chapter 4. We show how to it works by generating an abstract interpreter for the While language.

We presented this work in the Express/SOS workshop [16], specifically the design of an abstract interpreter for a toy imperative language from its skeletal semantic. We also presented how to derive a CFA analyser for $\lambda$-calculus from its skeletal semantics at JFLA [15].

Chapter 2

# Skeletal Semantics

Skeletal Semantics [4] is a proposition for machine-representable semantics. At its core there is **Skel** [37], a semantic functional language to describe the behaviour of languages. A Skeletal Semantics is a description of a language written in Skel. The Necro [37] ecosystem is a set of libraries and tools to manipulate skeletal semantics. From a skeletal semantics, the Necro Ocaml Generator generates an OCaml interpreter, Necro Coq generates a Coq formalisation, Necro Debug generates a step-by-step debugger etc...

In this Chapter, we present the language Skel, and how to describe a small imperative language called *While* by writing its skeletal semantics in Skel. Then, we define a natural semantics of Skel, and we show how by meta-interpretation of Skel one can define a big-step semantics for While.

## 2.1 The Skel language

Skel is a minimalist functional language, close to $\lambda$-calculus in Administrative Normal Form, meaning each argument of a function call must be trivial. It supports Algebraic Data Types, recursive definitions and higher-order. The grammar of Skel is presented on Figure 2.1. Skel is defined in more depth in [38] and supports polymorphism. However, our abstract interpretation does not support polymorphism, therefore we use a simply type version of Skel, also defined in [38]. A trivial argument is a *term*. Trivial means the evaluation, defined later in the Section, is immediate. A term can be a variable from a countable set Var, a constructor applied to a term, a tuple or a function $\lambda p : \tau \to S$ where $p$ is a pattern that serves as parameter and $S$ is the body of the function. A skeleton describes a *computation*, which can be a term, an unary application, a let-binding, a branching or a pattern matching. Branchings are a unique trait of Skel, they are used as a non-deterministic choice between several options. Their use will be illustrated later with examples. A pattern, used in let-binding, as parameter of functions, and in pattern-matching, can be a variable, a wildcard, a constructor applied to a pattern or

$$
\begin{array}{rrcl}
\textsc{Term} & t & ::= & x \in \textsc{Var} \mid C\ t \mid (t, .., t) \mid \lambda p : \tau \to S \\
\textsc{Skeleton} & S & ::= & t \mid t\ t \mid \textbf{let}\ p = S\ \textbf{in}\ S \mid \textbf{branch}\ S \,\text{or}\, .. \,\text{or}\, S\ \textbf{end} \\
 & & & \mid \textbf{match}\ t\ \textbf{with}\ p \to S .. p \to S\ \textbf{end} \\
\textsc{Pattern} & p & ::= & x \mid \_ \mid C(p) \mid (p, .., p) \\
\textsc{Type} & \tau & ::= & b \mid \tau \to \tau \mid (\tau, .., \tau) \\
\textsc{Type decl} & r_\tau & ::= & \textbf{type}\ b \mid \textbf{type}\ b = \text{"|"}\ C_1\ \tau_1 .. \text{"|"}\ C_n\ \tau_n \\
\textsc{Term decl} & r_t & ::= & \textbf{val}\ x : \tau \mid \textbf{val}\ x : \tau = t \\
\textsc{Skeletal Semantics} & \mathbb{S} & ::= & (r_t | r_\tau)^*
\end{array}
$$

Figure 2.1: Grammar of Skel and Skeletal Semantics

a tuple of patterns. A type can be a user-defined base type defined at top-level in a skeletal semantics, an arrow type or a tuple type. A type declaration is a name and an optional definition. A type declaration without a definition is said *unspecified*, and with a definition *specified*. Leaving some types unspecified is convenient not to force a restrictive definition. It is then possible to instantiate the unspecified types in different manners depending on the semantics that is defined. The definition of a specified type is an ADT. Similarly, there are term declarations with a name and a type and they can be specified or unspecified. Unspecified terms are often used to define functions that manipulate values with unspecified types, and therefore cannot be given a definition before the unspecified types are themselves given a definition. A Skeletal Semantics is a list of type and term declarations.

### 2.1.1 Skeletal Semantics of While

We present the Skeletal Semantics of While, a small imperative language, on Figure 2.2. We call the language described by the skeletal semantics, here While, the *object* language, to differentiate it from Skel. It starts with four declarations of unspecified types, meaning that these types do not have a definition for now. The `ident` type is the type of identifier for variables. The `lit` type is the type of literals in our programs. It is expected to be some subset of relative integers, as $\mathbb{Z}$ or the set of 32-bit signed integers. One benefit of having non-specified types is that their instantiations can be chosen depending on the context. The `store` type denotes a data structure binding identifiers to integers. The `int` type stands for integers.

There are two specified types declarations. The `expr` ADT defines the expressions of our language and contains four constructors. The `Const` constructor denotes the constants. The `Var` constructor defines the variables of the While language, not to confuse with the variables of Skel. The `Plus` constructor defines additions of two `expr`. The `Leq` constructor is the comparison of two `expr`. Finally, the `Rand` constructor defines a random expression, which denotes a random value within the bounds given as parameters. The `stmt` ADT defines the different statements of our program. The `Skip` constructor is the program that does nothing. The `Assign` constructor is the program that assigns the result of the evaluation of an expression to an identifier. The `Seq` is the sequence of two statements. The `If` constructor is the conditional branching, or "If Then Else" construct. Finally, the `While` constructor is the loop.

There are eight non-specified terms, that are all functions. As expected, all these functions have non-specified types as parameters or results. The `litToInt` function converts a literal into an integer. The `add` function sums two integers, and `lt` compares two integers. The `rand` function takes two literals and returns an integer that should be within the bounds passed as arguments. The `read` and `write` terms are functions to access the value bound to a variable in a store, and to add a new binding in a store respectively. Finally, there are two functions `isZero` and `isNotZero` that take an integer as parameter and return an empty tuple. We will detail their intended purpose later in this Section.

The most interesting part of the semantics are the two definitions of specified terms, which are functions. We detail the `eval_expr` function. The first line is syntactic sugar for

```
val eval_expr: (store, expr) -> int = λ (s, e): (store, expr) ->
```

`eval_expr` takes a tuple of a store and an expression as parameter and returns an integer. A pattern-matching is used to treat each expression. If it is a constant, the literal is converted to an integer using `litToInt`. If it is a variable, its value is fetched in the store using `read`. If it is an addition, both sub-expressions are evaluated and the result is returned using `add`. The comparison of expressions is similar. Finally, the evaluation of a random expression is the result of the `rand` function. The `eval_stmt` function takes a tuple of a store and a statement and returns a new store. The structure of the function is similar to `eval_expr` and makes a case analysis of the statement. The interesting cases are the last two, the conditional branching and the loop. To compute a conditional branching, the expression is evaluated. Then, there is a branching with two branches. The first branch corresponds to the case where the evaluation is true, considering that

```
type ident
type lit
type store
type int

type expr =
| Const lit
| Var ident
| Plus (expr, expr)
| Leq(expr, expr)
| Rand (lit, lit)

type stmt =
| Skip
| Assign (ident, expr)
| Seq (stmt, stmt)
| If (expr, stmt, stmt)
| While (expr, stmt)

val litToInt : lit → int
val add : (int,  int) → int
val lt: (int,  int) → int
val rand : (lit, lit) → int
val read : (ident, store) → int
val write : (ident, store, int) → store
val isZero: int → ()
val isNotZero: int → ()

val eval_expr ((s, e): (store, expr)): int =
  match e with
  | Const i → litToInt i
  | Var x → read (x, s)
  | Plus (e1, e2) →
     let v1 = eval_expr (s, e1) in
     let v2 = eval_expr (s, e2) in
     add (v1, v2)
  | Leq (e1, e2) →
     let v1 = eval_expr (s, e1) in
     let v2 = eval_expr (s, e2) in
     lt (v1, v2)
  | Rand (i1, i2) → rand (i1, i2)
  end
```

Figure 2.2: Skeletal Semantics of the While language

```
val eval_stmt ((s, t): (store, stmt)): store  =
  match t with
  | Skip → s
  | Assign (x, e) →
     let w = eval_expr (s, e) in
     write (x, s, w)
  | Seq (t1, t2) →
     let s' = eval_stmt (s, t1) in
     eval_stmt (s', t2)
  | If (cond, true, false) →
     let i = eval_expr (s, cond) in
     branch
       let () = isNotZero i in
       eval_stmt (s, true)
     or
       let () = isZero i in
       eval_stmt (s, false)
     end
   | While (cond, t') →
      let i = eval_expr (s, cond) in
      branch
        let () = isNotZero i in
        let s' = eval_stmt (s, t') in
        eval_stmt (s', t)
      or
        let () = isZero i in s
      end
  end
```

Figure 2.2: Skeletal Semantics of the While language

$$\Gamma \in \textsc{TypeEnv} = \textsc{Var} \hookrightarrow \textsc{Type}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ \textsc{Var}$$

$$\frac{\mathbf{val}\ x : \tau[= t] \in \mathbb{S}}{\Gamma \vdash x : \tau}\ \textsc{TermDef} \qquad \frac{\Gamma \vdash t : \tau \qquad \mathrm{C} : (\tau, \tau')}{\Gamma \vdash \mathrm{C}\, t : \tau'}\ \textsc{Const}$$

$$\frac{\forall i, \Gamma \vdash t_i : \tau_i}{\Gamma \vdash (t_1, \ldots, t_n) : (\tau_1, \ldots, \tau_n)}\ \textsc{Tuple} \qquad \frac{\Gamma + p \mapsto \tau \vdash S : \tau'}{\Gamma \vdash (\lambda p : \tau \to S) : \tau \to \tau'}\ \textsc{Fun}$$

$$\frac{\Gamma \vdash S_1 : \tau \qquad \ldots \qquad \Gamma \vdash S_n : \tau}{\Gamma \vdash (S_1 \ldots S_n) : \tau}\ \textsc{Branch} \qquad \frac{\Gamma \vdash S : \tau \qquad \Gamma + p \mapsto \tau \vdash S' : \tau'}{\Gamma \vdash \mathrm{let}\, p = S\, \mathrm{in}\, S' : \tau'}\ \textsc{LetIn}$$

$$\frac{\Gamma \vdash t_0 : \tau_1 \to \tau_2 \qquad \Gamma \vdash t_1 : \tau_1}{\Gamma \vdash (t_0\ t_1) : \tau_2}\ \textsc{App}$$

Figure 2.3: Typing Rules of Skel

all non-zero integers denote the truth value. The branch is guarded by `isNotZero i`. This function should act as a filter and be undefined on input `0`. Therefore, the branch cannot be taken if the condition is false. The second branch has a similar structure and is taken when the condition is false. Finally, to evaluate a while loop, the expression is first evaluated. If the condition is true, the body of the loop is evaluated and the loop is re-entered. If the condition is false, the second branch is taken and outputs the store unchanged. To obtain a semantics for While from its Skeletal Semantics, there remains to instantiate the unspecified types and terms and provide an interpretation of Skel. By meta-interpretation of Skel, one will be able to evaluate a While program.

### 2.1.2 Typing Rules of Skel

Skel is a strongly typed language, and the typing rules are given on Figure 2.3. Given two sets $F$ and $G$, $F \hookrightarrow G$ is the set of partial functions with domain $F$ and co-domain $G$. $\Gamma$ is a typing environment, that maps variables to types. The notation $C : (\tau, \tau')$ means that constructor $C$ expects a value of type $\tau$ and produces a value of type $\tau'$. $\mathbb{S}$ denotes an arbitrary Skeletal Semantics. There is a distinction between variables defined by let

$$\text{Funs} \in \textsc{TypeEnv} \times \textsc{Skeleton} \to \left\{ \Gamma, (\lambda p : \tau \to S) \;\middle|\; \begin{array}{l} \Gamma \in \textsc{TypeEnv} \\ p \in \textsc{Pattern} \\ \tau \in \textsc{Type} \\ S \in \textsc{Skeleton} \end{array} \right\}$$

$$\begin{aligned}
\text{Funs}(\Gamma, \textbf{let } p = S_1 \textbf{ in } S_2) &= \text{Funs}(\Gamma, S_1) \cup \text{Funs}(\Gamma + p \mapsto \tau, S_2) \\
\text{Funs}(\Gamma, \textbf{branch } S_1 \ldots S_n \textbf{ end}) &= \bigcup_{i=1}^{n} \text{Funs}(\Gamma, S_i) \\
\text{Funs}(\Gamma, t_0\ t_1 \ldots t_n) &= \bigcup_{i=0}^{n} \text{Funs}(\Gamma, t_i) \\
\text{Funs}(\Gamma, \lambda p : \tau \to S_0) &= \{\Gamma, \lambda p : \tau \to S_0\} \cup \text{Funs}(\Gamma + p \mapsto \tau, S_0) \\
\text{Funs}(\Gamma, (t_1, \ldots, t_n)) &= \bigcup_{i=1}^{n} \text{Funs}(\Gamma, t_i) \\
\text{Funs}(\Gamma, C\,t) &= \text{Funs}(\Gamma, t) \\
\text{Funs}(\Gamma, x) &= \emptyset
\end{aligned}$$

The set of abstractions in the Skeletal Semantics $\mathbb{S}$

$$\text{Funs}(\mathbb{S}) \equiv \bigcup_{\textbf{val } x:\tau = t \in \mathbb{S}} \text{Funs}(\varnothing, t)$$

Figure 2.4: Definition of Funs

bindings and variables defined in the Skeletal Semantics. This explains why there are two rules to type variables. Var types variables defined by let binding. TermDef types variables defined at top-level in the Skeletal Semantics.

A set Funs($\mathbb{S}$) is defined on Figure 2.4 that is the set of abstractions in the Skeletal Semantics $\mathbb{S}$.

## 2.2 A Big-Step Semantics for While

To get a semantics of While, we first need to define the values associated with each type. Then an instantiation of the unspecified terms must be given. Finally, by combining this with an interpretation of Skel, a big-step semantics of While is obtained.

$$\mathrm{NC}(\tau_1 \to \tau_2) = \{\, f \mid \textbf{val}\ f : \tau_1 \to \tau_2[= t] \in \mathbb{S} \,\}$$
$$\mathrm{AC}(\tau_1 \to \tau_2) = \{\, (\Gamma, p, S, E) \mid (\Gamma, \lambda p : \tau_1 \to \tau_2 \cdot S) \in \mathrm{Funs}(\mathbb{S}) \wedge \Gamma \vDash E \wedge \Gamma + p \mapsto \tau_1 \vdash S : \tau_2 \,\}$$
$$\textsc{Fun} = \mathrm{NC}(\tau_1 \to \tau_2) \cup \mathrm{AC}(\tau_1 \to \tau_2)$$

Figure 2.5: Named Closures and Anonymous Closures

$$\frac{\vdash v \in V(\tau) \qquad C : (\tau, \tau_a)}{\vdash C\ v \in V(\tau_a)}\ \textsc{Const} \qquad \frac{\forall 1 \le i \le n \qquad \vdash v_i \in V(\tau_i)}{\vdash (v_1, \ldots, v_n) \in V(\tau_1 \times \cdots \times \tau_n)}\ \textsc{Tuple}$$

$$\frac{(\Gamma, p, S, E) \in \mathrm{AC}(\tau_1 \to \tau_2)}{\vdash (\Gamma, p, S, E) \in V(\tau_1 \to \tau_2)}\ \textsc{Clos} \qquad \frac{f \in \mathrm{NC}(\tau_1 \to \tau_2)}{\vdash f \in V(\tau_1 \to \tau_2)}\ \textsc{Def}$$

$$\frac{\mathrm{dom}\,\Gamma = \mathrm{dom}\,E \qquad \forall x \in \mathrm{dom}\,E,\ \Gamma \vdash E(x) : \Gamma(x)}{\Gamma \vDash E}\ \textsc{Env}$$

$$\textsc{Val} = \bigcup_{\tau \in \textsc{Type}} V(\tau)$$
$$\textsc{Env} = \textsc{Var} \hookrightarrow \textsc{Val}$$

Figure 2.6: Definition of Values

### 2.2.1 Definition of Values for While

For each type $\tau$ of Skel, a set of value $V(\tau)$ is defined. One could first provide instantiation for the unspecified types, and values of specified types be generated by applying constructors to values. However, this is a limitation as an unspecified type could not be defined mutually recursively with a specified type for example. We define the interpretation of types such that unspecified types can be defined relative to other types.

First, named closures $\mathrm{NC}(\tau_1 \to \tau_2)$ and anonymous closures $\mathrm{AC}(\tau_1 \to \tau_2)$ are defined on Figure 2.5. A named closure of type $\tau_1 \to \tau_2$ is a function name $f$ with type $\tau_1 \to \tau_2$ that is defined at top-level in the skeletal semantics. An anonymous closure is a tuple of a typing environment $\Gamma$ mapping variables to type, a pattern $p$, a skeleton $S$, and an environment that maps variables to values, formally defined in the next paragraph.

The definitions of values for every type except unspecified ones are given in Figure 2.6.

$$\frac{id \in \mathcal{V} = \{x, y, z...\}}{\vdash id \in V(ident)}\ \textsc{Ident} \qquad \frac{l \in \mathbb{Z}}{\vdash l \in V(lit)}\ \textsc{Lit} \qquad \frac{i \in \mathbb{Z}}{\vdash i \in V(int)}\ \textsc{Int}$$

$$\frac{\forall x \in \text{dom } s,\ s(x) \in V(int)}{\vdash s \in V(store)}\ \textsc{Store}$$

Figure 2.7: Instantiation of Values for Unspecified Types

$$\begin{aligned}
[\![litToInt]\!](n) &= \{n\} & [\![add]\!](n_1, n_2) &= \{n_1 + n_2\} \\
[\![lt]\!](n_1, n_2) &= (n_1 < n_2)\ \texttt{?}\ \{1\}\ \texttt{:}\ \{0\} & [\![rand]\!](n_1, n_2) &= \{\ n \mid n_1 \leq n \leq n_2\ \} \\
[\![isZero]\!](n) &= (n = 0)\ \texttt{?}\ \{()\}\ \texttt{:}\ \{\} & [\![isNotZero]\!](n) &= (n \neq 0)\ \texttt{?}\ \{()\}\ \texttt{:}\ \{\} \\
[\![read]\!](x, s) &= \{s(x)\} & [\![write]\!](x, s, n) &= \{s[x \leftarrow n]\}
\end{aligned}$$

Figure 2.8: Instantiations of Unspecified Values for the While Language

The notation na($\tau$) means that $\tau$ is not an arrow type, and therefore not a function. The rule Env defines a special value, called an environment that maps Skel variables to values and is used to give meaning to free variables of a skeleton. We write Val the set of all values of every types and Env the set of partial functions from skeletal variables to values.

The definitions of values given previously are language-agnostic. The benefits of the Skeletal Semantics approach is that a lot of the work is done once and for all. We only need to instantiate the values of unspecified types, as presented on Figure 2.7.

Finally, there remains to instantiate the non-specified values. The specification of an unspecified value of type $\tau$ such that na($\tau$) is a set $\mathcal{P}(V(\tau))$: the specification can be a set as it is often useful to model non-determinism. The specification of an unspecified value of type $\tau_1 \to \tau_2$ is a function of type $V(\tau_1) \to \mathcal{P}(V(\tau_2))$. The use of sets is here also useful to model non-determinism: for given inputs there can be several outputs. The instantiations of unspecified values for the While language are given on Figure 2.8. The ternary $(cond)$ **?** $exp1$ **:** $exp2$ evaluates to $exp1$ if the condition $cond$ is true, or $exp2$ otherwise. There remains to interpret the meta-language Skel to obtain a semantics for our While language.

### 2.2.2 The Big-Step Semantics of Skel

The big-step semantics of Skel is presented on Figure 2.9. There are four rules to evaluate variables, as there are implicitly three types of variables. A variable defined by a let binding evaluates to what it is bound to in the current environment (VAR rule). A variable defined in the Skeletal Semantics of the language that has arrow type returns a pair of the name of the function (TERMCLOS rule). A variable that is defined in the Skeletal Semantics, bound to a specified term and which type is not an arrow is evaluated by using its definition in the Skeletal Semantics (TERMSPEC). A variable that is defined in the Skeletal Semantics, not bound to a term and which type is not an arrow returns non-deterministically a value from the set defined at instantiation of the variable (TERMUNSPEC rule). The evaluation of a constructor is the application of the constructor to the evaluation of the term (CONST rule). The evaluation of a tuple is the tuple of the evaluation of terms (TUPLE rule). The evaluation of a function is a closure: a triplet of the pattern representing the parameter, the code and the current environment to give meaning to free variables of the code (CLOS rule). Rule LETIN is usual for functional language. Rule BRANCH evaluates a branching by non-deterministically returning the result of the evaluation of a branch. To evaluate an application, we define another relation with three rules. Rule Rule CLOS is the application of a closure: a new binding is added in the environment where the pattern maps-to the argument of the function (the formal definition of the extension of environment is given later). Then, the body of the function is evaluated in this new environment. Rules SPEC and UNSPEC are the application of a named closure to an argument. In the SPEC rule, the function definition is fetched in the skeletal semantics, and is evaluated and applied to the argument. In the UNSPEC rule, the instantiation is applied to the argument, and one of the results is returned. Non-specified functions may be non-deterministic since their instantiation may return a set.

Finally, the extension of environment rules is defined in Figure 2.10 as a set of rule. It is defined as a relation between a triplet of an environment, a pattern, a value, to a new environment. The relation is defined by induction on the pattern. When the pattern is a wildcard, the relation maps the environment to the same environment (ASN-WILDCARD rule). When the pattern is a variable, the relation maps the environment to an identical environment but with a new binding between the variable and the value (ASN-VAR). When the pattern is a constructor applied to a pattern, the value is expected to be the same constructor applied to a value. The relation is defined recursively by removing the constructors (ASN-CONSTR rule). When the pattern is a tuple pattern, the environments

$$\Downarrow_t \in \mathcal{P}\left((\textsc{Env} \times \textsc{Term}) \times \textsc{Val}\right)$$
$$\Downarrow_S \in \mathcal{P}\left((\textsc{Env} \times \textsc{Skeleton}) \times \textsc{Val}\right)$$
$$\Downarrow_{app} \in \mathcal{P}\left((\textsc{Fun} \times \textsc{Val}) \times \textsc{Val}\right)$$

$$\frac{E(x) = v}{E, x \Downarrow_t v}\ \textsc{Var} \qquad \frac{\textbf{val}\ f : \tau_1 \to \tau_2 [= t]}{E, f \Downarrow_t f}\ \textsc{TermClos}$$

$$\frac{\textbf{val}\ x : \tau = t \in \mathbb{S} \qquad \mathrm{na}(\tau) \qquad \emptyset, t \Downarrow_t v}{E, x \Downarrow_t v}\ \textsc{TermSpec}$$

$$\frac{\textbf{val}\ x : \tau \in \mathbb{S} \qquad \mathrm{na}(\tau) \qquad v \in [\![x]\!]}{E, x \Downarrow_t v}\ \textsc{TermUnspec} \qquad \frac{E, t \Downarrow_t v}{E, (\mathrm{C}\, t) \Downarrow_t \mathrm{C}\, v}\ \textsc{Const}$$

$$\frac{E, t_1 \Downarrow_t v_1 \qquad .. \qquad E, t_n \Downarrow_t v_n}{E, (t_1, .., t_n) \Downarrow_t (v_1, .., v_n)}\ \textsc{Tuple}$$

$$\frac{}{E, (\lambda p : \tau \to S)\Gamma, \lambda p \cdot S) \in \mathrm{Funs}(\mathbb{S}) \Downarrow_t (\Gamma, p, S, E)}\ \textsc{Clos}$$

$$\frac{E, S_1 \Downarrow_S v \qquad \vdash E + p \mapsto v \rightsquigarrow E' \qquad E', S_2 \Downarrow_S w}{E, \mathrm{let}\, p = S_1 \,\mathrm{in}\, S_2 \Downarrow_S w}\ \textsc{LetIn}$$

$$\frac{E, S_i \Downarrow_S v}{E, (S_1, .., S_n) \Downarrow_S v}\ \textsc{Branch} \qquad \frac{i \in \{0, 1\}\ ,\ E, t_i \Downarrow_t v_i \qquad v_0\ v_1 \Downarrow_{app} w}{E, (t_0\ t_1) \Downarrow_S w}\ \textsc{App}$$

$$\frac{\vdash E + p \mapsto v \rightsquigarrow E' \qquad E', S \Downarrow_S w}{(\Gamma, p, S, E)\ v \Downarrow_{app} w}\ \textsc{Clos}$$

$$\frac{\textbf{val}\ f : \tau_1 \to \tau_2 = t \in \mathbb{S} \qquad \emptyset, t \Downarrow_t v_0 \qquad v_0\ v \Downarrow_{app} w}{f\ v \Downarrow_{app} w}\ \textsc{Spec}$$

$$\frac{\textbf{val}\ f : \tau_1 \to \tau_2 \in \mathbb{S} \qquad w \in [\![f]\!](v)}{f\ v \Downarrow_{app} w}\ \textsc{Unspec}$$

Figure 2.9: Big-Step Semantics of Skel

$$\vdash \_ + \_ \mapsto \_ \rightsquigarrow \_ \in \mathcal{P}\left((\textsc{Env} \times \textsc{Pattern} \times \textsc{Val}) \times \textsc{Env}\right)$$

$$\frac{}{\vdash E + \_ \mapsto v \rightsquigarrow E}\ \textsc{Asn-Wildcard} \qquad \frac{}{\vdash E + x \mapsto v \rightsquigarrow E + x \mapsto v}\ \textsc{Asn-Var}$$

$$\frac{\vdash E + p \mapsto v \rightsquigarrow E'}{\vdash E + C\,p \mapsto C\,v \rightsquigarrow E'}\ \textsc{Asn-Constr}$$

$$\frac{\vdash E + p_1 \mapsto v_1 \rightsquigarrow E_2 \qquad \ldots \qquad \vdash E_n + p_n \mapsto v_n \rightsquigarrow E'}{\vdash E + (p_1, .., p_n) \mapsto (v_1, .., v_n) \rightsquigarrow E'}\ \textsc{Asn-Tuple}$$

Figure 2.10: Rule for Extension of Environments and Pattern Matching

are piped through each pair of corresponding patterns and values to obtain the new environment (Asn-Tuple rule).

It is now possible to evaluate While programs by meta-interpretation of the language Skel. We have instantiated the unspecified types and terms. To write While programs in Skel, we need to express constants of type *lit*. Because it is an unspecified type, there is no value that can be written in Skel. We add a few unspecified values to the skeletal semantics that will be our constants.

```
val zero : lit
val one : lit
val two : lit
```

We suppose that $[\![zero]\!] = \{0\}$ and so on for the other constants. Let $E_0$ be a Skel environment mapping variables to values and defined as:

$$E_0 = \{s \mapsto \sigma_0\}$$

where $\sigma_0$ is the while store mapping $x$ to 0. Using the previously defined interpretation rules of Skel, one can derive statements of the form:

$$E_0, \texttt{eval}\ (s, While(Leq(x, two), Assign(x, Plus(x, one)))) \Downarrow_S two$$

## 2.3 Related Work

Our work is part of a large research effort to define sound analyses and build correct abstract interpreters from semantic descriptions of languages. At its core, our approach

is the Abstract Interpretation [9, 10] of a semantic meta-language. Abstract Interpretation is a method designed by Cousot and Cousot to define sound static analyses from a concrete semantics. In his Marktoberdorf lectures [8], Cousot describes a systematic way to derive an abstract interpretation of an imperative language from a concrete semantics and mathematically proved sound. We chose to define the Abstract Interpretation of Skel, as it is designed to mechanise the semantics of languages. The benefit of analysing a meta-language is that a large part of the work to define and prove the correctness of the analysis is done once for every semantics mechanised with Skel. However, it is often less precise than defining a language-specific abstract interpretation. Moreover, there have been several papers describing methods to derive abstract interpretation from different types of concrete semantics [9, 44, 32], we chose to derive abstract interpreters from a big-step semantics of Skel.

Schmidt [44] shows how to define an abstract interpretation for $\lambda$-calculus from a big-step semantics defined co-inductively. The abstract interpretation of Skel and its correctness proof follow the methods described in the paper. However, Skel has more complex constructs than $\lambda$-calculus, especially branches. Moreover, the big-step of Skel is defined inductively, thus reasoning about non-terminating programs is not possible. Also, to prove the correctness of the abstract interpretation of Skel, we relate the big-step derivation tree to the abstract derivation tree, similar to Schmidt, but a key difference is that our proof is inductive when Schmidt's proof is co-inductive.

Lim and Reps propose the TSL system [24]: a tool to define machine-code instructions set and abstract interpretations. The specification of an instruction set in TSL is compiled into a Common Intermediate Representation (CIR). An abstract interpretation is defined on the CIR, therefore an abstract interpreter is derivable from any instruction set description. However, the TSL system is aimed at specifying and analysing machine code, and not languages in general. Moreover, it is unclear how it would be possible to define analyses on languages with more complex control-flow, like $\lambda$-calculus.

In the paper on Skeletal semantics, Bodin *et al.* [4] used skeletal semantics to define concrete and abstract interpretation of a small imperative language, and to prove the correctness of the abstract interpretation, relative to the concrete interpretation. Our work is similar as the objective is identical: from a formal description of a language, define a concrete and an abstract interpretation that are related by a correctness property. However, there are also key differences between both works. Bodin *et al.* have defined a pure abstract interpretation by a greatest fixpoint. As a consequence, the derivations may

be infinite and therefore not computable. Our objective is to get a computable abstract interpretation, and we use classical tools of abstract interpretation to achieve it. We incrementally compute the correct abstraction of a program by adding a state to the abstract interpretation, and therefore it is not pure, like the concrete interpretation of Skel, but it carries a state. We provide a method to add widening to the abstract interpretation to enforce convergence. Our abstract interpretation can be executed as an analyser, as demonstrated by our implementation [41].

The idea of defining an abstract interpreter of a meta-language to define analyses for languages has been explored, for example by Keidel, Poulsen and Erdweg [20]. They use arrows [14] as meta-language to describe interpreters. The concrete and abstract interpreters share code using the unified interface of arrows. By instantiating language-dependent parts for the concrete interpretation and the abstract interpretation, they obtain two interpreters that can be proven sound compositionally by proving that the abstract language-dependent parts are sound approximation of the concrete language-dependent parts, similar to Skel. However, we chose to use a dedicated meta-language, Skel, as its library [37] makes defining interpreters for Skel convenient and one objective is to use the NecroCoq tool [36] to generate mechanised proofs that our derived abstract interpreters are correct.

Chapter 3

# Program Points

The objective of this thesis is to define abstract interpretations from skeletal semantics. Program points are an essential component of analyses, therefore we propose a generic method to add program points to interpretations of Skel in a systematic way. This method is used to add program points to the big-step interpretation of Skel, and later to the abstract interpretation of Skel.

## 3.1 Program Points in Skel

Given a program of the object language, the While language for example, that we call **prog**, a program point is a reference to a sub-term of **prog**. A similar concept was briefly introduced in Section 1.3.5 on Abstract Interpretation with CFG. The skeletal semantics of While contains the type `stmt`, and is the type of statements, thus of While programs. A While program in Skel is a value from the algebraic data-type `stmt`. Let **prog** $\in V($`stmt`$)$, it can be represented as an Abstract Syntax Tree (AST). Then, a sub-term of **prog** can be denoted by a path from the root of the AST to the sub-term in the tree.

Take program `x := 0; while x` $\leq$ `2 do x := x + 1`, in Skel, it is represented as a

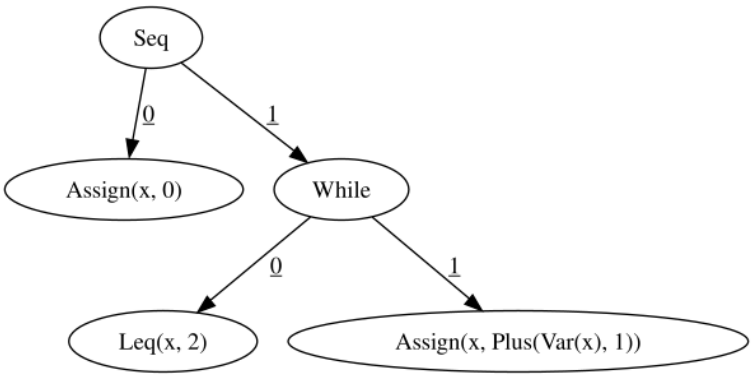



Figure 3.1: AST of a While Program in Skel

value from $V(\texttt{stmt})$. The AST of the program is shown in Figure 3.1. The nodes of the tree are constructors of specified types. The arcs from constructors to their children are labelled with numbers. The list of numbers from the root of the program to a sub-term is a program point. The integers of program points are underlined to differentiate them from integers of the While language. The empty program point $\underline{\epsilon}$ is a reference to the root of the tree, thus to the entirety of the program. The program point $\underline{0}$ maps-to the sub-term $Assign(x, 0)$. Indeed, it is the value obtained when starting from the root of the program and by taking the left-most child, labelled by $\underline{0}$. Similarly, the program point $\underline{1 \cdot 1}$ maps-to the sub-term $Assign(x, Plus(Var(x), 1))$.

**Definition 17** *The set of program points is the set of finite lists of natural integers. Given a set $S$, we note $S^*$ a finite list of elements of $S$.*

$$\mathit{ppoint} = \mathbb{N}^*$$

*We write $\underline{i} \cdot \text{pp}$ for the program point where the first element of the list is $\underline{i}$ and the rest is* pp*. Similarly, we write $\text{pp} \cdot \underline{i}$ for the program point where the beginning is the list* pp *and the last element is $\underline{i}$.*

We write $v@\,\text{pp}$ to denote the sub-term of $v$ at program point pp. The @ operator is formally defined as:

$$\begin{aligned} &v@\underline{\epsilon} = v \\ &C(v_0, .., v_{n-1})@\underline{i} \cdot \text{pp} = v_i@\,\text{pp} \qquad\qquad \text{when } 0 \leq i \leq n-1 \end{aligned}$$

Our method to add program points to an interpretation of Skel is to parameterise the interpretation by a *main program*, referred to as **prog**, of the object language. Then, the "program values" can be replaced by program points that are references to sub-terms of **prog**. A program value is a value that represents a program: for the While language, any value of type `stmt` is a program value. Therefore, given a skeletal semantics $\mathbb{S}$, some specified types must be said to be *program types*. We write ProgTypes for the set of program types: ADTs that describe programs. For the While language, a natural choice for program types is ProgTypes $= \{\texttt{stmt}, \texttt{expr}\}$. Indeed, statements and expressions are the programs of the While language. Now that we have defined the program types, we present new definitions for values. Values with program types are now expected to be program points that refer to the main program **prog**. The value **prog** must have a

$$\frac{\forall 1 \leq i \leq n \qquad \vdash v_i \in V^{\texttt{ppoint}}_{\mathbf{prog}}(\tau_i)}{\vdash (v_1, \ldots, v_n) \in V^{\texttt{ppoint}}_{\mathbf{prog}}(\tau_1 \times \cdots \times \tau_n)}\ \textsc{Tuple}$$

$$\frac{\vdash v \in V^{\texttt{ppoint}}_{\mathbf{prog}}(\tau) \qquad C : (\tau, \tau_a) \qquad \tau_a \notin \text{ProgTypes}}{\vdash C\ v \in V^{\texttt{ppoint}}_{\mathbf{prog}}(\tau_a)}\ \textsc{Const}$$

$$\frac{\tau_a \in \text{ProgTypes} \qquad \text{pp} \in \texttt{ppoint} \qquad \mathbf{prog}@\,\text{pp} \in V(\tau_a)}{\vdash \text{pp} \in V^{\texttt{ppoint}}_{\mathbf{prog}}(\tau_a)}\ \textsc{Const-ppoint}$$

$$\frac{(\Gamma, \lambda p : \tau_1 \to S) \in \text{Funs}(\mathbb{S}) \qquad \Gamma \vDash E \qquad \Gamma + p \mapsto \tau_1 \vdash S : \tau_2}{\vdash (p, S, E) \in V^{\texttt{ppoint}}_{\mathbf{prog}}(\tau_1 \to \tau_2)}\ \textsc{Clos}$$

$$\frac{\mathbf{val}\ f : \tau_1 \to \tau_2\ [= t] \in \mathbb{S}}{\vdash f \in V^{\texttt{ppoint}}_{\mathbf{prog}}(\tau_1 \to \tau_2)}\ \textsc{Def}$$

$$\frac{\text{dom}\,\Gamma = \text{dom}\,E \qquad \forall x \in \text{dom}\,E,\ \Gamma \vdash E(x) : \Gamma(x)}{\Gamma \vDash E}\ \textsc{Env}$$

$$\textsc{ValPP} = \bigcup_{\tau \in \textsc{Type}} V^{\texttt{ppoint}}_{\mathbf{prog}}(\tau)$$
$$\textsc{EnvPP} = \textsc{Var} \hookrightarrow \textsc{ValPP}$$

Figure 3.2: Definition of Values with Program Points

program type: $\mathbf{prog} \in V(\texttt{stmt})$. Thus, values with program points are defined relatively to values without program points. The new definitions of values with program points are presented on Figure 3.2. The relation $\vdash v \in V^{\texttt{ppoint}}_{\mathbf{prog}}(\tau)$ is defined similarly to the one of Section 2.2.1, but is now parameterised by the set ProgTypes, and by the main program **prog**. A new rule Const-ppoint introduces program points as values. If $\tau_a$ is a specified type and in the set of program types ProgTypes, then pp is a concrete value with the condition that the sub-value of **prog** at pp of the main program has type $\tau_a$. We write ValPP the new set of values with program points.

Let $\mathbf{prog} \in V(\texttt{stmt})$ be the program represented on Figure 3.1. Because $\mathbf{prog}@\underline{\epsilon} \in V(\texttt{stmt})$, then $\underline{\epsilon} \in V^{\texttt{ppoint}}_{\mathbf{prog}}(\texttt{stmt})$. Also, $\mathbf{prog}@\underline{10} = \mathit{Leq}(\mathit{Var}\,x, \mathit{Const}\,3) \in V(\texttt{expr})$, therefore $\underline{10} \in V^{\texttt{ppoint}}_{\mathbf{prog}}(\texttt{expr})$.

$$\text{unfold} : \mathcal{P}(\textsc{Type}) \times \textsc{Val} \times \texttt{ppoint} \hookrightarrow \textsc{ValPP}$$

$$\frac{C : (\tau_0, .., \tau_{n-1}) \qquad \mathbf{prog}@\,\text{pp} = C(v_0, .., v_{n-1}) \qquad v_i \in V(\tau_i) \qquad v'_i = \textbf{if } \tau_i \in \text{ProgTypes } \textbf{then } \text{pp} \cdot i \textbf{ else } \text{unfold}(\text{ProgTypes}, \mathbf{prog}, \text{pp} \cdot i)}{\text{unfold}(\text{ProgTypes}, \mathbf{prog}, \text{pp}) = (v'_0, .., v'_{n-1})}\ \textsc{Unfold}$$

Figure 3.3: Definition of the Unfolding of Program Points

### 3.1.1 Pattern-matching of Program Points

The big-step interpretation of Skel requires small changes to work with program points, and only the pattern-matching is impacted. Indeed, a program point pp matches a sub-program of **prog**, noted **prog**@ pp of the form $C(v_0, .., v_{n-1})$, as program types are ADTs. An ADT value may be matched with a pattern during the interpretation. To handle pattern-matching, a program point can be *unfolded*, meaning that the children of the program point are exhibited. Unfolding is formally defined on Figure 3.3. To unfold a program point pp, its corresponding value **prog**@ pp is inspected and is expected to be a constructor applied to values, the children. A child can be replaced by its corresponding program point if it has a program type, or it can recursively unfolded otherwise. For example, given **prog** of Figure 3.1, unfolding $\epsilon$ gives $(\underline{0}, \underline{1})$. Indeed, **prog**@$\underline{\epsilon}$ has the form $Seq(stmt_0, stmt_1)$ with $(stmt_0, stmt_1)$ a tuple of statements that are the children of $prog@\epsilon$. Because `stmt` $\in$ ProgTypes, the children are replaced by program points because they both have type `stmt` $\in$ ProgTypes. On the other hand, unfolding $\underline{0 \cdot 0}$ directly returns $x$, as $prog@\underline{0 \cdot 0} = x \in V(\texttt{ident})$ and identifiers have not program types.

This unfolding mechanism is added to pattern matching in the extension of environment presented in Figure 3.4. The extension of environments is now parameterised with ProgTypes and **prog**. The rules are similar to the ones given on Figure 2.10 defined for the big-step interpretation, but with a new rule: Asn-Unfold. To perform the pattern-matching of pp with $C\ p$, the value **prog**@ pp must have constructor $C$ at the root. The parameters that have a program type are replaced by their program points and the pattern-matching is performed recursively. Let **prog** be our running example presented on Figure 3.1. Here is what happens when an environment $E$ is extended with pattern

$$\vdash \_ + \_ \mapsto \_ \rightsquigarrow \_ \in \mathcal{P}\left((\textsc{EnvPP} \times \textsc{Pattern} \times \textsc{ValPP}) \times \textsc{EnvPP}\right)$$

$$\frac{}{\text{ProgTypes}, \mathbf{prog} \vdash E + \_ \mapsto v \rightsquigarrow E}\ \textsc{Asn-Wildcard}$$

$$\frac{}{\text{ProgTypes}, \mathbf{prog} \vdash E + x \mapsto v \rightsquigarrow (x, v) :: E}\ \textsc{Asn-Var}$$

$$\frac{\text{ProgTypes}, \mathbf{prog} \vdash E + p \mapsto v \rightsquigarrow E'}{\text{ProgTypes}, \mathbf{prog} \vdash E + C\,p \mapsto C\,v \rightsquigarrow E'}\ \textsc{Asn-Const}$$

$$\frac{\begin{array}{c}\text{ProgTypes}, \mathbf{prog} \vdash E + p_1 \mapsto v_1 \rightsquigarrow E_2 \\ \ldots \qquad \text{ProgTypes}, \mathbf{prog} \vdash E_n + p_n \mapsto v_n \rightsquigarrow E'\end{array}}{\text{ProgTypes}, \mathbf{prog} \vdash E + (p_1, .., p_n) \mapsto (v_1, .., v_n) \rightsquigarrow E'}\ \textsc{Asn-Tuple}$$

$$\frac{\text{ProgTypes}, \mathbf{prog} \vdash E + p \mapsto \text{unfold}(\text{ProgTypes}, \mathbf{prog}, \text{pp}) \rightsquigarrow E'}{\text{ProgTypes}, \mathbf{prog} \vdash E + C\,p \mapsto \text{pp} \rightsquigarrow E'}\ \textsc{Asn-Unfold}$$

Figure 3.4: Rule for Extension of Environments and Pattern Matching with Program Points

$Assign(\_, \_)$ and program point $\underline{1 \cdot 1}$:

$$\frac{\begin{array}{c}\mathbf{prog}@\underline{1 \cdot 1} = Assign(x, Plus(Var(x), 1)) \\ Assign : (\texttt{ident} \times \texttt{expr}, \texttt{stmt}) \qquad \text{ProgTypes}, \mathbf{prog} \vdash E + (\_, \_) \mapsto (x, \underline{1 \cdot 1 \cdot 1}) \rightsquigarrow E\end{array}}{\text{ProgTypes}, \mathbf{prog} \vdash E + Assign(\_, \_) \mapsto \underline{1 \cdot 1} \rightsquigarrow E}$$

First, we see that $\mathbf{prog}@\underline{1 \cdot 1}$ is $Assign(x, Plus(Var(x), 1))$ and can be matched with pattern $Assign(\_, \_)$. The subterm $x$ has type `ident` and is left as is. However $Plus(Var(x), 1)$ has type `expr` $\in$ ProgTypes. Therefore, it is replaced by its program point $\underline{1 \cdot 1 \cdot 1}$. Finally, the extension of environment is done recursively by matching pattern $(\_, \_)$ with $(x, \underline{1 \cdot 1 \cdot 1})$.

## 3.2 Big-Step Interpretation of Skel with Program Points

The big-step interpretation of Skel with program points is shown on Figure 3.5. It is almost identical to the big-step interpretation of Skel presented in previous sections. However, the evaluation of skeletons and applications are now parameterised by a main program

$$\Downarrow_t \in \mathcal{P}\left((\textsc{EnvPP} \times \textsc{Term}) \times \textsc{ValPP}\right)$$
$$\Downarrow_S^{\text{ProgTypes},\mathbf{prog}} \in \mathcal{P}\left((\textsc{EnvPP} \times \textsc{Skeleton}) \times \textsc{ValPP}\right)$$

$$\frac{E(x) = v}{E, x \Downarrow_t v}\ \textsc{Var} \qquad \frac{\mathbf{val}\ f : \tau_1 \to \tau_2[= t]}{E, f \Downarrow_t f}\ \textsc{TermClos}$$

$$\frac{\mathbf{val}\ x : \tau = t \in \mathbb{S} \qquad \mathrm{na}(\tau) \qquad \emptyset, t \Downarrow_t v}{E, x \Downarrow_t v}\ \textsc{TermSpec}$$

$$\frac{\mathbf{val}\ x : \tau \in \mathbb{S} \qquad v \in [\![x]\!]^{\texttt{ppoint}}}{E, x \Downarrow_t v}\ \textsc{TermUnspec} \qquad \frac{E, t \Downarrow_t v}{E, (\mathrm{C}\, t) \Downarrow_t \mathrm{C}\, v}\ \textsc{Const}$$

$$\frac{E, t_1 \Downarrow_t v_1 \qquad .. \qquad E, t_n \Downarrow_t v_n}{E, (t_1, .., t_n) \Downarrow_t (v_1, .., v_n)}\ \textsc{Tuple} \qquad \frac{}{E, (\lambda p : \tau \to S) \Downarrow_t (p, S, E)}\ \textsc{Clos}$$

$$\frac{\begin{array}{c} E, S_1 \Downarrow_S^{\text{ProgTypes},\mathbf{prog}} v \\ \text{ProgTypes}, \mathbf{prog} \vdash E + p \mapsto v \rightsquigarrow E' \qquad E', S_2 \Downarrow_S^{\text{ProgTypes},\mathbf{prog}} w \end{array}}{E, \mathrm{let}\, p = S_1 \,\mathrm{in}\, S_2 \Downarrow_S^{\text{ProgTypes},\mathbf{prog}} w}\ \textsc{LetIn}$$

$$\frac{E, S_i \Downarrow_S^{\text{ProgTypes},\mathbf{prog}} v}{E, (S_1, .., S_n) \Downarrow_S^{\text{ProgTypes},\mathbf{prog}} v}\ \textsc{Branch} \qquad \frac{E, t \Downarrow_t v \qquad f\ v \Downarrow_{app}^{\text{ProgTypes},\mathbf{prog}} w}{E, (f\ t) \Downarrow_S^{\text{ProgTypes},\mathbf{prog}} w}\ \textsc{App}$$

$$\frac{\text{ProgTypes}, \mathbf{prog} \vdash E + p \mapsto v \rightsquigarrow E' \qquad E', S \Downarrow_S^{\text{ProgTypes},\mathbf{prog}} w}{(p, S, E)\ v \Downarrow_{app}^{\text{ProgTypes},\mathbf{prog}} w}\ \textsc{Clos}$$

$$\frac{\mathbf{val}\ f : \tau_1 \to \tau_2 = t \in \mathbb{S} \qquad \emptyset, t \Downarrow_t v_0 \qquad v_0\ v \Downarrow_{app}^{\text{ProgTypes},\mathbf{prog}} w}{f\ v \Downarrow_{app}^{\text{ProgTypes},\mathbf{prog}} w}\ \textsc{Spec}$$

$$\frac{\mathbf{val}\ f : \tau_1 \to \tau_2 \in S \qquad w \in [\![f]\!]^{\texttt{ppoint}}(v)}{f\ v \Downarrow_{app}^{\text{ProgTypes},\mathbf{prog}} w}\ \textsc{Unspec}$$

Figure 3.5: Big-Step Semantics of Skel with Program Points

**prog** and a set of program types ProgTypes. The extension of environment in the LetIn rule and the Clos rule is the one presented in Section 3.1.1. The instantiation of an unspecified term for the interpretation with program points $x$ is written $[\![x]\!]^{\texttt{ppoint}}$.

Finally, to obtain a new big-step interpretation with program points, unspecified types and terms must be provided. With the While language, neither the definitions of the types nor the terms need to be modified. Thus, we use the definition presented on Figure 2.7 and on Figure 2.8.

Let **prog** be the program of Figure 3.1. Let $E = \{s \mapsto \{\}, t \mapsto \epsilon\}$. Then one can derive the following statement:

$$E, eval\ (s,t) \Downarrow_S^{\text{ProgTypes},\mathbf{prog}} \{x \mapsto 2\}$$

## 3.3 Correctness Theorem

**Definition 18** *To relate values with and without program points, we define the following function $\gamma$, such that $\forall \tau, \gamma_\tau \in V_{\boldsymbol{prog}}^{\boldsymbol{ppoint}}(\tau) \to V(\tau)$. The function $\gamma$ is parameterised by the main program **prog** of type $\tau \in$ ProgTypes and satisfies the following constraints.*

- $\gamma_\tau(\text{pp}) = \boldsymbol{prog}@\,\text{pp}$ *and* $\tau \in$ ProgTypes
- $\gamma_{\tau_1 \times .. \times \tau_n}((v_1, \ldots, v_n)) = (\gamma_{\tau_1}(v_1), .., \gamma_{\tau_n}(v_n))$
- $\gamma_{\tau_a}(C\ v) = C\ \gamma_\tau(v)$ *with* $C : (\tau, \tau_a)$
- *Suppose* $\Gamma \vDash E$, *and* $\Gamma \vDash E'$
  $\gamma_{env}(E') = E \iff \text{dom}\, E' = \text{dom}\, E \wedge \forall x \in \text{dom}\, E', \gamma_{\Gamma(x)}(E'(x)) = E(x)$
- $\gamma_{\tau_1 \to \tau_2}((p, S, E')) = (p, S, \gamma_{env}(E'))$
- *Suppose* $\boldsymbol{val}\ f : \tau_1 \to \tau_2\, [= t] \in \mathbb{S}$ *and* $\text{na}(\tau_2)$, *then*
  $\gamma_\tau(f) = f$

**Definition 19** *Take $f$ such that $\boldsymbol{val}\ f : \tau_1 \to \tau_2 \in \mathbb{S}$ and $\text{na}(\tau_2)$:*

$$\gamma_{unspec}([\![f]\!]^{\boldsymbol{ppoint}}) = [\![f]\!] \iff \forall (v', v) \in V_{\boldsymbol{prog}}^{\boldsymbol{ppoint}}(\tau_1) \times V(\tau_1)\ \textit{such that}\ \gamma_{\tau_1}(v') = v$$
$$\gamma_{\tau_2}([\![f]\!]^{\boldsymbol{ppoint}}(v')) = [\![f]\!](v)$$

**Theorem 2** *Let* $\gamma_{env}(E') = E$*, and suppose for all unspecified functions* $f$*,* $\gamma_{unspec}([\![f]\!]^{ppoint}) = [\![f]\!]$*, then:*

$$E, S \Downarrow_S v \implies \exists v', \quad E', S \Downarrow_S v' \text{ and } \gamma_\tau(v') = v$$

Chapter 4

# Abstract Interpretation of Skel

## 4.1 Methodology

We define an abstract interpretation of Skel that is sound with respect to the big-step semantics of Section 2.2.2. This abstract interpretation will serve as the foundation of a methodology for building abstract interpreters for languages from their skeletal semantics. In this methodology, the user provides several ingredients to generate such an abstract interpreter.

- For each unspecified type, an abstract domain must be defined. It consists of the definition of the abstract values of the type, a partial order, and an abstract union. To prove the correctness of the analysis, a concretisation function for each unspecified type is also required. From these definitions, the partial orders, the abstract unions, and the concretisation functions are automatically derived for every other types by our framework, without additional work required.

- A *State of the Abstract Interpretation* (AI-state in the following) is used to carry persistent information during the abstract evaluation. Abstract Interpretation of a language from its skeletal semantics is done by meta-interpretation of Skel, a pure functional language. Thus, a natural interpretation of Skel would be stateless, and without persistent information. However, in abstract interpretation, we want to compute correct approximations incrementally for every program points of the analysed program. Thus, we have added an AI-State for our interpretation of Skel. It is carried during the computations and holds information that is incrementally computed. For instance, in our While language, the AI-state records the current approximation of the abstract stores for every program point, as is done in Section 1.3.5 when defining an abstract interpretation of the While language from its CFG semantics.

- The AI-state may be modified at the start and end of a call to a specified function. This is typically used for evaluation functions, such as `eval_stmt`, to update the abstractions before and after executing a sub-program. Indeed, `eval_stmt` (and most of recursive evaluation functions) is called recursively on sub-programs of the analysed program. Therefore, the beginning and the end of such calls are relevant locations to update the abstractions stored in the AI-state.
- An instantiation of the unspecified terms must be provided, based on the abstract instantiations of types and of the AI-state.

Our framework provides an abstract meta-semantics of Skel that threads the AI-state through the evaluation, including calls to unspecified terms. As the foundational correctness property of the methodology, we prove that if the abstract instantiation of types and terms provided by the user satisfies some correctness criteria, then the whole abstract interpreter that is generated is also correct. The global correctness of the approach is proven once and for all assuming the correctness of the instantiations of the non-specified terms.

## 4.2 Definition of Abstract Values

For each type $\tau$ an abstract domain is defined and consist in a set of abstract values, a partial order, an abstract union, a smallest, and a biggest element must be defined. The abstract union must give an upper-bound, but we do not require that it is necessarily a least upper-bound.

**Definition 20** *We call an abstract domain of type $\tau$ a tuple $(V^\sharp(\tau), \sqsubseteq_\tau, \sqcup_\tau, \bot_\tau, \top_\tau)$ where*

- *$V^\sharp(\tau)$ is a set of abstract values*
- *$\sqsubseteq_\tau \in \mathcal{P}\left(V^\sharp(\tau) \times V^\sharp(\tau)\right)$ is a partial order*
- *$\sqcup_\tau \in V^\sharp(\tau) \times V^\sharp(\tau) \to V^\sharp(\tau)$ is an abstract union, which given two abstract values returns an upper-bound (ideally the least upper bound)*
- *$\bot_\tau$ and $\top_\tau$ are the least and greatest elements of $V^\sharp(\tau)$ respectively*

Abstract values for every type except unspecified ones are shown on Figure 4.2. Each type $\tau$ has a smallest and biggest element $\bot_\tau$ and $\top_\tau$ respectively (rules Bottom and

$$\mathrm{NC}^\sharp(\tau_1 \to \tau_2) = \{\ f \mid \mathbf{val}\ f : \tau_1 \to \tau_2[= t] \in \mathbb{S}\ \}$$
$$\mathrm{AC}^\sharp(\tau_1 \to \tau_2) = \left\{\ (\Gamma, p, S, E^\sharp) \;\middle|\; (\Gamma, \lambda p : \tau_1 \to \tau_2 \cdot S) \in \mathrm{Funs}(\mathbb{S}) \wedge \Gamma \vDash E^\sharp \wedge \Gamma + p \mapsto \tau_1 \vdash S : \tau_2 \ \right\}$$
$$\textsc{Funs}^\sharp = \mathrm{NC}^\sharp(\tau_1 \to \tau_2) \cup \mathrm{AC}^\sharp(\tau_1 \to \tau_2)$$

Figure 4.1: Named Closures and Anonymous Closures

$$\frac{}{\bot_\tau \in V^\sharp(\tau)}\ \textsc{Bottom} \qquad \frac{}{\top_\tau \in V^\sharp(\tau)}\ \textsc{Top}$$

$$\frac{\forall i \in \{1..n\}\, \forall j \in \{1..m\}\ \ v_i^j \in V^\sharp(\tau_i) \wedge v_i^j \neq \bot_{\tau_i} \qquad m \geq 1}{\left\{(v_1^1, .., v_n^1), .., (v_1^m, .., v_n^m)\right\} \in V^\sharp(\tau_1 \times .. \times \tau_n)}\ \textsc{Tuple}$$

$$\frac{v^\sharp \in V^\sharp(\tau) \qquad C : (\tau, \tau_a) \qquad \tau_a \notin \mathrm{ProgTypes} \qquad v^\sharp \neq \bot_\tau}{C\ v^\sharp \in V^\sharp(\tau_a)}\ \textsc{Alg}$$

$$\frac{F_1 \subseteq \mathrm{AC}^\sharp(\tau_1 \to \tau_2) \qquad F_2 \subseteq \mathrm{NC}^\sharp(\tau_1 \to \tau_2) \qquad F_1 \cup F_2 \neq \emptyset}{(F_1, F_2) \in V^\sharp(\tau_1 \to \tau_2)}\ \textsc{Clos}$$

$$\frac{\tau_a \in \mathrm{ProgTypes} \qquad \mathbf{prog}@\,\mathrm{pp} : \tau_a}{\mathrm{pp} \in V^\sharp(\tau_a)}\ \textsc{PPoint}$$

$$\frac{\mathrm{dom}\,\Gamma = \mathrm{dom}\,E^\sharp \qquad \forall x \in \mathrm{dom}\,E^\sharp\ E^\sharp(x) \in V^\sharp(\Gamma(x))}{\Gamma \vDash E^\sharp}\ \textsc{Env}$$

$$\textsc{Val}^\sharp = \bigcup_{\tau \in \textsc{Type}} V^\sharp(\tau)$$
$$\textsc{Env}^\sharp = \textsc{Var} \hookrightarrow \textsc{Val}^\sharp$$

Figure 4.2: Abstract Values

$$\frac{x \in \mathcal{X}}{x \in V^\sharp(\texttt{ident})}\ \textsc{Ident} \qquad \frac{l \in \mathbb{Z}}{l \in V^\sharp(\texttt{lit})}\ \textsc{Lit}$$

$$\frac{n_1, n_2 \in \mathbb{Z} \cup \{-\infty, +\infty\} \qquad n_1 \leq n_2}{[n_1; n_2] \in V^\sharp(\texttt{int})}\ \textsc{Int} \qquad \frac{s \in V^\sharp(\texttt{ident}) \hookrightarrow V^\sharp(\texttt{int})}{s \in V^\sharp(\texttt{store})}\ \textsc{Store}$$

Figure 4.3: Abstract Values for Unspecified Types

Top). An abstract tuple can be a set of tuples of abstract values that are not a bottom value (rule Tuple). We choose to have finite sets of tuples of abstract values as abstract tuples because we want to keep some relations that we would lose by collapsing a set of tuples into one abstract tuple. The next chapter will show a case where it is crucial to have an expressive abstraction of tuples when presenting an analysis for $\lambda$-calculus. A value of an algebraic datatype can be a constructor applied to an abstract value that is not bottom (rule Alg). An abstract value with type $\tau_1 \to \tau_2$ is a pair of two sets: a set of named closures and a set of anonymous closures, and the union of the two sets should be non-empty (rule Clos). The sets of anonymous and named closures are detailed in the next paragraph. The rule PPoint defines program points as abstract values. The rule Env of Figure 4.2 defines abstract environments. Abstract environments are not abstract values but are defined mutually recursively to abstract values. An abstract environment $E^\sharp$ is defined relatively to a typing environment $\Gamma$, and is a partial function mapping skeletal variables to abstract values with types consistent with the typing environment $\Gamma$.

The sets of abstract named and anonymous closures are defined on Figure 4.1. The set of abstract named closures of type $\tau_1 \to \tau_2$ is $\mathrm{NC}^\sharp(\tau_1 \to \tau_2)$. It is almost identical to concrete named closures: a set of function names defined in the skeletal semantics $\mathbb{S}$. The set of anonymous closures of type $\tau_1 \to \tau_2$ is $\mathrm{AC}^\sharp(\tau_1 \to \tau_2)$. An anonymous closure is a quadruple $(\Gamma, p, S, E^\sharp)$ where $\Gamma$ is the typing environment for free variables in $S$. The pattern $p$ contains the variables to be bound when calling the function. The body of the function $S$ is a skeleton that is the code of the function. $E^\sharp$ is the abstract environment mapping the free variables of $S$ to abstract values. We recall that the relation $\Gamma \vDash E^\sharp$ means that $\Gamma$ is consistent with the typing environment. Finally, the set $\textsc{Fun}^\sharp$ is the union of the sets of abstract and named closures.

These definitions of abstract values do not depend on a skeletal semantics. However, to define the abstract values of a language, given its skeletal semantics, the definitions

of the abstract values of the unspecified types must be provided. Such rules are given on Figure 4.3 for the skeletal semantics of the While language. The benefit of specifying values of unspecified types with induction rules is to be able to define them mutually recursively with every other abstract values. One example is the definition of abstract stores which depends on the definitions of abstract identifiers and abstract integers. It could also depend on abstract values with types other than non-specified. We give a few examples of abstract values below built from the two sets of rules of Figures 4.2 and 4.3. When an abstract tuple is a singleton containing one tuple, brackets are dropped. Thus we do not write $Plus(\{Var(x), Const(1)\})$ but $Plus(Var(x), Const(1))$.

$$\frac{1 \in \mathbb{Z}}{1 \in V^\sharp(\texttt{lit})}\ \textsc{Lit} \qquad \frac{0, +\infty \in \mathbb{Z} \cup \{-\infty, +\infty\} \qquad 0 \leq +\infty}{[\![0, +\infty]\!] \in V^\sharp(\texttt{int})}\ \textsc{Int}$$

$$\frac{Var : (\texttt{ident}, \texttt{expr}) \qquad \dfrac{x \in \mathcal{X}}{x \in V^\sharp(\texttt{ident})}\ \textsc{Ident}}{Var(x) \in V^\sharp(\texttt{expr})}\ \textsc{Const}$$

$$\frac{Const : (\texttt{lit}, \texttt{expr}) \qquad \dfrac{1 \in \mathbb{Z}}{1 \in V^\sharp(\texttt{lit})}\ \textsc{Lit}}{Const(1) \in V^\sharp(\texttt{expr})}\ \textsc{Const}$$

$$\frac{\dfrac{\cdots}{Var(x) \in V^\sharp(\texttt{expr})}\ \textsc{Const} \qquad \dfrac{\cdots}{Const(1) \in V^\sharp(\texttt{expr})}}{(Var(x), Const(1)) \in V^\sharp(\texttt{expr} \times \texttt{expr})}\ \textsc{Tuple}$$

$$\frac{Plus : (\texttt{expr} \times \texttt{expr}, \texttt{expr}) \qquad \dfrac{\cdots}{Var(x), Const(1) \in V^\sharp(\texttt{expr} \times \texttt{expr})}\ \textsc{Tuple}}{Plus(Var(x), Const(1)) \in V^\sharp(\texttt{expr})}\ \textsc{Const}$$

**Comparisons of Abstract Values** We introduce a new notation to compare two sets of comparable values.

**Definition 21** *Let $(S, \leq)$ be a poset. Let $T_1, T_2 \in \mathcal{P}(S)$. The relation $\leq_{set}$ is defined as*

$$T_1 \leq_{set} T_2 \iff \forall s \in T_1\ \exists s' \in T_2\ s \leq s'$$

$$\sqsubseteq^\sharp_\tau \in \mathcal{P}\left(V^\sharp(\tau) \times V^\sharp(\tau)\right)$$

$$\frac{C : (\tau : \tau_a) \qquad v^\sharp \sqsubseteq^\sharp_\tau w^\sharp}{C\ v^\sharp \sqsubseteq^\sharp_{\tau_a} C\ w^\sharp}\ \textsc{Alg} \qquad \frac{t_1^\sharp \sqsubseteq^{\text{tuple}}_{set} t_2^\sharp}{t_1^\sharp \sqsubseteq^\sharp_{\tau_1 \times .. \times \tau_n} t_2^\sharp}\ \textsc{Tuple}$$

$$\frac{F_1 \subseteq G_1 \qquad F_2 \sqsubseteq^{\text{aclos}}_{set} G_2}{(F_1, F_2) \sqsubseteq^\sharp_{\tau_1 \to \tau_2} (G_1, G_2)}\ \textsc{Arrow}$$

$$\frac{\Gamma \vDash E_1^\sharp \wedge \Gamma \vDash E_2^\sharp \wedge \forall x \in \operatorname{dom} E_1^\sharp\ E_1^\sharp(x) \sqsubseteq^\sharp_{\Gamma(x)} E_2^\sharp(x)}{E_1^\sharp \sqsubseteq^\sharp_\Gamma E_2^\sharp}\ \textsc{Env} \qquad \frac{}{v^\sharp \sqsubseteq^\sharp_\tau \top_\tau}\ \textsc{Top}$$

$$\frac{}{\bot_\tau \sqsubseteq^\sharp_\tau v^\sharp}\ \textsc{Bot}$$

$$\frac{i \in \{1..n\}\ v_i^\sharp \sqsubseteq_{\tau_i} w_i^\sharp}{(v_1^\sharp, .., v_n^\sharp) \sqsubseteq^{\text{tuple}}_{\tau_1 \times .. \times \tau_n} (w_1^\sharp, .., w_n^\sharp)}$$

$$(\Gamma, p, S, E_1^\sharp) \sqsubseteq^{\text{aclos}}_{\tau_1 \to \tau_2} (\Gamma, p, S, E_2^\sharp) \iff \forall x \in \operatorname{dom} \Gamma^\sharp\ E_1^\sharp(x) \sqsubseteq_{\Gamma(x)} E_2^\sharp(x)$$

Figure 4.4: Comparisons of Abstract Values

*In other words, for every element $s$ of $T_1$, there exists an element $s'$ of $T_2$ such that $s$ is smaller than $s'$.*

Before defining the comparisons of abstract values, we defined two auxiliary comparison functions: one for tuples of abstract values and one for anonymous closures. These definitions will be used to define the comparisons of abstract values.

$$\frac{\forall i \in \{1..n\}\ v_i^\sharp \sqsubseteq_{\tau_i} w_i^\sharp}{(v_1^\sharp, .., v_n^\sharp) \sqsubseteq^{\text{tuple}} (w_1^\sharp, .., w_n^\sharp)} \qquad \frac{E_1^\sharp \sqsubseteq_\Gamma E_2^\sharp}{(\Gamma, p, S, E_1^\sharp) \sqsubseteq^{\text{aclos}} (\Gamma, p, S, E_2^\sharp)}$$

Now that the sets of abstract values have been instantiated for each type, we explain how they can be compared. The definitions of comparisons are similar to the definitions of abstract values: for every type other than unspecified, the comparison is given. Given a skeletal semantics, the comparisons must be provided for every unspecified type. The comparisons are order relations (reflexive, transitive, antisymmetric) $\sqsubseteq_\tau$ that satisfy the

$$\frac{m_1 \leq n_1 \qquad n_2 \leq m_2}{[n_1; n_2] \sqsubseteq_{\texttt{int}} [m_1; m_2]}\ \textsc{Int} \qquad \frac{\operatorname{dom} s_1 = \operatorname{dom} s_2 \qquad \forall x \in \operatorname{dom} s_1\ s_1(x) \sqsubseteq_{\texttt{int}} s_2(x)}{s_1 \sqsubseteq_{\texttt{store}} s_2}\ \textsc{Store}$$

Figure 4.5: Comparisons of Unspecified Types

conditions on Figure 4.4. Values from ADT are comparable only when they have the same constructor, and comparing two abstract values with the same constructor is equivalent to comparing their parameters (rule Alg). An abstract tuple $t_1^\sharp$ is smaller than an abstract tuple $t_2^\sharp$ if each tuple of abstract values in $t_1^\sharp$ is smaller than a tuple of abstract values in $t_2^\sharp$ (rule Tuple). The definition of the comparison of abstract tuple is done with the auxiliary comparison function of tuple of abstract values $\sqsubseteq^{\text{tuple}}_{\tau_1 \to \tau_2}$. Abstract functions are pairs of sets of anonymous and named closures. An abstract function $(F_1, F_2)$ is smaller than a second one $(G_1, G_2)$ when the named closures $F_1$ are included in $G_1$. Moreover, each anonymous closures of $F_2$ must be smaller than some anonymous closure in $G_2$ (rule Arrow). The comparison of anonymous closure is done using an auxiliary comparison function $\sqsubseteq^{\text{aclos}}_{\tau_1 \to \tau_2}$ that compares two abstract closures. An anonymous closure is smaller than a second one when it has the same typing environment, pattern and skeleton but the abstract environment is smaller than the second anonymous closure. Abstract environments are compared relatively to a typing environment $\Gamma$. An environment $E_1^\sharp$ is smaller than a second one $E_2^\sharp$ when both are typed by the same typing environment $\Gamma$, and thus have the same domain. Moreover, each skeletal variable $x$ in the domain $E_1^\sharp$ maps to a value that is smaller than $E_2^\sharp(x)$. Finally, $\top_\tau$ and $\bot_\tau$ are the greatest and least elements of $V^\sharp(\tau)$.

Given a skeletal semantics, the comparison of abstract values for unspecified types must be provided. The comparisons of every types are defined mutually recursively. For the While language, the comparisons are presented on Figure 4.5. The comparison of integers is the comparison of intervals. The comparison of stores is the point-wise extension of the comparison of integers. There are no rules for the comparisons of identifiers and literals that have been defined. It does not mean that values of these types can never be compared but that we have defined a flat lattice. Indeed, the definition in Figure 4.4 ensures that for every type, top is the greatest element, and bottom is the least element.

One may wonder, what is the difference between $[-\infty; +\infty]$ and $\top_{\texttt{int}}$. By definition, $\top_{\texttt{int}}$ is the bigger value and by our definition of the abstract comparison, both values are different. However, $[-\infty; +\infty]$ is bigger than every other values and the concretisation of both abstract values are equal. Hence, the only difference is that one is bigger than the

other by convention. The same argument can be made for the empty set and $\bot_{\texttt{int}}$.

A few examples of comparisons of abstract values are presented below. As statements and expressions are defined as ADT, two statements or two expressions can be compared which is unusual for an abstract interpretation. Indeed, abstract domains are defined for every types of the skeletal semantics, even for the program types: the types that define the programs of the language.

$$\frac{}{Assign(x, Const(1)) \sqsubseteq_{\texttt{Const}} Assign(x, Const(1))}\ \textsc{Refl}$$

$$\frac{Assign : (\texttt{ident} \times \texttt{expr}, \texttt{stmt}) \qquad \dfrac{\dfrac{}{x \sqsubseteq_{\texttt{ident}} x}\ \textsc{Refl} \qquad \dfrac{}{Const(1) \sqsubseteq_{\texttt{ident}} \top}\ \textsc{Top}}{(x, Const(1)) \sqsubseteq_{\texttt{ident}\times\texttt{expr}} (x, \top)}\ \textsc{Tuple}}{Assign(x, Const(1)) \sqsubseteq_{\texttt{Const}} Assign(x, \top)}\ \textsc{Const}$$

$$\frac{\dfrac{}{\bot_{\texttt{lit}} \sqsubseteq_{\texttt{lit}} 4}\ \textsc{Bot} \qquad \dfrac{[\![0,1]\!] \subseteq [\![0,+\infty]\!]}{[\![0,1]\!] \sqsubseteq_{\texttt{int}} [\![0,+\infty]\!]}\ \textsc{Int}}{\{x \mapsto \bot_{\texttt{lit}}, y \mapsto [\![0,1]\!]\} \sqsubseteq_{\texttt{store}} \{x \mapsto 4, y \mapsto [\![0,+\infty]\!]\}}\ \textsc{Store}$$

**Lemma 2** *$\sqsubseteq^\sharp_{\texttt{ident}}$, $\sqsubseteq^\sharp_{\texttt{lit}}$, $\sqsubseteq^\sharp_{\texttt{int}}$ and $\sqsubseteq^\sharp_{\texttt{store}}$ are orders (reflexive, transitive, antisymmetric). $\sqcup^\sharp_{\texttt{ident}}$, $\sqcup^\sharp_{\texttt{lit}}$, $\sqcup^\sharp_{\texttt{int}}$ and $\sqcup^\sharp_{\texttt{store}}$ give upper bounds.*

**Abstract Unions of Abstract Values** Abstract unions are defined by induction and are presented on Figure 4.6. An abstract union for type $\tau$ is a total function that given two abstract values of type $\tau$, returns a new value of type $\tau$ bigger than both arguments. The abstract union of two values of an ADT with the same constructor is the constructor applied to the abstract union of the parameters (rule Alg). The abstract union of two values of an ADT with different constructors is top (rule Alg-Top). The abstract union of tuples is set union (rule Tuple). The abstract union of abstract functions is a pair of sets (rule Arrow). The union for named closures is the set union of the sets of named closures. The union for anonymous closure is a set where anonymous closures with the same code are merged by the abstract union of their abstract environments. Therefore, for a given anonymous function of the skeletal semantics, an abstract function contains at most one abstract closure for this anonymous function. Supposing $F_2$ and $G_2$ are finite sets of anonymous closures, then $H_2$ as defined in rule Arrow is also finite. This ensures that the set of anonymous closures during the interpretation is kept finite. There is a

$$\sqcup^\sharp_\tau \in V^\sharp(\tau) \times V^\sharp(\tau) \to V^\sharp(\tau)$$

$$\frac{C : (\tau_1, \tau_2) \qquad v^\sharp = v_1^\sharp \sqcup_{\tau_1} v_2^\sharp}{(C\ v_1^\sharp) \sqcup^\sharp_{\tau_2} (C\ v_2^\sharp) = C\ v^\sharp}\ \textsc{Alg} \qquad \frac{C : (\tau_1, \tau_2) \qquad D : (\tau_1', \tau_2) \qquad C \neq D}{(C\ v_1^\sharp) \sqcup^\sharp_{\tau_2} (D\ v_2^\sharp) = \top_{\tau_2}}\ \textsc{Alg-Top}$$

$$\frac{\forall i \in \{1,2\}\ \ v_i^\sharp \neq \top \wedge v_i^\sharp \neq \bot \qquad v^\sharp = v_1^\sharp \cup v_2^\sharp}{v_1^\sharp \sqcup^\sharp_{\tau_1 \times .. \times \tau_n} v_2^\sharp = v^\sharp}\ \textsc{Tuple}$$

$$\frac{\begin{array}{c} H_1 = F_1 \cup G_1 \\ H_2 = \left\{ (\Gamma, p, S, E^\sharp) \;\middle|\; E^\sharp = \bigsqcup^\sharp \left\{ E_i^\sharp \;\middle|\; (\Gamma, p, S, E_i^\sharp) \in F_2 \cup G_2 \right\} \right\} \end{array}}{(F_1, F_2) \sqcup^\sharp_{\tau_1 \to \tau_2} (G_1, G_2) = (H_1, H_2)}\ \textsc{Arrow}$$

$$\frac{\forall i \in \{1,2\}\ \ \Gamma \vDash E_i^\sharp \qquad E^\sharp = \left\{ x \in \operatorname{dom}\Gamma \mapsto E_1^\sharp(x) \sqcup^\sharp_{\Gamma(x)} E_2^\sharp(x) \right\}}{E_1^\sharp \sqcup^\sharp_\Gamma E_2^\sharp = E^\sharp}\ \textsc{Env}$$

$$\frac{}{v^\sharp \sqcup^\sharp_\tau \top_\tau = \top_\tau \sqcup^\sharp_\tau v^\sharp = \top_\tau}\ \textsc{Top} \qquad \frac{}{v^\sharp \sqcup^\sharp_\tau \bot_\tau = \bot_\tau \sqcup^\sharp_\tau v^\sharp = v^\sharp}\ \textsc{Bot}$$

Figure 4.6: Abstract Union of Abstract Values

$$\frac{}{[n_1; n_2] \sqcup_{\texttt{int}} [m_1; m_2] = [\min(n_1, m_1); \max(n_2, m_2)]}\ \textsc{Int}$$

$$\frac{}{\{x_1 \mapsto i_1, .., x_n \mapsto i_n\} \sqcup_{\texttt{store}} \{x_1 \mapsto j_1, .., x_n \mapsto j_n\} = \{x_1 \mapsto i_1 \sqcup_{\texttt{int}} j_1, .., x_n \mapsto i_n \sqcup_{\texttt{int}} j_n\}}\ \textsc{Store}$$

Figure 4.7: Abstract Union of Values with Unspecified Types

finite number of named closures because a skeletal semantics is a finite set of type and term definitions. Therefore, in an abstract interpretation, an abstract function is a finite set of anonymous and named closures.

Abstract union rules for the unspecified types of While are given on Figure 4.7. The abstract union of two intervals is a convex hull (rule Int). The abstract union of stores is the point-wise extension of the abstract union of intervals (rule Store). The abstract unions for `ident` and `lit` are not shown here, but are defined to get flat lattices.

A few examples of abstract unions are given below.

$$\frac{}{[0; 1] \sqcup_{\texttt{int}} [4; 5] = [0; 5]}\ \textsc{Int}$$

$$\frac{}{\{x \mapsto [0; 1], y \mapsto [-\infty; 0]\} \sqcup_{\texttt{store}} \{x \mapsto [4; 5], y \mapsto [-1; 0]\} = \{x \mapsto [0; 5], y \mapsto [-\infty; 0]\}}\ \textsc{Store}$$

$$\frac{Const : (\texttt{lit}, \texttt{expr})}{Const(1) \sqcup_{\texttt{expr}} Const(2) = Const(\top_{\texttt{lit}})}\ \textsc{Const}$$

**Lemma 3** *If for every unspecified type $\tau_u$, $\sqsubseteq^\sharp_{\tau_u}$ is an order and $\sqcup^\sharp_{\tau_u}$ gives an upper bound, then for all $\tau$, $\sqsubseteq^\sharp_\tau$ is an order and $\sqcup^\sharp_\tau$ gives an upper bound.*

**Proof 1** *By induction on type $\tau$. Full proof in Chapter A.*

## 4.3 State of the Abstract Interpretation

The *State of the Abstract Interpretation* $\mathcal{A}$ (abbreviated in AI-state) contains information collected throughout the abstract interpretation. It is dependent on the analysis and the language and therefore must be provided, similarly to unspecified values. Skel is a pure language, and when defining an abstract interpretation, it is convenient to use an AI-state such that at every step of the analysis, it contains the current abstractions that have been

$$\frac{\begin{array}{c}\operatorname{dom}\mathcal{A}_1 \subseteq \operatorname{dom}\mathcal{A}_2 \qquad \texttt{Pos} \in \{\texttt{In}, \texttt{Out}\} \\ (\mathrm{pp}, \texttt{Pos}) \in \operatorname{dom}\mathcal{A}_1 \implies \mathcal{A}_1(\mathrm{pp}, \texttt{Pos}) \sqsubseteq^\sharp_{\texttt{store}} \mathcal{A}_2(\mathrm{pp}, \texttt{Pos})\end{array}}{\mathcal{A}_1 \sqsubseteq^\sharp \mathcal{A}_2} \textsc{Comparison}$$

$$\frac{\begin{array}{c}\operatorname{dom}\mathcal{A} = \operatorname{dom}\mathcal{A}_1 \cup \operatorname{dom}\mathcal{A}_2 \qquad \texttt{Pos} \in \{\texttt{In}, \texttt{Out}\} \\ (\mathrm{pp}, \texttt{Pos}) \in \operatorname{dom}\mathcal{A}_1 \cup \operatorname{dom}\mathcal{A}_2 \implies \mathcal{A}(\mathrm{pp}, \texttt{Pos}) = \mathcal{A}_1(\mathrm{pp}, \texttt{Pos}) \sqcup^\sharp_{\texttt{store}} \mathcal{A}_2(\mathrm{pp}, \texttt{Pos})\end{array}}{\mathcal{A}_1 \sqcup^\sharp \mathcal{A}_2 = \mathcal{A}} \textsc{Union}$$

Figure 4.8: Comparison and Union for the AI-state of While

computed. When analysing a given program, the analyser may study the same sub-term of the program several times in different contexts and the AI-state should maintain and merge the different abstractions computed at this point of the analysis. Having a persistent state is used in traditional abstract interpretation. For instance, in Section 1.3.5 about abstract interpretation from CFG for the While language, an abstract store is computed for every program point. We write `AbstState` the set of AI-states for a given skeletal semantics. The AI-state of the While skeletal semantics is a mapping from program points to abstract stores. We actually record two abstract stores per program point: one before (`In`) and after (`Out`) the evaluation of the sub-term. The notation `Pos` stands for either `In` or `Out`. We then define $\mathcal{A}$ as a mapping from program points and a `Pos` to abstract stores.

**Definition 22** *We define the set of AI-states for the while language as:*

$$\mathit{AbstState} \equiv (\mathit{ppoint} \times \{\mathit{In}, \mathit{Out}\}) \hookrightarrow V^\sharp(\mathit{store})$$

When $(\mathrm{pp}, \texttt{Pos}) \notin \operatorname{dom}\mathcal{A}$, we suppose $\mathcal{A}(\mathrm{pp}, \texttt{Pos}) = \bot_{\texttt{store}}$. The comparison and union of AI-state for While are presented on Figure 4.8. The comparison of AI-state is the point-wise extension of the comparison of stores. The abstract union is the point-wise extension of the union of stores.

During the abstract interpretation, an AI-state can be modified when calling a specified function, as it will be explained later. For this, update functions must be provided. There can be up to two update functions per specified function, as the AI-state can be modified before and after the call to the specified function. Suppose that $f : \tau_1 \to \tau_2$, then the

update functions have type:

$$\mathbf{update}_f^{in} : \texttt{AbstState} \times V^\sharp(\tau_1) \to V^\sharp(\tau_1)$$
$$\mathbf{update}_f^{out} : \texttt{AbstState} \times V^\sharp(\tau_1) \times V^\sharp(\tau_2) \to V^\sharp(\tau_2)$$

The update mechanism will be fully explained when defining the abstract interpretation of Skel in Section 4.7.

## 4.4 Concretisation of Abstract Values

To ensure the correctness of the abstract analysis, the abstract domains are linked to the concrete domains with concretisation functions. The same methodology used for comparisons and unions is used: the definitions of the concretisation functions for not unspecified types are defined, and the definitions of the concretisation functions for non-specified types must be provided for a given skeletal semantics. A concretisation function returns the set of concrete values denoted by an abstract value. There is one concretisation function per type. We assume they are provided for non-specified types, and show in this section how to extend them to all types.

A concretisation function for type $\tau$ maps an AI-state and an abstract value in $V^\sharp(\tau)$ to $\mathcal{P}\left(V^{\texttt{ppoint}}(\tau)\right)$, a set of concrete values. It takes the AI-state because we will see in the abstract interpretation for $\lambda$-calculus that an abstraction may depend on global information stored in the AI-state. We also define a function of concretisation $\gamma_\Gamma$ which maps abstract environments to sets of concrete environments. Formally, a concretisation function has type:

$$\gamma_\tau : \texttt{AbstState} \times V^\sharp(\tau) \to \mathcal{P}\left(V^{\texttt{ppoint}}(\tau)\right)$$

The concretisation functions are shown on Figure 4.9. The concretisation of an abstract tuple $t^\sharp$ is the union of the concretisation of the tuples of abstract values in $t^\sharp$. The concretisation of a tuple of abstract values is the Cartesian product of concretisations of each abstract value of the tuple. The concretisation of a constructor applied to an abstract value $C\, v^\sharp$ is the set of concrete values where $C$ is applied to values in the concretisation of $v^\sharp$. The concretisation of a program point is a singleton containing this

$$\gamma_{\tau_0 \times .. \times \tau_{n-1}}(\mathcal{A}, t^\sharp) = \bigcup_{(v_0^\sharp, .., v_{n-1}^\sharp) \in t^\sharp} \gamma_{\tau_0}(\mathcal{A}, v_0^\sharp) \times .. \times \gamma_{\tau_{n-1}}(\mathcal{A}, v_{n-1}^\sharp)$$

$$\gamma_{\tau_2}(\mathcal{A}, C\, v^\sharp) = \left\{ C\, v \mid C : (\tau_1, \tau_2),\ v \in \gamma_{\tau_2}(\mathcal{A}, v^\sharp) \right\}$$

$$\gamma_\tau(\mathcal{A}, \mathrm{pp}) = \{\mathrm{pp}\}$$

$$\gamma_{\tau_1 \to \tau_2}(\mathcal{A}, (F_1, F_2)) = F_1 \cup \left\{ (\Gamma, p, S, E) \;\middle|\; (\Gamma, p, S, E^\sharp) \in F_2 \ \wedge\ E \in \gamma_\Gamma(\mathcal{A}, E^\sharp) \right\}$$

$$\gamma_\Gamma(\mathcal{A}, E^\sharp) = \left\{ E \;\middle|\; \Gamma \vDash E \wedge \Gamma \vDash E^\sharp \wedge \forall x \in \mathrm{dom}\,\Gamma, E(x) \in \gamma_{\Gamma(x)}(\mathcal{A}, E^\sharp(x)) \right\}$$

$$\gamma(\mathcal{A}, \bot_\tau) = \emptyset$$

$$\gamma(\mathcal{A}, \top_\tau) = V^{\texttt{ppoint}}(\tau)$$

Figure 4.9: Concretisation Functions of Abstract Values

program point. The concretisation of an abstract function $(F_1, F_2)$ is a set containing the named closures $F_1$, and concrete anonymous closures of the form $(\Gamma, p, S, E)$ approximated by an abstract closure $(\Gamma, p, S, E^\sharp)$ in $F_2$. The concretisation of an abstract environment $E^\sharp$ is parameterised by a typing environment $\Gamma$. The concretisation of $E^\sharp$ is the set of concrete environments that are typed by $\Gamma$ and are the point-wise concretisations of $E^\sharp$. The concretisation of top and bottom of type $\tau$ are the sets $V^\sharp(\tau)$ and $\emptyset_\tau$ respectively.

Concretisation functions for unspecified types must be provided. In the case of While, the concretisation function for `ident` and `lit` is immediate as they are flat lattices. The concretisation function for an interval $i$ is the set of integers it contains. The concretisation of an abstract store $\sigma^\sharp$ is the point-wise concretisation of integers.

**Definition 23** *The concretisations of unspecified types for the While skeletal semantics are defined as:*

$$\gamma_{\texttt{ident}}(\mathcal{A}, x) = \{x\}$$

$$\gamma_{\texttt{ident}}(\mathcal{A}, l) = \{l\}$$

$$\gamma_{\texttt{int}}(\mathcal{A}, i) = \{\, n \mid n \in i \,\}$$

$$\gamma_{\texttt{store}}(\mathcal{A}, \sigma^\sharp) = \left\{ \sigma \;\middle|\; \mathrm{dom}\,\sigma = \mathrm{dom}\,\sigma^\sharp \wedge \forall x \in \mathrm{dom}\,\sigma, \sigma(x) \in \gamma_{\texttt{int}}(\sigma^\sharp(x)) \right\}$$

In this case, the concretisation functions do not depend on the state of the abstract interpretation. An example of an analysis for another language where the concretisation functions depend on the AI-state will be presented in the Section 5. Similarly to the

$$\Pi \in \text{CSTACKS} = (\text{VAR} \times \text{ABSTSTATE} \times \text{VAL}^\sharp)^*$$

$$\frac{}{\varepsilon \in \Pi} \qquad \frac{\mathcal{A} \in \texttt{AbstState} \qquad \textbf{val}\ f : \tau_1 \to \tau_2 = t \in \mathbb{S} \qquad v \in V^\sharp(\tau_1) \qquad \pi \in \Pi}{(f, \mathcal{A}, v) :: \pi \in \Pi}$$

Figure 4.10: Inductive Definition of Callstacks

comparison and union definitions, the concretisation for top and bottom values are already defined for unspecified values. Another property that must be true to prove the correctness of the analysis is the monotonicity of the concretisation functions.

**Definition 24** *A concretisation function $\gamma_\tau$ is* monotonic *if and only if for any $v_1^\sharp \sqsubseteq_\tau^\sharp v_2^\sharp$ and $\mathcal{A}_1 \sqsubseteq^\sharp \mathcal{A}_2$. we have $\gamma_\tau(\mathcal{A}_1, v_1^\sharp) \subseteq \gamma_\tau(\mathcal{A}_2, v_2^\sharp)$.*

**Lemma 4** *$\gamma_{\texttt{ident}}$, $\gamma_{\texttt{lit}}$, $\gamma_{\texttt{int}}$ and $\gamma_{\texttt{store}}$ are monotonic.*

We now relate these notions with two properties. For each of them, we show that if the property is satisfied for the instantiation of unspecified types, then it holds for every type.

**Lemma 5** *If for all unspecified types $\tau$, $\gamma_\tau$ is monotonic, then for all $\tau$, $\gamma_\tau$ is also monotonic.*

**Proof 2** *By induction on type $\tau$. Full proof in Chapter A.*

## 4.5 Callstack to Ensure Termination

The abstract interpretation maintains a callstack of specified function calls. It is used for loop detection and to prevent infinite computations. The callstack is inspected at each call to a specified function to detect identical nested calls and stops the computation, as a fixpoint has been reached. Callstacks are ordered lists of frames. A frame is a tuple of the name of the function, the AI-state, and the parameter of the function when it was called.

$$\vdash \_ + \_ \mapsto^\sharp \_ \rightsquigarrow \_ \in \mathcal{P}\left((\textsc{Env}^\sharp \times \textsc{Pattern} \times \textsc{Val}^\sharp) \times \textsc{Env}^\sharp\right)$$

$$\frac{}{\text{ProgTypes}, \mathbf{prog} \vdash \xi + \_ \mapsto^\sharp v \rightsquigarrow \xi} \text{ A-Asn-Wildcard}$$

$$\frac{}{\text{ProgTypes}, \mathbf{prog} \vdash \xi + x \mapsto^\sharp v \rightsquigarrow \left\{ E^\sharp + x \mapsto v^\sharp \;\middle|\; E^\sharp \in \xi \right\}} \text{ A-Asn-Var}$$

$$\frac{\text{ProgTypes}, \mathbf{prog} \vdash \xi + p \mapsto^\sharp v \rightsquigarrow \xi'}{\text{ProgTypes}, \mathbf{prog} \vdash \xi + C\,p \mapsto^\sharp C\,v \rightsquigarrow \xi'} \text{ A-Asn-Constr}$$

$$\frac{\begin{array}{c}(v_1, .., v_n) \in t \qquad \text{ProgTypes}, \mathbf{prog} \vdash \xi + p_1 \mapsto^\sharp v_1 \rightsquigarrow \xi_{v_1} \\ .. \qquad \text{ProgTypes}, \mathbf{prog} \vdash \xi_{(v_1,..,v_{n-1})} + p_n \mapsto^\sharp v_n \rightsquigarrow \xi_{(v_1,..,v_n)}\end{array}}{\text{ProgTypes}, \mathbf{prog} \vdash \xi + (p_1, .., p_n) \mapsto^\sharp t \rightsquigarrow \bigcup_{(v_1,..,v_n) \in t} \xi_{(v_1,..,v_n)}} \text{ A-Asn-Tuple}$$

$$\frac{\begin{array}{c}\mathbf{prog}@\,\text{pp} = C\,(v'_1, .., v'_n) \\ C : (\tau_1 \times .. \times \tau_n, \tau) \qquad v_j = \mathbf{if}\ \tau_j \in \text{ProgTypes}\ \mathbf{then}\ \text{pp} \cdot j\ \mathbf{else}\ v'_j \\ \text{ProgTypes}, \mathbf{prog} \vdash \xi + p \mapsto^\sharp \{(v_1, .., v_n)\} \rightsquigarrow \xi'\end{array}}{\text{ProgTypes}, \mathbf{prog} \vdash \xi + C\,p \mapsto^\sharp \text{pp} \rightsquigarrow \xi'} \text{ A-Asn-Unfold}$$

Figure 4.11: Abstract Pattern Matching

## 4.6 Extension of Environments

The extension of environments, or pattern matching, is presented on Figure 4.11. The pattern matching is parameterised by ProgTypes, the set of program types, and the analysed program **prog**. A set of environments $\xi$ is matched with a pattern and an abstract value. The rule A-Asn-Wildcard matches a wildcard pattern with an abstract value. The set of abstract environments is unchanged. The rule A-Asn-Variable matches a variable with an abstract value. Each abstract environment $E^\sharp$ of $\xi$ is extended with a binding where the variable maps to the abstract value. The rule A-Asn-Constr matches a constructor $C$ applied to a pattern $p$ with the constructor $C$ applied to an abstract value $v^\sharp$. The extension of environments is done by matching the pattern $p$ with the value $v^\sharp$. The rule A-Asn-Tuple matches a tuple of patterns $(p_1, .., p_n)$ with a tuple $t^\sharp$. An abstract tuple is a set of tuple of abstract values. Not to lose too much precision, each abstract environment is extended for each tuple of abstract values in the abstract tuple. A tuple of abstract values $(v_1^\sharp, .., v_n^\sharp)$ of $t^\sharp$ is matched with the pattern $(p_1, .., p_n)$ by successively matching each $v_i^\sharp$ with $p_i$. The rule A-Asn-Unfold matches a constructor $C$ applied to a pattern $p$ with a program point. The program point must be unfolded, as seen in Chapter 3. The sub-program at program point pp is expected to have the form $C(v_1'^\sharp, .., v_n'^\sharp))$. The tuple $(v_1'^\sharp, .., v_n'^\sharp)$ is modified in $(v_1^\sharp, .., v_n^\sharp)$ where $v_i^\sharp$ is replaced by its program point if its type is in ProgTypes, otherwise it is equal to $v_i'^\sharp$.

## 4.7 Abstract Interpretation of Skel

One ingredient is missing to get an abstract interpretation of the While language: the specifications of the unspecified terms. The instantiations of the unspecified terms are allowed to modify the AI-state. It is not useful here, but it is used in the abstract interpretation of $\lambda$-calculus, in the next section.

$$[\![litToInt]\!]^\sharp(\mathcal{A}, n) = \mathcal{A}, [n; n]$$
$$[\![add]\!]^\sharp(\mathcal{A}, [n_1; n_2], [m_1; m_2]) = \mathcal{A}, [n_1 + m_1; n_2 + m_2]$$
$$[\![read]\!]^\sharp(\mathcal{A}, x, s^\sharp) = \mathcal{A}, s^\sharp(x)$$
$$[\![write]\!]^\sharp(\mathcal{A}, x, s^\sharp, [n_1; n_2]) = \mathcal{A}, s^\sharp[x \mapsto [n_1; n_2]]$$

We recall that if $x$ is an unspecified term of type $\tau$, then $[\![x]\!]^{\texttt{ppoint}}$ is a **set of concrete**

$$\Downarrow_t \in \mathcal{P}\left((\textsc{Env}^\sharp \times \textsc{Term}) \times \textsc{Val}^\sharp\right)$$

$$\Downarrow_S^\sharp \in \mathcal{P}\left((\textsc{CStacks} \times \textsc{AbstState} \times \textsc{Env}^\sharp \times \textsc{Skeleton}) \times (\textsc{Val}^\sharp \times \textsc{AbstState})\right)$$

$$\frac{E^\sharp(x) = v^\sharp}{E^\sharp, x \Downarrow_t v^\sharp}\ \textsc{Var} \qquad \frac{\mathbf{val}\ f : \tau_1 \to \tau_2[= t] \in \mathbb{S}}{E^\sharp, f \Downarrow_t (\{f\}, \{\ \})}\ \textsc{TermClos}$$

$$\frac{\mathbf{val}\ x : \tau = t \in \mathbb{S} \qquad \emptyset, t \Downarrow_t v^\sharp \qquad \mathrm{na}(\tau)}{E^\sharp, x \Downarrow_t v^\sharp}\ \textsc{TermSpec}$$

$$\frac{\mathbf{val}\ x : \tau \in \mathbb{S} \qquad \mathrm{na}(\tau)}{E^\sharp, x \Downarrow_t [\![x]\!]^\sharp}\ \textsc{TermUnspec} \qquad \frac{E^\sharp, t \Downarrow_t v^\sharp}{E^\sharp, \mathrm{C}\, t \Downarrow_t \mathrm{C}\, v^\sharp}\ \textsc{Alg}$$

$$\frac{\forall i \in \{1..n\}\ \ E^\sharp, t_i \Downarrow_t v_i^\sharp}{E^\sharp, (t_1, .., t_n) \Downarrow_t \left\{(v_1^\sharp, .., v_n^\sharp)\right\}}\ \textsc{Tuple} \qquad \frac{(\Gamma, \lambda p : \tau \cdot S) \in \mathrm{Funs}(\mathbb{S})}{\pi, E^\sharp, \lambda p : \tau \cdot S \Downarrow_S^\sharp (\{\ \}, \left\{(p, S, E^\sharp)\right\})}\ \textsc{Clos}$$

$$\frac{\forall i \in \{1..n\}\ \ \pi, \mathcal{A}, E^\sharp, S_i \Downarrow_S^\sharp v_i^\sharp, \mathcal{A}_i}{\pi, \mathcal{A}, E^\sharp, (S_1..S_n) \Downarrow_S^\sharp \sqcup^\sharp v_i^\sharp, \sqcup^\sharp \mathcal{A}_i}\ \textsc{Branch}$$

$$\frac{\begin{array}{c}\pi, \mathcal{A}, E^\sharp, S_1 \Downarrow_S^\sharp v^\sharp, \mathcal{A}' \qquad \mathrm{ProgTypes}, \mathbf{prog} \vdash \left\{E^\sharp\right\} + p \mapsto v^\sharp \rightsquigarrow \left\{E_1^\sharp, .., E_n^\sharp\right\} \\ \forall i \in \{1..n\}\ \ \pi, \mathcal{A}', E_i^\sharp, S_2 \Downarrow_S^\sharp w_i^\sharp, \mathcal{A}_i\end{array}}{\pi, \mathcal{A}, E^\sharp, \mathbf{let}\ p\ =\ S_1\ \mathbf{in}\ S_2 \Downarrow_S^\sharp \sqcup^\sharp w_i^\sharp, \sqcup^\sharp \mathcal{A}_i}\ \textsc{LetIn}$$

$$\frac{\begin{array}{c}E^\sharp, t_0 \Downarrow_t (F_1, F_2) \\ E^\sharp, t_1 \Downarrow_t v^\sharp \qquad F_1 \cup F_2 = \bigcup \{f_i\} \qquad \forall i \in \{1..n\}\ \ \pi, \mathcal{A}, f_i\ v^\sharp \Downarrow_{app} v_i^\sharp, \mathcal{A}_i\end{array}}{\pi, \mathcal{A}, E^\sharp, t_0\ t_1 \Downarrow_S^\sharp \sqcup_i v_i^\sharp, \sqcup_i \mathcal{A}_i}\ \textsc{App}$$

Figure 4.12: Abstract Interpretation of Skeletons and Terms

$$\Downarrow_{app} \in \mathcal{P}\left((\textsc{CStacks} \times \textsc{AbstState} \times \textsc{Fun}^\sharp \times \textsc{Val}^\sharp) \times (\textsc{Val}^\sharp \times \textsc{AbstState})\right)$$

$$\mathbf{update}_f^{in} \in (\textsc{AbstState} \times V^\sharp(\tau_1)) \to (\textsc{AbstState} \times V^\sharp(\tau_1)) \qquad f : \tau_1 \to \tau_2$$
$$\mathbf{update}_f^{out} \in (\textsc{AbstState} \times V^\sharp(\tau_1) \times V^\sharp(\tau_2)) \to (\textsc{AbstState} \times V^\sharp(\tau_2)) \quad f : \tau_1 \to \tau_2$$

$$\frac{\begin{array}{c}\text{ProgTypes}, \mathbf{prog} \vdash \left\{E^\sharp\right\} + p \mapsto^\sharp v^\sharp \rightsquigarrow \left\{E_1^\sharp, .., E_m^\sharp\right\} \\ \forall E_i^\sharp \in \left\{E_1^\sharp, .., E_m^\sharp\right\}\ \pi, \mathcal{A}, E_i^\sharp, S \Downarrow_S w_i^\sharp, \mathcal{A}_i\end{array}}{\pi, \mathcal{A}, (p, S, E^\sharp)\ v^\sharp \Downarrow_{app} \sqcup^\sharp w_i^\sharp, \sqcup^\sharp \mathcal{A}_i'} \textsc{Clos}$$

$$\frac{\begin{array}{c}\mathbf{val}\ f : \tau_1 \to \tau_2 = t \in \mathbb{S} \\ \emptyset, t \Downarrow_t w^\sharp \qquad \mathbf{update}_f^{in}(\mathcal{A}, v^\sharp) = \mathcal{A}_1, v'^\sharp \qquad (f, \mathcal{A}_1, v'^\sharp) \notin \pi \\ (f, \mathcal{A}_1, v'^\sharp) :: \pi, \mathcal{A}_1, w^\sharp\ v'^\sharp \Downarrow_{app} u^\sharp, \mathcal{A}_2 \qquad \mathbf{update}_f^{out}(\mathcal{A}_2, v'^\sharp, u^\sharp) = \mathcal{A}_3, u'^\sharp\end{array}}{\pi, \mathcal{A}, f\ v^\sharp \Downarrow_{app} u'^\sharp, \mathcal{A}_3} \textsc{Spec}$$

$$\frac{\mathbf{val}\ f : \tau_1 \to \tau_2 = t \in \mathbb{S} \qquad \mathbf{update}_f^{in}(\mathcal{A}, v^\sharp) = \mathcal{A}_1, v'^\sharp \qquad (f, \mathcal{A}_1, v'^\sharp) \in \pi}{\pi, \mathcal{A}, f\ v^\sharp \Downarrow_{app} \bot, \mathcal{A}_1} \textsc{Spec-Loop}$$

$$\frac{\mathbf{val}\ f : \tau_1 \to \tau_2 \in \mathbb{S} \qquad [\![f]\!]^\sharp(\mathcal{A}, v^\sharp) = w^\sharp, \mathcal{A}'}{\pi, \mathcal{A}, f\ v^\sharp \Downarrow_{app} w^\sharp, \mathcal{A}'} \textsc{Unspec}$$

Figure 4.13: Abstract Interpretation: Application

**interpretations for** $x$: $[\![x]\!]^{\texttt{ppoint}} \in \mathcal{P}\left(V^{\texttt{ppoint}}(\tau)\right)$

**Definition 25** *Let $x$ be an unspecified term of type $\tau$, such that* $\mathrm{na}(\tau)$*. We say that $[\![x]\!]^\sharp$ is a* ***sound approximation*** *of $[\![x]\!]^{ppoint}$ if and only if:*

$$\forall \mathcal{A},\ [\![x]\!]^{ppoint} \subseteq \gamma(\mathcal{A}, [\![x]\!]^\sharp)$$

**Definition 26** *Let $f$ be an unspecified term of type $\tau_1 \to \tau_2$. We say that $[\![f]\!]^\sharp$ is a* ***sound approximation*** *of $[\![f]\!]^{ppoint}$ if and only if $v \in V^{ppoint}(\tau_1)$, $v^\sharp \in V^\sharp(\tau_1)$, and for all AI-state $\mathcal{A}$, then*

$$\left.\begin{array}{r} v \in \gamma_{\tau_1}(\mathcal{A}, v^\sharp) \\ [\![f]\!]^\sharp(\mathcal{A}, v^\sharp) = \mathcal{A}', w^\sharp \end{array}\right\} \implies [\![f]\!]^{ppoint}(v) \subseteq \gamma_{\tau_2}(\mathcal{A}', w^\sharp)$$

**Lemma 6** *The abstract instantiations of the unspecified terms for While are sound approximation of the concrete instantiations of the unspecified terms.*

The abstract interpretation of skeletons is a relation and is defined on Figure 4.12. The abstract interpretation of skeletons is similar to the big-step interpretation. The Var rule evaluates a variable by fetching the corresponding value in the abstract environment. The TermClos rule evaluates a variable that is a specified or unspecified term, with arrow type whose definition is defined at top-level in the skeletal semantics. It returns an abstract function, which is a set of closures containing only one. The TermSpec rule evaluates a variable that is a specified term whose type is not an arrow, by evaluating the definition of the specified term. The TermUnspec rule evaluates a variable that is an unspecified term. The value returned is the definition provided for the given skeletal semantics. The Alg rule evaluates a constructor applied to a term by returning the constructor applied to the evaluation of the term. The Tuple rule evaluates a tuple by evaluating each component of the tuple, and returns a singleton with a tuple of abstract values. The Clos rule evaluates a closure by returning an abstract function: a singleton containing an anonymous abstract closure.

The Branch rule evaluates all branches, and then the results and abstract states are joined. The LetIn rule evaluates a let binding by first evaluating the skeleton to be bound, it gives an abstract value. Then, the returned value is used with the pattern to extend the current abstract environment. The extension of environment returns a set of environments. The body of the let binding is evaluated in each environment. The result is the abstract union of the obtained abstract values and the abstract union of the obtained

AI-state. The App rule evaluates an application by evaluating both terms: the function and its parameter. The evaluation of the function returns a set of anonymous and named closures, the application of each closure to the parameter is evaluated by the $\Downarrow_{app}$ relation.

The $\Downarrow_{app}$ relation evaluates the application of an anonymous or named closure to an abstract value. It is parameterised by two user-provided functions that are dependent on the skeletal semantics: $\mathbf{update}^{in}$ and $\mathbf{update}^{out}$. The update functions are used to modify the AI-state before and after a call to a specified function to ensure invariant. The Clos rule evaluates the application of an anonymous closure to an abstract value. The abstract environment of the closure is extended with the parameter and the pattern of the closure. With each of the obtained environments, the skeleton of the closure is evaluated. The abstract values and AI-states obtained are joined and returned. Spec rule evaluates the call to a specified function and maintains invariants. To evaluate a specified function, its definition is fetched in the skeletal semantics and evaluated giving an abstract value. Moreover, the parameter and AI-state are modified according to the input update function, in order to maintain input invariants. The stack is inspected to ensure it is not an identical nested call, which would lead to an infinite loop. Invariants depend on the analysis and the AI-state, therefore, the update functions to maintain invariants before and after a call must be provided. App rule evaluates all terms and passes a list of values to the application relation, defined in Figure 4.13. Because the abstraction of a function is a set of closures and named closures, the App-Set rule evaluates each one individually. The Clos rule evaluates the body of the function $S$ in all abstract environments returned by the matching of the pattern against the argument.

The update functions must respect the following monotonicity constraints to ensure soundness:

**Definition 27** *Let val $f : \tau_1 \to \tau_2 = t \in \mathbb{S}$. The update functions are said to be monotonic if and only if:*

$$\boldsymbol{update}_f^{in}(\mathcal{A}, arg^\sharp) = args'^\sharp, \mathcal{A}' \implies arg^\sharp \sqsubseteq^\sharp_{\tau_1} arg'^\sharp \wedge \mathcal{A} \sqsubseteq^\sharp \mathcal{A}'$$
$$\boldsymbol{update}_f^{out}(\mathcal{A}, arg^\sharp, res^\sharp) = res'^\sharp, \mathcal{A}' \implies res^\sharp \sqsubseteq^\sharp_{\tau_2} res'^\sharp \wedge \mathcal{A} \sqsubseteq^\sharp \mathcal{A}'$$

The update functions of While must maintain the AI-state mapping program points and input/output tags to abstract stores. The property that should be ensured for a value analysis is that the AI-state contains a correct approximation of the stores before and after

the evaluation of each sub-program of the analysed program.

$$\mathbf{update}^{in}_{\texttt{eval\_stmt}}(\mathcal{A}, (s^\sharp_i, \text{pp})) = \mathcal{A}[(\text{pp}, \texttt{In}) \mapsto s^\sharp], (s^\sharp, \text{pp}) \text{ with } s^\sharp = s^\sharp_i \sqcup^\sharp \mathcal{A}(\text{pp}, \texttt{In})$$
$$\mathbf{update}^{out}_{\texttt{eval\_stmt}}(\mathcal{A}, (s^\sharp_i, \text{pp}), s^\sharp_o) = \mathcal{A}[(\text{pp}, \texttt{Out}) \mapsto s^\sharp], (s^\sharp, \text{pp}) \text{ with } s^\sharp = s^\sharp_o \sqcup^\sharp \mathcal{A}(\text{pp}, \texttt{Out})$$

$\mathbf{update}^{in}_{\texttt{eval\_stmt}}(\mathcal{A}, (s^\sharp_i, \text{pp}))$ updates the input abstract store associated to program point pp for a greater abstract store that contains $s^\sharp_i$, and the call to `eval_stmt` is done with this new abstract store. $\mathbf{update}^{out}_{\texttt{eval\_stmt}}(\mathcal{A}, (s^\sharp_i, \text{pp}), s^\sharp_o)$ makes a similar change to the AI-state for the output store. The update functions for `eval_expr` do not change the argument, the result, nor the AI-state.

$$\mathbf{update}^{in}_{\texttt{eval\_expr}}(\mathcal{A}, (s^\sharp, e)) = \mathcal{A}, (s^\sharp, e)$$
$$\mathbf{update}^{out}_{\texttt{eval\_expr}}(\mathcal{A}, (s^\sharp, e), i^\sharp) = \mathcal{A}, i^\sharp$$

**Lemma 7** *The update functions for While previously defined are monotonic.*

**Example 1** *Take* ***prog*** $\equiv$ `x = 0; while (x < 3) x := x + 1;` (***prog*** *is a value of the ADT* `stmt`*, but we present it in a more readable syntax).*

*One can show that:*

$$E_0, \texttt{eval\_stmt}\ (s, t) \Downarrow_S \{x \mapsto 3\}$$
$$\varepsilon, \mathcal{A}_0, E^\sharp_0, \texttt{eval\_stmt}\ (s, t) \Downarrow^\sharp_S \{x \mapsto [0, +\infty]\}, \mathcal{A}$$

## 4.8 Correctness of the Abstract Interpretation

Our methodology aims to define mathematically correct abstract interpreters from Skeletal Semantics. In this section, we present a theorem stating that the abstract interpreter of Skel computes a correct approximation of the big-step semantics of Skel. We can prove the correctness of the abstract interpretation when specified functions do not return values of inductive types. This is discussed in the Appendix A.2.

**Theorem 3** *Let $\mathbb{S}$ be a Skeletal Semantics with unspecified terms $Te$ and unspecified types $Ty$, and let $E$ and $E^\sharp$ be a concrete and abstract environment, respectively. Let $\mathcal{A}_0$ be an AI-state. Suppose*

- $\forall x \in Te$, $[\![x]\!]^\sharp$ *is a sound approximation of* $[\![x]\!]^{ppoint}$.

- $\forall \tau \in Ty$, $\gamma_\tau$ *is monotonic.*
- $\boldsymbol{update}^{in}$ *and* $\boldsymbol{update}^{out}$ *are monotonic.*
- *Specified functions do not return values of inductive types.*

*Then:*

$$\left.\begin{array}{c} E \in \gamma_\Gamma(\mathcal{A}_0, E^\sharp) \\ E, S \Downarrow_S v \\ \varepsilon, \mathcal{A}_0, E^\sharp, S \Downarrow_S^\sharp v^\sharp, \mathcal{A} \end{array}\right\} \implies v \in \gamma(\mathcal{A}, v^\sharp)$$

**Proof 3** *The proof is done by defining a relation between concrete and abstract values* $v \sim_{\mathcal{A}} v^\sharp$ *such that the AST of* $v$ *and the AST of* $v^\sharp$ *have the same shape and for every leaf* $v'$ *of* $v$ *with unspecified type* $\tau$*, and its corresponding leaf* $v'^\sharp$ *of* $v^\sharp$*, then* $v' \in \gamma_\tau(\mathcal{A}, v'^\sharp)$*. The relation* $\sim$ *is defined by induction, and one can prove that* $v \sim_{\mathcal{A}} v^\sharp \implies v \in \gamma(\mathcal{A}, v^\sharp)$ *by induction on the derivation tree of* $v \sim_{\mathcal{A}} v^\sharp$*.*

*Supposing that:* $E \in \gamma_\Gamma(\mathcal{A}_0, E^\sharp)$ *and* $E, S \Downarrow_S v$ *and* $\varepsilon, \mathcal{A}_0, E^\sharp, S \Downarrow_S^\sharp v^\sharp, \mathcal{A}$*, one can prove by induction on the concrete derivation tree that* $v \sim_{\mathcal{A}} v^\sharp$*, which implies the theorem.*

Therefore, to prove the soundness of the analysis, it is sufficient to prove that the abstract instantiation of terms are sound approximation of the concrete ones, and that the update functions and concretisation functions are monotonic.

Let $\sigma_0 \in V^{\texttt{ppoint}}(\texttt{store})$ and $\sigma_0^\sharp \in V^\sharp(\texttt{store})$ be the concrete and abstract stores with empty domains. $\epsilon$ is the program point of the root of the analysed program, **prog**. Let $E_0 = \{s \mapsto \sigma_0, t \mapsto \epsilon\}$ and $E_0^\sharp = \left\{s \mapsto \sigma_0^\sharp, t \mapsto \epsilon\right\}$ be a concrete and abstract Skel environments. Let $\mathcal{A}_0$ be the empty mapping from program points and flow tags (`In` or `Out`) to abstract stores.

**Lemma 8**

$$\sigma_0 \in \gamma_{store}(\mathcal{A}_0, \sigma_0^\sharp)$$

**Proof 4** *By definition of* $\gamma_{store}$ *and because* $\sigma_0$ *and* $\sigma_0^\sharp$ *have an empty domain.*

**Lemma 9** *Let* $\Gamma = \{s \mapsto \texttt{store}, t \mapsto \texttt{stmt}\}$*.*

$$E_0 \in \gamma_\Gamma(\mathcal{A}, E_0^\sharp)$$

**Proof 5** $E_0$ *and* $E_0^\sharp$ *have the same domain,* $E_0(t) \in \gamma(\mathcal{A}_0, E_0^\sharp(t))$*, and* $E_0(s) \in \gamma(\mathcal{A}_0, E_0^\sharp(s))$ *by Lemma 8. Therefore, the Lemma is true by definition of* $\gamma_\Gamma$*.*

The abstract interpreter computes an abstract closure that is a correct approximation of the concrete closure returned by the big-step semantics.

**Theorem 4**

$$\begin{array}{c} E_0, \texttt{eval\_stmt}\ (s,t) \Downarrow_S \sigma \\ \varepsilon, \mathcal{A}_0, E_0^\sharp, \texttt{eval\_stmt}\ (s,t) \Downarrow_S^\sharp \sigma^\sharp, \mathcal{A} \end{array} \implies \sigma \in \gamma(\mathcal{A}, \sigma^\sharp)$$

**Proof 6** *The proof uses Theorem 3. The concretisation functions are monotonic (Lemma 4 and Lemma 5). The abstract instantiations of the unspecified terms are sound approximations of the concrete instantiations of the unspecified terms (Lemma 6). The update functions are monotonic (Lemma 7). Furthermore, $E_0 \in \gamma_\Gamma(\mathcal{A}_0, E_0^\sharp)$ (Lemma 8). Therefore, Theorem 3 applies, and by instantiating it with $S =$ `eval_stmt` $(s,t)$, we obtain the desired result.*

**Example 2** *Let* ***prog*** $\equiv$ `x = 0; while (x < 3) x := x + 1;`.

*We have seen previously that:*

$$\begin{array}{c} E_0, \texttt{eval\_stmt}\ (s,t) \Downarrow_S \{x \mapsto 3\} \\ \varepsilon, \mathcal{A}_0, E_0^\sharp, \texttt{eval\_stmt}\ (s,t) \Downarrow_S^\sharp \{x \mapsto [0, +\infty]\}, \mathcal{A} \end{array}$$

*By applying Theorem 4, it is true that:*

$$\{x \mapsto 3\} \in \gamma(\mathcal{A}, \{x \mapsto [0, \infty]\})$$

One can notice that the abstract interpreter returns an imprecise result. The negation of the condition after the While loop is not used to refine the abstract store. We plan to do it as future work, and is discussed in Chapter 7.

We present another example that does not terminate.
Take **prog** $\equiv$ `x = 1; while (0 < x) x := x + 1;` (the condition and the initialisation of `x` have changed).

$$\varepsilon, \mathcal{A}_0, E_0^\sharp, \texttt{eval\_stmt}\ (s,t) \Downarrow_S^\sharp \{x \mapsto [0, +\infty]\}, \mathcal{A}$$

In the abstract interpretation, the loop is exited and the abstraction for the start of the loop has converged in the AI-state thanks to the Unspec-Loop rule of the abstract

interpretation of Skel.

Chapter 5

# A Control Flow Analysis for $\lambda$-calculus

Our skeleton-based methodology for designing program analyses has proven capable of computing a simple value analysis for an imperative language. To show the versatility of the approach, another type of analysis, *viz.* a Control Flow Analysis [34], (CFA) for the simple $\lambda$-calculus is defined following the same methodology.

In higher-order languages, the control flow of a program cannot be obtained directly from the program syntax alone. There are many Control Flow Analyses that have been developed [28, 45] whose goal is to over-approximate the control flow of a given program.

In this Chapter, we show how to derive a CFA for $\lambda$-calculus from its skeletal semantics, using the abstract interpretation of Skel defined in the previous section.

## 5.1 Simple $\lambda$-calculus and CFA

The $\lambda$-calculus uses two syntactic categories:

$$x \in \texttt{var} \qquad \text{and} \qquad t \in \texttt{lterm} \ ::= \ x \ \mid \ \lambda x.t \ \mid \ t\ t$$

The set `var` contains variables. The `lterm` set contains the $\lambda$-terms that can either be a variable $x$, a $\lambda$-abstraction (a function), or an application $t_1\ t_2$ where $t_1$ is the function and $t_2$ is the parameter.

To precisely refer to sub-terms of a $\lambda$-term program points are added to the analysed $\lambda$-term. The program points presented here serve a similar purpose to what has been shown for skeletal semantics. The idea is to add labels to a $\lambda$-term to precisely refer to

$$\texttt{clos} = \texttt{var} \times \texttt{lterm} \times \texttt{env}$$
$$\texttt{env} = \texttt{var} \hookrightarrow \texttt{clos}$$
$$\Downarrow^{\lambda} \in \mathcal{P}\left((\texttt{env} \times \texttt{lterm}) \times \texttt{clos}\right)$$

$$\frac{}{\sigma, x \Downarrow^{\lambda} \sigma(x)}\ \textsc{Var} \qquad \frac{}{\sigma, \lambda x.t \Downarrow^{\lambda} (x, t, \sigma)}\ \textsc{Lam}$$

$$\frac{\sigma, t_0 \Downarrow^{\lambda} (x, t, \sigma') \qquad \sigma, t_1 \Downarrow^{\lambda} v_1 \qquad \sigma'' = \sigma' + x \mapsto v_1 \qquad \sigma'', t \Downarrow^{\lambda} v}{\sigma, t_0\ t_1 \Downarrow^{\lambda} v}\ \textsc{App}$$

Figure 5.1: Semantics of λ-calculus with environments

sub-programs. Formally, adding program points to a λ-term is defined as:

$$[\![x]\!]^{\text{pp}} = x^{\text{pp}}$$
$$[\![\lambda x.t]\!]^{\text{pp}} = (\lambda x.[\![t]\!]^{\text{pp}\cdot 1})^{\text{pp}}$$
$$[\![t_0\ t_1]\!]^{\text{pp}} = ([\![t_0]\!]^{\text{pp}\cdot 0}\ [\![t_1]\!]^{\text{pp}\cdot 1})^{\text{pp}}$$

Take the term $(\lambda x.x)\ (\lambda x.x)$, the annotated version of this term is:

$$[\![(\lambda x.x)\ (\lambda x.x)]\!]^{\epsilon} = ((\lambda x.x^{01})^{0}\ (\lambda x.x^{11})^{1})^{\epsilon}$$

We note $\epsilon$ the empty program point. The identity function $\lambda x.x$ appears twice in the λ-term, but the program points make it possible to differentiate them.

The semantics of λ-calculus operates with environments that map variables to values. In the pure λ-calculus there is only one kind of value: closures. A closure is of the form $(x, t, \sigma)$ and represents a function where $x$ is the variable to be bound, $t$ the body of the function and $\sigma$ the environment mapping free variables of $t$ to closures. The semantics of λ-calculus with environments is formally defined on Figure 5.1. The Var rule evaluates a variable by fetching its content in the environment. The Lam rule evaluates a λ-abstraction by returning a closure with the variable to be bound, the code of the λ-abstraction and the current environment. The App rule evaluates an application by recursively evaluating the function and its parameter. The function evaluates to a closure, a new environment is defined by extending the closure environment so that the variable

$$\dfrac{\overline{\{\,\},(\lambda x.x) \Downarrow^{\lambda} (x,x,\{\,\})} \qquad \overline{\{\,\},(\lambda y.y) \Downarrow^{\lambda} c_y} \qquad \overline{\{x \mapsto c_y\},x \Downarrow^{\lambda} c_y}}{\{\,\},(\lambda x.x)\ (\lambda y.y) \Downarrow^{\lambda} c_y}\ \textsc{App}$$

Figure 5.2: Semantics of a Simple λ-calculus Program

maps-to the result of the evaluation of the parameter. The code in the closure is evaluated with the new environment. An example of the evaluation of the term $(\lambda x.x)\ (\lambda y.y)$ is given on Figure 5.2. We set the notation $c_y \equiv (y, y, \{\,\})$. The root of the tree is the application of the identity to the identity from an empty environment. The function and the argument are computed and both return the same closures up to renaming. Then, the body of the function $x$ is evaluated in a new environment where the variable $x$ is bound to $c_y$.

We present 0-CFA for λ-calculus, a Control Flow Analysis that is context-insensitive. The analysis should give an approximation of the functions that may be called at each call-site. A call-site of a λ-term $t$ is a sub-term of $t$ which is an application, because these are the locations where a call to a function may happen. A context insensitive analysis is an analysis where the abstraction of the functions that may be called at a call-site only depends on its location in the program (its program point) and not on the calling context. Given a λ-term $t$, 0-CFA defines two maps, $C$ and $\rho$, where $C$ maps program points to sets of λ-abstractions, and $\rho$ maps variables to sets of λ-abstractions.

$$C : \texttt{ppoint} \to \{\, \lambda x.t \mid x \in \texttt{var} \wedge t \in \texttt{lterm} \,\}$$
$$\rho : \texttt{ident} \to \{\, \lambda x.t \mid x \in \texttt{var} \wedge t \in \texttt{lterm} \,\}$$

The set $C(\text{pp})$ contains the λ-abstractions the sub-term at program point pp may evaluate to. The set $\rho(x)$ contains the λ-abstractions that may be bound to variable $x$.

**Definition 28** *Let $t$ be an annotated λ-term. $C$ and $\rho$ must respect the following equations*

*for every sub-terms of* $t$.

$$
\begin{aligned}
(C,\rho) \vDash x^{\mathrm{pp}} &\iff \rho(x) \subseteq C(\mathrm{pp}) \\
(C,\rho) \vDash (\lambda x.t^{\mathrm{pp}\cdot 1})^{\mathrm{pp}} &\iff \left\{(\lambda x.t^{\mathrm{pp}\cdot 1})\right\} \subseteq C(\mathrm{pp}) \\
(C,\rho) \vDash (t_0^{\mathrm{pp}\cdot 0}\ t_1^{\mathrm{pp}\cdot 1})^{\mathrm{pp}} &\iff (C,\rho) \vDash t_0^{\mathrm{pp}\cdot 0} \\
&\qquad (C,\rho) \vDash t_1^{\mathrm{pp}\cdot 1} \\
&\qquad \forall (\lambda x.t^{\mathrm{pp}'}) \in C(\mathrm{pp}\cdot 0), \quad (C,\rho) \vDash t^{\mathrm{pp}'} \\
&\qquad\qquad\qquad\qquad C(\mathrm{pp}\cdot 1) \subseteq \rho(x) \wedge C(\mathrm{pp}') \subseteq C(\mathrm{pp})
\end{aligned}
$$

- If $t@\,\mathrm{pp} = x^{\mathrm{pp}}$
  All values the variable $x$ can be bound to are possible results when evaluating the sub-term at program point pp, explaining the equation $\rho(x) \subseteq C(\mathrm{pp})$

- If $t@\,\mathrm{pp} = (\lambda x.t)^{\mathrm{pp}}$
  It is clear that $C(\mathrm{pp})$ must contain the $\lambda$-abstraction $\lambda x.t$

- If $t@\,\mathrm{pp} = (t_0\ t_1)^{\mathrm{pp}}$
  The mappings $C$ and $\rho$ must be valid for $t_0$ and $t_1$. Every $\lambda$-abstraction in $C(\mathrm{pp}\cdot 0)$ is a potential function that may be called at call-site pp. In particular, if $\lambda x.t^{\mathrm{pp}'} \in C(\mathrm{pp}\cdot 0)$, then $C$ and $\rho$ must be valid for $t'$. Moreover, $x$ can be bound to any value that may come up when evaluating the parameter $t_1$, hence the equation $C(\mathrm{pp}\cdot 1) \subseteq \rho(x)$. Finally, the evaluation of $t$, the body of the function, is the result of the application, and naturally $C(\mathrm{pp}') \subseteq C(\mathrm{pp})$.

Take $t \equiv (\lambda f.(f\ (\lambda x.x))^{\mathrm{pp}_x}\ (f\ (\lambda y.y))^{\mathrm{pp}_y})\ (\lambda g.g^{\mathrm{pp}_g})$, we did not put every program point to keep the notation lightweight. The most precise result of a 0-CFA analysis is presented in the table below. Because $C$ and $\rho$ have the same co-domain, we write $C + \rho$ the function with domain $\texttt{ppoint} \cup \texttt{var}$ such that it is equal to $C$ on domain `ppoint`, and $\rho$ on domain `var`.

| $C+\rho$ | |
|---|---|
| $x$ | $\{\lambda y.y\}$ |
| $y$ | $\{\}$ |
| $g$ | $\{\lambda x.x, \lambda y.y\}$ |
| $\mathrm{pp}_x$ | $\{\lambda x.x, \lambda y.y\}$ |
| $\mathrm{pp}_y$ | $\{\lambda x.x, \lambda y.y\}$ |
| $\epsilon$ | $\{\lambda x.x, \lambda y.y\}$ |

In this simple program, $g$ can be bound to two $\lambda$-abstractions: $\lambda x.x$ or $\lambda y.y$ depending on the calling context. Indeed, at call-site $\text{pp}_x$, $g$ is bound to $\lambda x.x$ and at call-site $\text{pp}_y$, $g$ is bound to $\lambda y.y$. A 0-CFA analysis does not take into account the calling context, therefore all values that $g$ can be bound to are mixed in the result of the analysis. The analysis concludes that sub-terms at program points $\text{pp}_x$ and $\text{pp}_y$ can evaluate to $\{\lambda x.x, \lambda x.y\}$ and this imprecision is carried until $C(\epsilon)$.

## 5.2 Skeletal Semantics of λ-calculus

The syntax and semantics of the $\lambda$-calculus are similar to the syntax and semantics of Skel. Indeed, Skel is essentially $\lambda$-calculus in ANF. Therefore, some terminologies are common and ambiguous, like *closures*. To avoid confusion, the $\lambda$-calculus will be referred to as the *object language*. We write *l-closures* and *l-environments* for closures and environments of the $\lambda$-calculus and simply *closures* and *environments* for closures and environments of Skel.

We propose a skeletal semantics to mechanise the semantics of $\lambda$-calculus with environment:

```
type ident
type env
type clos
type lterm =
     | Lam (ident, lterm)
     | Var ident
     | App (lterm, lterm)

val extEnv : (env, ident, clos) → env
val getEnv : (ident, env) → clos
val mkClos : (ident, lterm, env) → clos
val getClos : clos → (ident, lterm, env)

val eval ((s, l): (env, lterm)): clos =
  match l with
  | Lam (x, t) → mkClos (x, t, s)
  | Var x → getEnv (x, s)
```

```
| App (t1, t2) →
    let v = eval (s, t1) in
    let (x, t, s') = getClos v in
    let w = eval (s, t2) in
    let s'' = extEnv (s', x, w) in
    eval (s'', t)
```

The skeletal semantics contains four types: the identifiers (the type of variables), l-environments, l-closures and λ-terms. Only the type of λ-terms is specified. There are four unspecified functions to: extend a l-environment by adding a new binding, get the l-closure associated to an identifier in a l-environment, create a l-closure from a triple and convert a l-closure to a triple. Finally, the specified `eval` function evaluates a λ-term in a given l-environment.

We define a big-step semantics for λ-calculus as seen previously for the While language in Section 2.2. We set the set of program types to be: ProgTypes ≡ {`lterm`}. First, unspecified types and terms must be instantiated.

$$\frac{x \in \mathcal{X}}{\vdash x \in V^{\texttt{ppoint}}_{\mathbf{prog}}(\texttt{ident})}\ \textsc{Ident}$$

$$\frac{\sigma = \{x_1 \mapsto v_1, .., x_n \mapsto v_n\} \qquad \forall i \in \{1..n\}\ x_i \in V^{\texttt{ppoint}}(\texttt{ident}) \wedge v_i \in V^{\texttt{ppoint}}(\texttt{clos})}{\vdash \sigma \in V^{\texttt{ppoint}}_{\mathbf{prog}}(\texttt{env})}\ \textsc{Env}$$

$$\frac{c \in V^{\texttt{ppoint}}(\texttt{ident}) \times V^{\texttt{ppoint}}(\texttt{lterm}) \times V^{\texttt{ppoint}}(\texttt{clos})}{\vdash c \in V^{\texttt{ppoint}}_{\mathbf{prog}}(\texttt{clos})}\ \textsc{Clos}$$

The set of identifiers is defined to be some countable set. A l-environment is a partial function with a finite domain from identifiers to closures. A l-closure is a triple composed of an identifier, a λ-term and a l-environment. The next step is to define the unspecified terms.

$$[\![extEnv]\!]^{\texttt{ppoint}}(\sigma, x, c) = \sigma + x \mapsto c$$
$$[\![mkClos]\!]^{\texttt{ppoint}}(x, t, \sigma) = (x, t, \sigma)$$
$$[\![getEnv]\!]^{\texttt{ppoint}}(x, \sigma) = \sigma(x)$$
$$[\![getClos]\!]^{\texttt{ppoint}}(x, t, \sigma) = (x, t, \sigma)$$

The $[\![extEnv]\!]^{\texttt{ppoint}}$ function given a l-environment $\sigma$, a variable $x$ and a closure $c$ returns a new environment similar to $\sigma$ except that $x$ maps to $v$. The $[\![mkClos]\!]^{\texttt{ppoint}}$ function given a l-variable $x$, a $\lambda$-term $t$ and a l-environment $\sigma$ returns a closure $(x, t, \sigma)$: in this context, it is the identity function. The function $[\![getEnv]\!]^{\texttt{ppoint}}$, given a l-variable $x$ and a l-environment $\sigma$ returns the value bound to $x$ in $\sigma$. Finally, the function $[\![getClos]\!]^{\texttt{ppoint}}$ is the inverse function of $[\![mkClos]\!]^{\texttt{ppoint}}$, thus it is the identity. By combining these instantiations with the Big-step Semantics of Skel of Section 2.2.2, we obtain a Natural Semantics for $\lambda$-calculus.

**Example 3** *Take* $\textbf{\textit{prog}} \equiv App(Lam(x, Var(x)), Lam(x, Var(x)))$ *where* $x$ *is a skeletal variable. Let* $E = \{x \mapsto \texttt{x}, s \mapsto \{\ \}\}$ *where* `x` *is a variable of the* $\lambda$*-calculus. The term* ***prog*** *is a representation in Skel of the program* `(λ x.x) (λ x.x)`*. Then, one can show that:*

$$E, eval\ s\ \textbf{\textit{prog}} \Downarrow_S Clos(x, Var(x), \{\})$$

The value $Clos(x, Var(x), \{\})$ is the Skel representation of the closure $(\texttt{x}, \texttt{x}, \{\})$ of $\lambda$-calculus. We have redefined the semantics of $\lambda$-calculus with environment from its skeletal semantics.

## 5.3 A CFA for λ-calculus using Skeletal Semantics

A concrete semantics for the $\lambda$-calculus has been defined from its skeletal semantics. We present a CFA analysis obtained using the abstract interpretation of Skel that is correct with respect to the concrete semantics. First, we define the abstractions of our analysis by instantiating the unspecified types and the AI-state. Our first abstraction is to define the set of program types: ProgTypes = {`lterm`}. As a consequence, all $\lambda$-terms will be replaced by program points in the abstract interpretation. Therefore, the abstractions of $\lambda$-terms are relative to the program being analysed, which we call **prog**. The identifiers are abstracted by a flat lattice, where $\mathcal{X}$ is a countable set.

$$\frac{x \in \mathcal{X}}{x \in V^\sharp(\texttt{ident})}\ \textsc{Ident}$$

The analysis computes an abstraction of the l-closures each sub-term of the program can evaluate to. An abstract l-closure $c^\sharp$ is a set of program points, such that $\forall\, \text{pp} \in$

$c^\sharp$, $\mathbf{prog}@\,\mathrm{pp} = Lam(x,t)$: each program point in the abstract closure maps-to a λ-abstraction in the analysed program. One consequence is that $x$ is at program point $\mathrm{pp}\cdot 0$, and $t$ is at program point $\mathrm{pp}\cdot 1$.

$$\frac{c^\sharp \in \mathcal{P}\,(\texttt{ppoint}) \qquad \mathrm{pp} \in c^\sharp \implies \mathbf{prog}@\,\mathrm{pp} = Lam(x,t)}{c^\sharp \in V^\sharp(\texttt{clos})}\ \textsc{Clos}$$

In our definition of abstract l-closures, each program point maps-to a λ-abstraction $Lam(x,t)$, but a concrete closure also contains a l-environment $\sigma$, that cannot be reconstructed from an abstract closure. We choose to put the abstract l-environments in the AI-state:

$$\mathcal{A} : \texttt{ppoint} \to V^\sharp(\texttt{env})$$

Therefore, for each $\mathrm{pp} \in c^\sharp$, where $c^\sharp$ is an abstract closure, the associated abstract environment is $\mathcal{A}(\mathrm{pp})$. The abstract environment $\mathcal{A}(\mathrm{pp})$ and all the λ-abstractions $Lam(x,t)$ such that $\exists\,\mathrm{pp} \in c^\sharp\ \mathbf{prog}@\,\mathrm{pp} = Lam(x,t)$ are abstractions of the closures that can be obtained when evaluating the sub-term at program point pp.

An abstract environment is a partial function with a finite domain from identifiers to abstract closures.

$$\frac{\sigma^\sharp = \left\{x_1 \mapsto c_1^\sharp, .., x_n \mapsto c_n^\sharp\right\} \qquad \forall i \in \{1..n\}\ x_i \in V^\sharp(\texttt{ident}) \wedge c_i^\sharp \in V^\sharp(\texttt{clos})}{\sigma^\sharp \in V^\sharp(\texttt{env})}\ \textsc{Env}$$

The lattice for $V^\sharp(\texttt{ident})$ is the flat lattice. The lattice for $V^\sharp(\texttt{clos})$ is the set lattice. $V^\sharp(\texttt{env})$ is the lattice obtained by point-wise extension of the $V^\sharp(\texttt{clos})$. We do not detail the formal definitions of the abstract unions and abstract comparisons.

The concretisation functions of the unspecified types are defined as:

$$\gamma_{\texttt{env}}(\mathcal{A},\sigma^\sharp) = \left\{\, \sigma \in V^{\texttt{ppoint}}(\texttt{env}) \;\middle|\; \mathrm{dom}\,\sigma = \mathrm{dom}\,\sigma^\sharp \wedge \forall x \in \mathrm{dom}\,\sigma, \sigma(x) \in \gamma_{\texttt{clos}}(\mathcal{A},\sigma^\sharp(x)) \,\right\}$$

$$\gamma_{\texttt{ident}}(\mathcal{A},i) = \{i\} \qquad \gamma_{\texttt{ident}}(\mathcal{A},\bot) = \{\} \qquad \gamma_{\texttt{ident}}(\mathcal{A},\top) = V^{\texttt{ppoint}}(\texttt{ident})$$

$$\gamma_{\texttt{clos}}(\mathcal{A},c^\sharp) = \left\{ (x,\mathrm{pp}\cdot 1,\sigma) \in V^{\texttt{ppoint}}(\texttt{clos}) \;\middle|\; \mathrm{pp} \in c^\sharp \wedge \begin{array}{l} \mathbf{prog}@\,\mathrm{pp} = Lam(x,\_) \\ \sigma \in \gamma_{\texttt{env}}(\mathcal{A},\mathcal{A}(\mathrm{pp})) \end{array} \right\}$$

**Lemma 10** *The concretisation functions of the unspecified types are monotonic in both*

*their arguments.*

$$[\![extEnv]\!]^\sharp(\mathcal{A}, \sigma^\sharp, x, v^\sharp) = \mathcal{A}, \sigma^\sharp + x \mapsto v^\sharp$$
$$[\![getEnv]\!]^\sharp(\mathcal{A}, x, \sigma^\sharp) = \mathcal{A}, \sigma^\sharp(x)$$
$$[\![mkClos]\!]^\sharp(\mathcal{A}, x, \mathrm{pp}\cdot 1, \sigma^\sharp) = \mathcal{A}[\mathrm{pp} \leftarrow \sigma^\sharp \sqcup^\sharp \mathcal{A}(\mathrm{pp})], \{\mathrm{pp}\}$$
$$[\![getClos]\!]^\sharp(\mathcal{A}, c^\sharp) = \mathcal{A}, \left\{ (x, \mathrm{pp}\cdot 1, \sigma^\sharp) \;\middle|\; \mathrm{pp} \in c^\sharp \wedge \mathbf{prog}@\,\mathrm{pp} = Lam(x, \_) \wedge \mathcal{A}(\mathrm{pp}) = \sigma^\sharp \right\}$$

We instantiate the unspecified terms. The $[\![extEnv]\!]^\sharp$ and $[\![getEnv]\!]^\sharp$ are identical to those for the concrete interpretation. The $[\![mkClos]\!]^\sharp$ function is called when evaluating a λ-abstraction $Lam(x, \mathrm{pp}\cdot 1)$ in abstract environment $\sigma^\sharp$. The AI-state is modified: the abstract environment at program point pp in the AI-state must be bigger than $\sigma^\sharp$, as it is an abstraction of the environments used to evaluate the sub-term a program point pp. The abstract l-closure returned is the singleton {pp}, because this program point maps-to the λ-abstraction. The `getClos` function returns a set of triples from an abstract closure. For each program point pp of the abstract closure, it corresponds to a λ-abstraction of the main program: $\mathbf{prog}@\,\mathrm{pp} = Lam(x, t)$. The program point of $t$ is $\mathrm{pp}\cdot 1$ and the associated abstract environment is, by definition, $\mathcal{A}(\mathrm{pp})$.

**Lemma 11** *For all unspecified values $x$ of λ-calculus, $[\![x]\!]^\sharp$ is a sound approximation of $[\![x]\!]^{ppoint}$.*

It remains to define the update functions. The $\mathbf{update}^{in}_{\texttt{eval}}$ function updates the AI-state such that the input l-environment is included in the abstract l-environment in the AI-state corresponding to the program point of the λ-term. Therefore, when a call `eval` $(\sigma^\sharp, \mathrm{pp})$ is performed, the update function ensures that the sub-term at program point pp will be called on increasingly bigger abstract l-environment. Therefore, increasingly bigger abstractions will be computed for the evaluation of this sub-term. The $\mathbf{update}^{out}$ does nothing.

$$\mathbf{update}^{in}_{\texttt{eval}}(\mathcal{A}, (\sigma^\sharp, \mathrm{pp}_t)) = \mathcal{A}[\mathrm{pp}_t \mapsto \sigma'^\sharp], (\sigma'^\sharp, \mathrm{pp}_t) \qquad \text{with } \sigma'^\sharp = \mathcal{A}(\mathrm{pp}_t) \sqcup^\sharp \sigma^\sharp$$
$$\mathbf{update}^{out}_{\texttt{eval}}(\mathcal{A}, (\sigma^\sharp, \mathrm{pp}_t), c^\sharp) = \mathcal{A}, c^\sharp$$

The abstract interpretation is guaranteed to terminate. There is a finite number of abstract l-closures because there is a finite number of program points. There is a finite number of variables for a given λ-term, thus there is a finite number of abstract l-environment and of AI-states. The analysis will converge in a finite number of steps.

**Lemma 12** *The update functions are monotonic in both arguments, in the sense of definition 27.*

We show how to compute a CFA and prove that our analysis is correct. Let $\sigma_0 \in V^{\texttt{ppoint}}(\texttt{env})$ and $\sigma_0^\sharp \in V^\sharp(\texttt{env})$ be the concrete and abstract l-environments with empty domains. Let $E_0 = \{s \mapsto \sigma_0, t \mapsto \epsilon\}$ and $E_0^\sharp = \left\{s \mapsto \sigma_0^\sharp, t \mapsto \epsilon\right\}$ be a concrete and an abstract Skel environments. Let $\mathcal{A}_0$ be the empty mapping from program points to abstract l-environments.

**Lemma 13**

$$\sigma_0 \in \gamma_{env}(\mathcal{A}_0, \sigma_0^\sharp)$$

**Proof 7** *By definition of $\gamma_{env}$ and because $\sigma_0$ and $\sigma_0^\sharp$ have an empty domain.*

**Lemma 14** *Let $\Gamma = \{s \mapsto \texttt{env}, t \mapsto \texttt{lterm}\}$.*

$$E_0 \in \gamma_\Gamma(\mathcal{A}, E_0^\sharp)$$

**Proof 8** *$E_0$ and $E_0^\sharp$ have the same domain, $E_0(t) \in \gamma(\mathcal{A}_0, E_0^\sharp(t))$, and $E_0(s) \in \gamma(\mathcal{A}_0, E_0^\sharp(s))$ by Lemma 13. Therefore, the Lemma is true by definition of $\gamma_\Gamma$.*

$\epsilon$ is the program point of the root of the analysed program, **prog**. The abstract interpreter computes an abstract closure that is a correct approximation of the concrete closure returned by the big-step semantics.

**Theorem 5**

$$\begin{array}{c} E_0, \texttt{eval}\ s\ t \Downarrow_S (x, \mathrm{pp}, \sigma) \\ \varepsilon, \mathcal{A}_0, E_0^\sharp, \texttt{eval}\ s\ t \Downarrow_S^\sharp c^\sharp, \mathcal{A} \end{array} \implies (x, \mathrm{pp}, \sigma) \in \gamma(\mathcal{A}, c^\sharp)$$

**Proof 9** *The proof uses Theorem 3. The concretisation functions are monotonic (Lemma 10). The abstract instantiations of the unspecified terms are sound approximations of the concrete instantiations of the unspecified terms (Lemma 11). The update functions are monotonic (Lemma 12). Furthermore, $E_0 \in \gamma_\Gamma(\mathcal{A}_0, E_0^\sharp)$ (Lemma 14). Therefore, Theorem 3 applies, and by instantiating it with $S = \texttt{eval}\ s\ t$, we obtain the desired result.*

## 5.4 Example

We set $\mathbf{prog} \equiv (\lambda f.(f\ \ \mathrm{id}_x)^{\mathrm{pp}_x}\ \ (f\ \ \mathrm{id}_y)^{\mathrm{pp}_y})\ \ (\lambda g.g^{\mathrm{pp}_g})$, the example of Section 5.1. Let $E_0^\sharp = \left\{s \mapsto \sigma_0^\sharp, t \mapsto \epsilon\right\}$ where $\sigma_0^\sharp$ is the abstract environment with an empty domain. Let $\mathcal{A}_0$ be the empty AI-state. Then, one can show that: $\mathcal{A}_0, E_0^\sharp, eval\ (s, \mathbf{prog}) \Downarrow_S^\sharp \left\{\mathrm{pp}_x, \mathrm{pp}_y\right\}, \mathcal{A}$. The result of the computation is an abstract closure $\left\{\mathrm{pp}_x, \mathrm{pp}_y\right\}$. As said previously, a program point pp can be interpreted as an abstraction of the closure $(x, t, \sigma)$, such that $\mathbf{prog}@\,\mathrm{pp} = Lam(x, t)$ and $\sigma \in \gamma(\mathcal{A}, \mathcal{A}(\mathrm{pp}))$.

Let $E_0 = \{s \mapsto \sigma_0, t \mapsto \epsilon\}$ where $\sigma_0$ is the concrete environment with an empty domain. Then one can show that $E_0, \texttt{eval}\ s\ t \Downarrow_S (y, y, \{\})$ and by applying Theorem 5, it is true that:

$$(y, y, \{\}) \in \gamma(\mathcal{A}, \left\{\mathrm{pp}_x, \mathrm{pp}_y\right\})$$

Because the analysis is finite, one can also analyse non-terminating program. Take $\mathbf{prog} \equiv (\lambda x.x\ x)\ (\lambda x.x\ x)$, a famous looping $\lambda$-term then on can derive the following statement:

$$\mathcal{A}_0, E_0^\sharp, eval\ (s, t) \Downarrow_S^\sharp \{\}$$

We successfully define a Control Flow Analysis for $\lambda$-calculus using our methodology of abstract interpretation using skeletal semantics. What is interesting about this analysis is that the control flow of the program is computed on the fly to perform the analysis.

Chapter 6

# An Abstract Interpreter Generator

Taiga (The Abstract Interpreter GenerAtor) is a 3K line of code, OCaml program that generates abstract interpreters from skeletal semantics. Given a skeletal semantics $\mathbb{S}$, an abstract interpreter is generated by providing definitions of the abstract domains for the unspecified types, and instantiations for the unspecified terms. Taiga uses the Necro library [37], a set of tools to parse, typecheck skeletal semantics, and more. The full code of Taiga is available at [41].

Taiga uses OCaml *modules*: a module is a collection of definitions of type definitions and expression definitions. A *module type*, or *signature*, is equivalent to a type but for modules: it is a collection of type definitions and typed names of expressions. An example is given on Figure 6.1. On the left column, a signature named `EXAMPLE` is defined. A type `t`, without definition, is declared, along with a value named `is_int` of type `t -> bool`. In the right column, the module `Example` with signature `EXAMPLE` must therefore provide a `t` type and a function `is_int` of the same type, otherwise OCaml will raise a type-checking error. A value `v` is also defined in the `Example` module. Outside the module, types and values of a module can be accessed by prefixing the name of the module then the name of the value or type, like `Example.v`. In this Chapter, we set the convention where module names start with a capital letter, like `Example`, and signature names are capitalised, like `EXAMPLE`.

Taiga makes extensive use of *functors*. A functor is an operator that given a module, generates a new module: it can be interpreted as a function that maps a module to another module. On Figure 6.2, a module `Example` is defined on the left-hand side, and a simple functor `F` is defined on the right-hand side. The functor `F` takes a module with signature `EXAMPLE` and returns a new module. The functor can be applied to a module with a correct signature and returns a new module that contains the types and values defined by the functor.

```
module EXAMPLE = sig
  type t

  val is_int: t -> bool
end
```

```
module Example : EXAMPLE = struct
type t =
  | Int of int
  | Bool of bool

let is_int (v: t): bool =
  match v with
  | Int _ -> true
  | _ -> false

let v = Int 3
end
let _ = assert_true (Example.is_int Example.v)
```

Figure 6.1: Example of an OCaml module with its signature

```
module Example = struct
type t =
  | Int of int
  | Bool of bool

let is_int (v: t): bool =
  match v with
  | Int _ -> true
  | _ -> false
end
```

```
module F(Ex: EXAMPLE) = struct
type s =
  | Some of Ex.t
  | None

let is_int (v: s): bool =
  match v with
  | Some v' -> Ex.is_int v'
  | None -> false
end

module NewModule = F(Example)
```

Figure 6.2: Example of an OCaml Functor

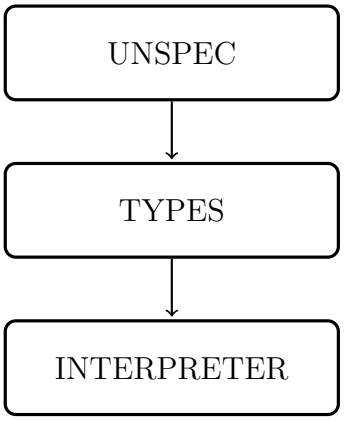



Figure 6.3: Schematics of the Infrastructure of Taiga

## 6.1 Overview of the Infrastructure

The infrastructure of Taiga is presented of Figure 6.3. An OCaml module with signature `UNSPEC` must be written by the user. A module with signature `UNSPEC` contains the definitions of the abstract domains for unspecified types. Indeed, it contains the instantiations of the unspecified types, along with their abstract comparisons and unions. The module also contains the skeletal semantics and the definition of the state of the abstract interpretation. Then, the `BuildTypes:UNSPEC -> TYPES` functor generates a module `TYPES` that contains what was in the `UNSPEC` module, the type of abstract values, comparisons and abstract unions for abstract values, mappings containing the definitions of the unspecified terms, and the update functions. Then, the `MakeInterpreter: TYPES -> INTERPRETER` functor generates an `INTERPRETER` module that contains the abstract interpreter of the language.

To produce an abstract interpreter, the user writes an `UNSPEC` module. Then, the `BuildTypes` functor generates a `TYPES` module. The `TYPES` module is incomplete at this point because the unspecified terms and update functions are undefined. Once they are added to the module by the user, the completed `TYPES` module is passed to the `MakeInterpreter` functor that generates the abstract interpreter.

## 6.2 State Monad

The abstract interpreter generated by Taiga uses the state monad to facilitate the manipulation of the AI-state. We give a quick overview of the state monad. A monad is

a type constructor, a generator and a binding operator. Let `m` be a type constructor, `return: 'a -> 'a m` a generator and `bind: 'a m -> ('a -> 'b m) -> 'b m` a binding operator. The generator `return` wraps a value into the monad. Then, computations can be sequenced with the binding operator. In OCaml, it is possible to define the notation `let* v = expr1 in expr2` for `binding expr1 (fun v -> expr2)` to get a more natural syntax.

The state monad is used to write stateful computations in a functional way. In OCaml, it can be defined as:

```
type ('s, 'a) m = ('s -> ('s * 'a))
let return v = (fun s -> (s, v))
let bind mv f = (fun s -> let (s', v') = mv s in f v' s')

let get () = (fun s -> (s, s))
let put s' = (fun _ -> (s', ()))

let ( let* ) = bind
```

A monadic value with type `(state, t) m` is a function which given a state, returns a new state and a value. The value `return v` is the computation that does not modify the state and returns the value `v`. The binding operator is used to sequence stateful computations. We have added the `get` and `put` functions. They are used to retrieve the state and to update the state of the monad respectively. Finally, in OCaml, we can define let-operator to write more readable code with monads. Here `let*` is defined as the binding operator and can be used to pipe computations involving the state monad.

```
let f (x: int): ('s, int) m = ...
let g (x: int): ('s, int) m = ...

let* n = f 0 in
g n
```

Here, the call `f 0` returns a stateful computation. The `let*` operator is used to pipe the result encapsulated in the state monad to the input of `g`. It is equivalent to writing:

```
bind (f 0) (fun n -> g n)
```

## 6.3 The Unspecified Module

We show how to derive an abstract interpreter for the While language using Taiga. To get a working abstract interpreter, one must define language-specific abstractions. This is the role of the **UNSPEC** module. We start by defining a module called **Unspec**, where we define the state of the abstract interpretation.

```
module Unspec = struct

  type 'abst ai_state = { stores: ((string * 'abst) list) PPMap.t; prg:
  ↪  'abst }
  let init_ai_state (prg: 'abst) =
    { stores = PPMap.add (Ppoint.epsilon) [] PPMap.empty; prg = prg }
...
```

The `ai_state` type is parameterised by the `'abst` type variable that represents types of abstract values. Because the type of abstract values is not defined at this point, we use a type variable. The `'a PPMap.t` type is a mapping from program points to values of type `'a`. The `ai_state` is a record containing a field `stores` which is a mapping from program points to abstract stores represented as an association list. This definition of the AI-state is similar to the definition given in Chapter 4.

A program point is defined in a Taiga library as:

```
type flowpp =
  | In
  | Out
type ppoint =
  | SubTerm of flowpp * int list
```

The `ppoint` type is a flow direction (In or Out) and a list of integers. The AI-state we have defined is similar to the one defined on paper in Section 4.3. The list of integers maps to a sub-term of the analysed program. The flow direction differentiates the store before evaluating the sub-term, and the store obtained after the evaluation of the sub-term at the program point.

The unspecified types are instantiated by defining a `base_value` type which is an Algebraic Data Type. Each constructor of the ADT corresponds to an unspecified type. To do an interval analysis for the While language, the definition can be:

```
type 'abst base_value =
  | Ident of string
  | Lit of int
  | Store of (string * Interv.t) list
  | Int of Interv.t
```

An `Indent` is defined as a string. A `Lit` is defined as an integer. A `Store` is defined as an association list. An `Int` is defined as an interval (we use an external library). The `base_value` type is parameterised by a type variable `'abst` which represents the type of abstract values that is not defined yet. In our example, it is unused but sometimes we want the base values to be defined mutually recursively with the whole set of abstract values.

Printing functions, abstract comparisons and unions for unspecified types must be provided. We only show the comparison functions, because the printing functions and the abstract unions follow the same code structure.

```
(* Comparison functions*)
let rec lesser_base_value (lesser: 'abst -> 'abst -> bool) (bv1: 'abst
↪  base_value) (bv2: 'abst base_value): bool =
  match (bv1, bv2) with
  | (Int i1, Int i2) -> Interv.subset i1 i2
  | (Store s1, Store s2) ->
    lesser_store lesser s1 s2
  | (Lit l1, Lit l2) -> l1 = l2
  | _ -> bv1 = bv2
```

The `lesser_base_value` function is parameterised by a function `lesser` to compare abstract values. The parameters `bv1` and `bv2` are the values to compare. The `lesser_base_value` will only be called on values of the same Skel type, thus one can pattern match `bv1` and `bv2` and suppose they are built using the same constructor of the `base_value` type. Comparison of two values of the `int` Skel type is set inclusion. Comparison of two values of the `store` Skel type is done using an auxiliary function `lesser_store` which does a point-wise extension of the comparison. The comparison of the literals is the comparison for a flat lattice. The top and bottom values are not defined for the unspecified values at this point, as it was done when defining the abstract values in Section 4.2.

## 6.4 The Types Module

From the UNSPEC module, the BuildTypes functor produces a TYPES module. A TYPES signature contains the UNSPEC signature, with the definitions of all abstract values, and abstract environments. Moreover, it contains printing functions, comparisons, and unions for abstract values. Finally, the BuildTypes functor generates default update functions, and two empty mappings with the unspecified terms and the unspecified functions. The user must manually complete these mappings to provide implementations for unspecified terms and functions. The BuildTypes functor start by defining the abstract values:

```
module BuildTypes (Unspec: UNSPEC) = struct
  module SMap = Util.SMap
  include Unspec

  (* Type of program points *)
  type ppoint = Ppoint.t

  (* Base values *)
  type base_value = abst_value Unspec.base_value

  (* Abstract values *)
  and abst_value =
    | Tuple of abst_value tuple set
    | Fun of abst_function set
    | Const of constructor * abst_value
    | Base of base_value
    | PPoint of ppoint
    | Bot
    | Top

  (* The constructor for the different functions *)
  and abst_function =
    | SpecFun of int * abst_value list * string * abst_value
    | UnspecFun of int * abst_value list * string
    | Clos of pattern * env * skeleton
```

```
  (* The abstract environment, mapping skeletal variables to abstract
  ↪  values *)
  and env = (string * abst_value) list

  (* The state of the abstract interpretation *)
  type ai_state = abst_value Unspec.ai_state
```

The `SMap` module defines a mapping type where the keys are strings. A mapping of type `a SMap.t` is a mapping from string to values of type `a`. The `ppoint` type is the type presented in the previous section. The types `base_value`, `abst_value`, `abst_function`, and `env` are defined mutually recursively. The `base_value` type is the same as the one defined in the `Unspec` module, except the type variable has been replaced with the `abst_value` type. The `abst_value` type is the type of abstract values. An abstract value can be:

- An abstract tuple, a set of tuple of abstract values
- An abstract function, a set of closures
- An abstract algebraic value, a constructor applied to an abstract value
- A base value, a value of an unspecified type
- A program point
- The bottom or top value

The `abst_function` type defines the closures. A closure can be a named closure or an anonymous closure. The `env` type defines the abstract environments: partial function from variables of Skel to abstract values, here implemented as an association list. The `ai_state` type is the type of the AI-states from the `Unspec` module parameterised by the type of abstract values.

The `BuildTypes` functor defines default update functions:

```
(* Default update functions *)
let update_in (_: fun_name) (param: abst_value): (ai_state, abst_value
↪  list) SM.t =
  SM.return params
let update_out (_: fun_name) (_: abst_value) (res: abst_value) =
  SM.return res
```

The default update functions do not modify the ai-state, the parameter or the result. This is the first time the state monad appears. Rather than explicitly piping the AI-state during the computations, the state monad is used to get more readable code. A value of type `(ai_state, abst_value) SM.t` is a function taking an AI-state and returning a new AI-state and an abstract value. We recall that the expression `SM.return param` is a function that, given an AI-state `ais`, returns a pair `(ais, param)`: the AI-state and the parameter are not modified.

Finally, the `BuildTypes` functor defines empty mappings containing the specifications of the unspecified terms

```
(* The mapping containing the unspecified functions *)
let unspec_funs: (abst_value list -> (ai_state, abst_value) SM.t) SMap.t
↪  = SMap.empty

(* The mapping containing the unspecified values *)
let unspec_vals: abst_value SMap.t = SMap.empty
```

The `unspec_funs` value maps function names to their specifications: functions that given a parameter return new abstract value encapsulated in the state monad because unspecified functions may modify the AI-state. The `unspec_vals` value maps unspecified term names to their specifications: abstract values.

From the `Unspec` module that we have defined for or While language, we generate a `TYPES` module using the functor.

```
module WhileTypes = BuildTypes(Unspec)
```

Let us see how to instantiate the unspecified terms, in particular unspecified functions for our While language.

```
let read (v: abst_value): (ai_state, abst_value) SM.t =
  match v with
  | Tuple [[ Base (Ident x) ; Base (Store st) ]] ->
    SM.return @@ List.assoc x st
  | _ -> failwith "read: parameter has wrong type"

let unspec_funs = SMap.add "read" read unspec_funs
```

Because values of skeletal semantics are deep embedded, a function takes an abstract value, the parameter, and returns an abstract value encapsulated in the state monad, the result. The `read` function expects a tuple of an identifier, which is a base type, and a store which is also a base type. The value bound to the identifier is fetched in the store and returned, encapsulated in the state monad.

Finally, the update functions must be instantiated. We present the `update_in` function of the While language.

```
(* The updates functions *)
let update_in (fun_name: string) (v: abst_value): (ai_state, abst_value
↪  list) SM.t =
  if fun_name = "eval_stmt" then
    match v with
    | Tuple [[ Base (Store (st)) ; PPoint(ppt) ]] ->
       let* (ais: ai_state) = SM.get () in

       let st_ppt = PPMap.find_default ppt [] ais.stores in
       let* new_st = join_store (join_f widen_base_value) st_ppt st in

       let new_ais = {ais with stores = (PPMap.add ppt new_st
       ↪  ais.stores); } in

       let* () = SM.put new_ais in

       SM.return @@ [[ Base (Store (new_st)) ; PPoint (ppt) ]]
    | _ -> failwith "update_in: invalid values"
  else
    SM.return vl
```

It takes the name of the specified function and its parameter. If the function called is `eval_stmt`, then the list of arguments is pattern matched and is expected to be a store and a program point mapping to the sub-term of the program being evaluated. First, the AI-state is retrieved using the `SM.get` function of the state monad. The store associated with the program point is fetched in the global mapping of the AI-state. Then, a new store is obtained by joining the abstract store given as argument of the `eval_stmt` and

the store of the global mapping. This new store is put in the global mapping and the AI-state is updated using the `put` function of the state monad. Finally, the new parameter of the `eval_stmt` function is returned with the new store and is encapsulated in the state monad. The `update_out` function is similar but modifies the output store of the function rather than the store passed as parameter. This concludes the definition of the `TYPES` module, and we can now derive an abstract interpreter in the next section.

Because of the deep embedding, writing the definitions of the unspecified values is cumbersome and can be error-prone because of the loss of typechecking information. All abstract values have type `abst_value` and therefore typing information of Skel typechecker has been lost. One axis of improvement of the abstract interpreter generator would be to generate a `BuildTypes` module from a skeletal semantics where each unspecified type would be defined independently, rather than having one `base_value` type with one constructor for each unspecified type. Moreover, typing information could be kept using Generalised Algebraic Data-types, like Necrodebug [35] does. Necrodebug is a step-by-step debugger for Skel written in OCaml. Similarly to Taiga, Necrodebug uses deep-embedding to represent Skel values, but thanks to GADTs the typing information is not lost.

## 6.5 The Abstract Interpreter Generator

Now that the unspecified types and terms have been instantiated, it is time to analyse programs. The user has nothing more to do other than provide the programs to analyse. Taiga provides a functor that takes a `TYPES` module, like the one we have just defined for the While language and produces a new module containing the abstract interpreter. The abstract interpreter module contains many functions for pattern matching, unfolding, and more. Here we present the functions of abstract interpretation of terms and skeletons. We start with the function of evaluation of terms. The function takes an abstract environment, a term and returns an abstract value.

```
(** Computes the abstract interpretation of a term
    @param e the abstract environment
    @param t the skeletal term to compute
    @return the result of the abstract interpretation
 *)
let rec abstract_interpretation_term (e: env) (t: term): abst_value =
  match t with
```

```
  | TVar(TVLet(x, _)) -> assoc_env x e
  | TVar(TVTerm(Unspec, _, tname, _, _)) -> SMap.find tname unspec_vals
  | TVar(TVTerm(Spec, _, tname, _, nt)) ->
    ...
  | TConstr(c, _, t) -> Const(c, abstract_interpretation_term e t)
  | TTuple(tl) -> Tuple(Set.singleton (List.map
  ↪  (abstract_interpretation_term e) tl))
```

The term is pattern-matched, the first three cases correspond to when the term is a variable. A variable is not evaluated in the same manner depending on how it was defined. The first case is when the variable is defined by a let-binding. Then its content, stored in the environment, is fetched with the `assoc_env` function. The second case is when the variable is an unspecified term defined in the skeletal semantics. The value returned is stored in the mapping of unspecified values `unspec_vals`, defined in the `TYPES` module. The third case is when the variable is a specified term defined in the skeletal semantics, the code is not shown here. The next case is when the term is a constructor applied to a term. The evaluation is the constructor applied to the evaluation of the term. The last case is the evaluation of a tuple term. A tuple term is a list of terms, representing a tuple of terms. Each term is evaluated, and put into a singleton set, giving an abstract tuple.

Then comes the evaluation of skeletons. We only show the evaluation of branching. The evaluation of skeletons takes as parameters: a callstack, an abstract environment, and a skeleton. A branching skeleton is a list of skeletons, each one being one branch. To evaluate a branching, each branch is evaluated, which gives one abstract value per branch. All abstract values are then merged using the abstract union.

```
(** Computes the abstract interpretation of a term
    @param e the abstract environment
    @param t the skeletal term to compute
    @return the result of the abstract interpretation
 *)
let rec abstract_interpretation_term (e: env) (t: term): abst_value =
  match t with
  | TVar(TVLet(x, _)) -> assoc_env x e
  | TVar(TVTerm(Unspec, _, tname, _, _)) -> SMap.find tname unspec_vals
  | TVar(TVTerm(Spec, _, tname, _, nt)) ->
    ...
```

```
  | TConstr(c, _, t) -> Const(c, abstract_interpretation_term e t)
  | TTuple(tl) -> Tuple(Set.singleton (List.map
  ↪  (abstract_interpretation_term e) tl))
```

To analyse a program, it must be written in the skeletal semantics. The terms `zero`, `one`, and `two` are unspecified terms of type *lit* and are instantiated, as seen earlier, as integers 0, 1, and 2 respectively.

```
(* x := 0; if (x = 0) { y := 1 } else { y := 2 } *)
val main2: stmt =
  (Seq(Assign(x, Const zero),
       If(Equal(Var x, Const zero),
          Assign(y, Const one),
          Assign(y, Const two))))

(* x := rand(0, 3); if (x = 0) { y := 1 } else { y := 2 } *)
val main3: stmt =
  (Seq(Assign(x, Rand(zero, three)),
       If(Equal(Var x, Const zero),
          Assign(y, Const one),
          Assign(y, Const two))))
```

The analysis returns the abstract value returned by the program and the AI-state, which gives the abstract stores in input and output for each program points.

```
main1: [x: Int([0., 0.]), y: Int([1., 1.])]
11_out: [x: Int([0., 0.]), y: Int([1., 1.])]
1_out: [y: Int([1., 1.]), x: Int([0., 0.])]
0_out: [x: Int([0., 0.])]
epsi_out: [x: Int([0., 0.]), y: Int([1., 1.])]
11_in: [x: Int([0., 0.])]
1_in: [x: Int([0., 0.])]
0_in: []
epsi_in: []


main2: [y: Int([1., 2.]), x: Int([0., 3.])]
```

```
12_out: [x: Int([0., 3.]), y: Int([2., 2.])]
11_out: [x: Int([0., 3.]), y: Int([1., 1.])]
1_out: [x: Int([0., 3.]), y: Int([1., 2.])]
0_out: [x: Int([0., 3.])]
epsi_out: [y: Int([1., 2.]), x: Int([0., 3.])]
12_in: [x: Int([0., 3.])]
11_in: [x: Int([0., 3.])]
1_in: [x: Int([0., 3.])]
0_in: []
epsi_in: []
```

The store returned by `main1` is the abstract store $\{x \mapsto [0;0], y \mapsto [1;1]\}$, which is expected to be a correct approximation of the result of the `main1` program.

Because the `update_in` function uses the widening operator, the abstract interpreter can analyse terminating and non-terminating loops. Provided that the instantiations of unspecified terms are correct, the result of the analysis is correct by Theorem 3.

We presented a method to derive correct and working abstract interpreters from a skeletal semantics and we successfully analysed programs. However, because of the deep embedding, writing the definitions of the unspecified values is cumbersome and can be error-prone because of the loss of typechecking information. All abstract values have type `abst_value` and therefore typing information of Skel type-checker has been lost. One axis of improvement of the abstract interpreter generator would be to generate a `BuildTypes` module from a skeletal semantics where each unspecified type would be defined independently, rather than having one `base_value` type with one constructor for each unspecified type. Moreover, typing information could be kept using Generalised Algebraic Data-types, like Necrodebug [35] does. Necrodebug is a step-by-step debugger for Skel written in OCaml. Similar to Taiga, Necrodebug uses deep-embedding to represent Skel values, but thanks to GADTs the typing information is not lost.

Chapter 7

# Conclusion and Future Work

## 7.1 Conclusion

Proving property about programs is important to guarantee that they are safe. However, designing correct analysers for languages is difficult, hence we design a methodology to get working analysers from formal descriptions of languages. Our goal is to build a usable framework to design proven correct analyses, as Verasco [17] does, but without trying to match their level of granularity.

The semantic description framework we chose is Skeletal Semantics [4]: it is expressive enough to describe complex languages like JavaScript [22], and simple enough to easily build tools, like our abstract interpreter generator. Moreover, there already has been work to define big-step semantics [39, 4] of languages from their skeletal semantics, that we have re-used. Indeed, our vision is that defining a semantics of a language and an analyser from the same object should make the proof of correctness of the analyser easy to show.

Our first contribution is to add program point support to the big-step semantics of Skel. The semantics is parameterised by the program of the object language. Sub-terms of the program are then referenced by program points. Our method works not only for the big-step interpretation but also for the abstract interpretation. It was pivotal to add support for program points in the interpretations of Skel as they are ubiquitous in the analysis of programs.

Then, we define the abstract interpretation of Skel. It covers all possible executions of the big-step semantics, and we think it terminates if a small lemma about the update functions holds. A theorem of correctness is also presented and proved, which says that the abstract interpretation of Skel computes a correct over-approximation of the big-step semantics of Skel, provided small lemmas on unspecified terms hold. We show how to compute an interval analysis on While programs with some examples.

We also present how to compute a control flow analysis for $\lambda$-calulus to demonstrate the versatility of the approach. We start with the definition of the skeletal semantics of

$\lambda$-calulus. The unspecified types and terms are instantiated, the abstract comparisons and unions are defined, along with the update functions. We get an abstract interpretation that computes a CFA for $\lambda$-calculus. By using the theorem of correctness of the abstract interpretation, we are able to prove that the derived abstract interpretation is indeed computing a correct CFA.

Finally, we present Taiga, an abstract interpreter generator. Given a skeletal semantics, and an abstract instantiation of unspecified types and terms, abstract unions and comparisons, Taiga generates an abstract interpreter capable of analysing languages. We test it on the While language and $\lambda$-calculus, we are able to get working abstract interpreters.

Did we succeed in providing a recipe to generate correct analyses from formal descriptions of languages? We propose a method to define abstract interpretation of languages from their skeletal semantics. We give examples on how to do an interval analysis on the While language, and a control flow analysis on $\lambda$-calculus. An abstract interpretation is obtained only by defining the abstract domains for the unspecified types, making easy the definition of analyses, supposing there is a skeletal semantics for the language. Our vision is that a language should be formally defined to get correct concrete and abstract interpreters and more. Thus, demanding a skeletal semantics for the language is a strong requirement but it can be amortised: the Necro tools [36, 27, 35] make a skeletal semantics interesting to generate a concrete interpreter, Coq code, LaTeXcode etc... The analyses can be guaranteed correct by proving small lemmas only on the language-specific parts, thus minimising the effort. Taiga, an abstract interpreter generator is implemented and allows the definition of abstract interpreters from their skeletal semantics by providing an implementation of the abstract domains. Taiga has been tested on While and the $\lambda$-calculus and successfully computes correct analyses.

However, some points can be greatly improved. The proof of termination of the abstract interpretation must be written, it is an essential property of an abstract interpretation. Then, the language-independent approach is a lot less precise than a language-specific method. For instance, the conditions of if-statements and loops are not used to refine the abstract states for the While language. The precision can also be improved by adding support for relational abstract domains, which are abstractions that establish relations between variables of a program, for instance, inequalities. Also, Taiga, the abstract interpreter generator proves difficult to use, and therefore not usable on large languages. Finally, to guarantee the correctness of the abstract interpretation, we should use a theorem prover, like Coq. Then, we could generate a proof of correctness of the abstract

interpretation from a skeletal semantics, provided a proof of the language-dependent lemmas.

## 7.2 Future Work

### 7.2.1 Proof of Termination of the Abstract Interpretation

One key issue with our abstract interpretation is that we did not prove it terminates for every skeleton. Indeed, given a skeleton $S$, and an abstract environment $E^\sharp$, the skeleton $S$ should be interpreted in finite time. Thus, for any AI-state $\mathcal{A}$, there should exist a $v^\sharp \in V^\sharp(\tau)$, an AI-state $\mathcal{A}'$, and a finite derivation tree with conclusion $\varepsilon, \mathcal{A}, E^\sharp, S \Downarrow_S^\sharp v^\sharp, \mathcal{A}'$. Hence, we would be able to guarantee the termination of our abstract interpreter, and that our abstract interpreter can analyse any program. To prove the termination, what can be done is un-intuitively define the abstract interpretation of Skel by co-induction. This is similar to what Schmidt does in [44]: we obtain potentially infinite abstract derivation trees. Therefore, the abstract interpretation of any skeleton would be defined. We think we can show that if the specifications of the update functions respect a convergence criteria, which will be presented later, then the abstract derivation trees are indeed finite.

We think the convergence of the analysis can be ensured by two ingredients. The Spec-Loop rule of the abstract interpretation (Figure 4.12) detects identical nested calls to a specified function and stops the computation when a fixpoint has been detected. Second, the $\mathbf{update}^{in}$ should ensure that a recursive function cannot call itself indefinitely. Thus, it should force identical nested calls for non-terminating recursive functions. We think the only condition for an abstract evaluation not to terminate is that there is an infinite chain of nested calls to a specified function $f : \tau_1 \to \tau_2$. An infinite chain of nested call is when there is a sequence $(v_n^\sharp)_{n\in\mathbb{N}} \in V^\sharp(\tau_1)$ such that for every $n \in \mathbb{N}$ the evaluation of $f\ v_n^\sharp$ induces a call $f\ v_{n+1}^\sharp$, and all $v_n^\sharp$ are different. We think that if the derivation tree of $\pi, \mathcal{A}, E^\sharp, S \Downarrow_S^\sharp v^\sharp, \mathcal{A}$ is infinite, then there is an infinite chain of nested calls for at least one specified function. Hence, by contraposition, if there is a finite number of calls to every specified function, then the derivation tree of $\pi, \mathcal{A}, E^\sharp, S \Downarrow_S^\sharp v^\sharp, \mathcal{A}$ is finite.

If we can prevent non-converging nested calls, then we should be able to prove that the abstract derivation tree is finite. To do that, we think it is enough that the $\mathbf{update}^{in}$ function respects a simple convergence criteria. Let $\mathbb{S}$ be a skeletal semantics. We say the $\mathbf{update}^{in}$ function has the widening property with an increasing sequence of $(\mathcal{A}_i, v_i^\sharp)$, the

sequence $x_{i+1} = \mathbf{update}_f^{in}(x_i)$ with $x_0 = (\mathcal{A}_i, v_i^\sharp)$ then the sequence $x_i$ converges in finite time.

If the $\mathbf{update}^{in}$ function has the widening property, then there should not be an infinite chain of nested calls to the same specified function. Suppose the derivation tree of $\pi, \mathcal{A}, E^\sharp, S \Downarrow_S^\sharp v^\sharp, \mathcal{A}$ is infinite. Then there is a least one infinite chain of nested calls for a specified function $f$. This infinite chain of nested calls is not caught by the Spec-Loop rule of the abstract interpretation, meaning that the chain does not converge. But it is impossible as $\mathbf{update}^{in}$ has the widening property by definition. By contradiction, the derivation tree must be finite. Essentially, proving that the $\mathbf{update}^{in}$ has the widening property is equivalent to showing in classical abstract interpretation that the widening operator converges in finite time. This way, we hope we can prove that our analysis terminates.

### 7.2.2 Adding Guards and Relational Analyses

To improve the precision of our analyses, we must add support for guards. Guards are tests, like $isZero\ x$, that can be used to refine the abstractions. Suppose the following Skel code is in the skeletal semantics of While, and we keep the abstractions of Chapter 4, where integers are abstracted as intervals.

```
let x = rand(zero, one) in
branch
  let () = isZero x in
  return x
or
  let () = isNotZero x in
  return zero
end
```

In the first branch, $x$ should be equal to $zero$, and so should be the returned value. However, because $x$ is returned, and the condition $isZero\ x$ is not used to refine the abstractions, the interval $[0; 1]$ is returned. In the second branch, the returned value is $zero$. Thus the analysis should be able to deduce that returning the value $zero$ is correct. However, because the conditions at the start of each branch are not used to improve the abstractions, our abstract interpretation returns an over-approximation.

One method to use guards is to define the *opposite* of the unspecified functions. Indeed, when the result of the computation is known, what can be said about the inputs? If we suppose that $() = isZero\ [0;1]$, then we can refine the input, it is interval $[0;0]$. To propagate this to the variable $x$, we can use indirections. Let us say that $x$ maps to a symbolic variable $\widetilde{x}$. Then, we can maintain a symbolic table that maps symbolic variables to values. This table can be modified by the opposite instantiations of the unspecified functions. For instance, in the first branch of the example, we start in a table where $\widetilde{x}$ is equal to interval $[0;1]$. The opposite of function $isZero$ written $\overline{isZero}$, refines the symbolic table given the parameter and the result of the function.

$$\left\{\ \widetilde{x} \mapsto [0;1]\ \right\} \Rightarrow \overline{isZero}(\widetilde{x}, ()) = \widetilde{x} \mapsto \widetilde{x} \cap [0;0] \Rightarrow \left\{\ \widetilde{x} \mapsto [0;0]\ \right\}$$

It also paves the way for relational abstract domains. A relational abstract domain computes relations between variables. For the While language, a relational abstract domain can be the abstraction of stores as a set of inequalities between variables. One must find a method to propagate the information that $lt(x, y) = 1$ to the abstract while store. An easy solution would be to change the skeletal semantics of While such that $lt$ takes an abstract store as parameter and returns a new abstract store. Then, one could write the opposite function of $lt$ as:

$$\left\{\begin{array}{c} s \mapsto \top \\ \widetilde{x} \mapsto \top \\ \widetilde{y} \mapsto \top \end{array}\right\} \Rightarrow \overline{lt}((s, \widetilde{x}, \widetilde{y}), 1) = s \mapsto s \cup \{\widetilde{x} \leq \widetilde{y}\} \Rightarrow \left\{\begin{array}{c} s \mapsto \{\widetilde{x} \leq \widetilde{y}\} \\ \widetilde{x} \mapsto \top \\ \widetilde{y} \mapsto \top \end{array}\right\}$$

However, it implies modifying the skeletal semantics of the object language, which ideally should not be necessary.

### 7.2.3 Real-world languages

The approach remains to be tested on real languages, but we see several issues before it can be realistically done. We present the biggest issues we think must be tackled before going forward.

First, as it was discussed in Chapter 6, the implementation is not easy to use yet. The deep embedding makes the instantiations of the unspecified types and terms not as easy as we would want. The instantiations of all unspecified types are put in a `base_value` type, each unspecified type has its dedicated constructor which is impractical for the user.

Also, Skel type information is lost with the deep embedding. Thus, when instantiating the unspecified terms, for instance, the user may give a definition that should not be accepted by the type checker of Skel, leading to errors. On small languages with a few unspecified types and terms, like the While language, this is not an issue as the unspecified types and terms instantiations are very simple. However, for larger languages with dozens of unspecified types and terms, errors of instantiation will probably be too difficult to debug. One solution would be to code a generator of skeletal semantics specific `UNSPEC` type module. For instance, given the skeletal semantics of While, Taiga would generate an `UNSPEC_WHILE` module that could look like this:

```
module type UNSPEC_WHILE = struct
  type int
  type lit
  type store

  val join_int: int -> int -> int
  val join_lit: lit -> lit -> lit
  val join_lit: store -> store -> store
  ...
end
```

Therefore, the user would define an `UNSPEC_WHILE` module rather than an `UNSPEC`. A module of type `UNSPEC_WHILE` is easier to write as there is one OCaml type to define for each unspecified type. Then, the functor `BuildTypes` would also be generated from the skeletal semantics. For the While skeletal semantics, the `BuildTypes` functor would look like:

```
module BuildTypesOfWhile (Unspec: UNSPEC_WHILE): TYPES = struct
  type base_value =
  | Int of Unspec.int
  | Lit of Unspec.lit
  | Store of Unspec.store

  ...
end
```

A similar approach can be done for the unspecified term: by generating OCaml code we think we can ease the instantiations of the unspecified terms.

Second, a skeletal semantics must be defined for the language to be analysed. There are projects to write skeletal semantics for real-world languages, like JSkel [22], or PySkel [2]. Both projects do not cover all features of the language they describe. JSkel supports a large part of JavaScript, except function calls. It is a 15K line of Skel project, therefore the scale of the project is much bigger than the skeletal semantics we experimented on. PySkel is a 1.5K line of Skel project. This partial formalisation of Python has nevertheless support for functions, classes, objects... Because of its size and large cover of Python functionalities, it seems a good candidate to experiment on.

Third, our approach does not support polymorphism. JSkel and PySkel both make extensive use of polymorphism.

Once these three issues are solved, the abstract interpretation of real-world language may be at reach. Then, using the current or future implementation of JSkel and PySkel, the approach could be tested on.

### 7.2.4 Formal Proof in Coq

To get a formally correct abstract interpreter, we should use an interactive theorem prover, like Coq. There already exists a formalisation of Skel big-step semantics in Coq [36]. Moreover, NecroCoq can generate a Coq representation of any skeletal semantics.

To derive a verified abstract interpreter in Coq, we would first need to define the abstract interpreter of Skel in Coq. It should most likely be close to the OCaml implementation, Taiga. However, in Coq, only well-defined functions that terminate can be defined, and the proof of termination of the abstract interpretation would probably be necessary. Then, the theorem of correctness between the big-step semantics of Skel and the abstract interpreter of Skel must be proved correct in Coq. The proof is mainly an induction on the derivation tree of the abstract interpretation. The infrastructures to let the user specify the unspecified types and terms, the abstract comparisons and unions, and the definition of the AI-state must then be added, similar to the Taiga ones. Finally, the lemmas of correctness of the unspecified terms could be generated. The proofs are left to the user. By composing these lemmas with the theorem of correctness we would obtain a proven correct abstract interpreter for the language that can be executed, and complete the objective of mechanically proven correct abstract interpreters.

Another way to address the issue of expressing computations that are non trivially ter-

minating is to use Interaction Trees. Interaction trees [48] are a recent method to describe possibly non-terminating computations in Coq. One major benefit is that a computation described by an Interaction Tree can be extracted to working OCaml code. Also, the approach has already been tested to relate two semantics: by proving the correctness of a toy compiler for a minimal imperative language. However, we do not know how non-determinism can be handled with interaction trees, and this issue must be addressed to make our approach work with Interaction Tree.

# Bibliography


[1] Alexander Aiken and Edward L Wimmers, "Solving systems of set constraints", *in*: *[1992] Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, IEEE, 1992, pp. 329–340.

[2] Martin Andrieux and Alan Schmitt, "Skeletal Semantics of a Fragment of Python", *in*: *JFLA 2024–35es Journées Francophones des Langages Applicatifs*, 2024.

[3] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli, "A survey of attacks on ethereum smart contracts (sok)", *in*: *Principles of Security and Trust: 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings 6*, Springer, 2017, pp. 164–186.

[4] Martin Bodin, Philippa Gardner, Thomas Jensen, and Alan Schmitt, "Skeletal semantics and their interpretations", *in*: *Proceedings of the ACM on Programming Languages* 3.*POPL* (2019), pp. 1–31.

[5] Denis Bogdanas and Grigore Roşu, "K-Java: A complete semantics of Java", *in*: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2015, pp. 445–456.

[6] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez, "Moving fast with software verification", *in*: *NASA Formal Methods Symposium*, Springer, 2015, pp. 3–11.

[7] The Coq Development Team, *The Coq Proof Assistant, version 8.8.0*, version 8.8.2, Apr. 2018, DOI: 10.5281/zenodo.1219885, URL: https://inria.hal.science/hal-01954564.

[8] Patrick Cousot, *The Calculational Design of a Generic Abstract Interpreter*, Marktoberdorf Course Notes, 1998.

[9] Patrick Cousot and Radhia Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints", *in*: *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1977, pp. 238–252.

[10] Patrick Cousot and Radhia Cousot, "Systematic design of program analysis frameworks", *in*: *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1979, pp. 269–282.

[11] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival, "The ASTRÉE analyzer", *in*: *European Symposium on Programming*, Springer, 2005, pp. 21–30.

[12] Vikram Dhillon, David Metcalf, Max Hooper, Vikram Dhillon, David Metcalf, and Max Hooper, "The DAO hacked", *in*: *blockchain enabled applications: Understand the blockchain Ecosystem and How to Make it work for you* (2017), pp. 67–78.

[13] Manuel Fähndrich, Jeffrey S Foster, Zhendong Su, and Alexander Aiken, "Partial online cycle elimination in inclusion constraint graphs", *in*: *ACM SIGPLAN Notices*, vol. 33, 5, ACM, 1998, pp. 85–96.

[14] John Hughes, "Generalising monads to arrows", *in*: *Science of computer programming* 37.*1-3* (2000), pp. 67–111.

[15] Thomas Jensen, Vincent Rébiscoul, and Alan Schmitt, "Building CFA for $\lambda$-calculus from Skeletal Semantics", *in*: *JFLA 2023-34èmes Journées Francophones des Langages Applicatifs*, 2023, pp. 121–140.

[16] Thomas Jensen, Vincent Rébiscoul, and Alan Schmitt, "Deriving Abstract Interpreters from Skeletal Semantics", *in*: *arXiv preprint arXiv:2309.07298* (2023).

[17] Jacques-Henri Jourdan, "Verasco: a formally verified C static analyzer", PhD thesis, Universite Paris Diderot-Paris VII, 2016.

[18] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie, "A formally-verified C static analyzer", *in*: *ACM SIGPLAN Notices* 50.*1* (2015), pp. 247–259.

[19] Gilles Kahn, "Natural semantics", *in*: *Annual symposium on theoretical aspects of computer science*, Springer, 1987, pp. 22–39.

[20] Sven Keidel, Casper Bach Poulsen, and Sebastian Erdweg, "Compositional soundness proofs of abstract interpreters", *in*: *Proceedings of the ACM on Programming Languages* 2.*ICFP* (2018), pp. 1–26.

[21] Adam Khayam, Victoire Noizet, and Alan Schmitt, "A Faithful Description of ECMAScript Algorithms", *in*: *Proceedings of the 24th International Symposium on Principles and Practice of Declarative Programming*, PPDP '22, Tbilisi, Georgia: Association for Computing Machinery, New York, NY, USA, 2022, ISBN: 9781450397032, DOI: 10.1145/3551357.3551381, URL: https://doi.org/10.1145/3551357.3551381.

[22] Adam Khayam, Victoire Noizet, and Alan Schmitt, "A Faithful Description of ECMAScript Algorithms", *in*: *Proceedings of the 24th International Symposium on Principles and Practice of Declarative Programming*, 2022, pp. 1–14.

[23] John Kodumal and Alex Aiken, "Banshee: A scalable constraint-based analysis toolkit", *in*: *International Static Analysis Symposium*, Springer, 2005, pp. 218–234.

[24] Junghee Lim and Thomas Reps, "TSL: A system for generating abstract interpreters and its application to machine-code analysis", *in*: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 35.*1* (2013), pp. 1–59.

[25] Jacques-Louis Lions, Lennart Luebeck, Jean-Luc Fauquembergue, Gilles Kahn, Wolfgang Kubbat, Stefan Levedag, Leonardo Mazzini, Didier Merle, and Colin O'Halloran, *Ariane 5 flight 501 failure report by the inquiry board*, 1996.

[26] Alan Schmitt Martin Andrieux, *PySkel, https://gitlab.inria.fr/skeletons/pyskel*, URL: https://gitlab.inria.fr/skeletons/pyskel.

[27] Enzo Crance Martin Bodin Nathanael Courant and Victoire Noizet, *Necro Ocaml Generator, https://gitlab.inria.fr/skeletons/necro-ml*, URL: https://gitlab.inria.fr/skeletons/necro-ml.

[28] Jan Midtgaard, "Control-flow analysis of functional programs", *in*: *ACM computing surveys (CSUR)* 44.*3* (2012), p. 10.

[29] Jan Midtgaard and Thomas Jensen, "A calculational approach to control-flow analysis by abstract interpretation", *in*: *International Static Analysis Symposium*, Springer, 2008, pp. 347–362.

[30] Jan Midtgaard and Thomas P Jensen, "Control-flow analysis of function calls and returns by abstract interpretation", *in*: *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, 2009, pp. 287–298.

[31] Dominic P Mulligan, Scott Owens, Kathryn E Gray, Tom Ridge, and Peter Sewell, "Lem: reusable engineering of real-world semantics", *in*: *ACM SIGPLAN Notices* 49.*9* (2014), pp. 175–188.

[32] Flemming Nielson, "A denotational framework for data flow analysis", *in*: *Acta Informatica* 18.*3* (1982), pp. 265–287.

[33] Flemming Nielson, "Two-level semantics and abstract interpretation", *in*: *Theoretical Computer Science* 69.*2* (1989), pp. 117–242.

[34] Flemming Nielson, Hanne R Nielson, and Chris Hankin, *Principles of program analysis*, Springer, 2015.

[35] Victoire Noizet, *Necro Debugger*, URL: `https://gitlab.inria.fr/skeletons/necro-debug`.

[36] Victoire Noizet, *Necro Gallina Generator, https://gitlab.inria.fr/skeletons/necro-coq*, URL: `https://gitlab.inria.fr/skeletons/necro-coq`.

[37] Victoire Noizet, *Necro Library, https://gitlab.inria.fr/skeletons/necro*, URL: `https://gitlab.inria.fr/skeletons/necro`.

[38] Victoire Noizet and Alan Schmitt, "Semantics in Skel and Necro", *in*: *ICTCS 2022 - Italian Conference on Theoretical Computer Science*, CEUR Workshop Proceedings, Rome, Italy, Sept. 2022.

[39] Victoire Noizet and Alan Schmitt, "Semantics in Skel and Necro", *in*: *ICTS 2022- Italian Conference on Theoretical Computer Science*, 2022.

[40] Gordon D Plotkin, *A structural approach to operational semantics*, Aarhus university, 1981.

[41] Vincent Rébiscoul, *Abstract Interpreter Generator, https://gitlab.inria.fr/skeletons/abstract-interpreter-generator*, URL: `https://gitlab.inria.fr/skeletons/abstract-interpreter-generator`.

[42] Grigore Roșu and Traian Florin Șerbănuță, "An overview of the K semantic framework", *in*: *The Journal of Logic and Algebraic Programming* 79.*6* (2010), pp. 397–434.

[43] Amr Sabry and Matthias Felleisen, "Reasoning about Programs in Continuation-Passing Style", *in*: *LISP AND SYMBOLIC COMPUTATION*, 1993, pp. 288–298.

[44] David A Schmidt, "Natural-semantics-based abstract interpretation (preliminary version)", *in*: *International Static Analysis Symposium*, Springer, 1995, pp. 1–18.

[45] Olin Shivers, "Control flow analysis in scheme", *in*: *ACM SIGPLAN Notices*, vol. 23, ACM, 1988, pp. 164–174.

[46] David Van Horn and Matthew Might, "Abstracting abstract machines", *in*: *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, 2010, pp. 51–62.

[47] Makarius Wenzel, Lawrence C Paulson, and Tobias Nipkow, "The isabelle framework", *in*: *Theorem Proving in Higher Order Logics: 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings 21*, Springer, 2008, pp. 33–38.

[48] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C Pierce, and Steve Zdancewic, "Interaction trees", *in*: *Proceedings of the ACM on Programming Languages* 4 (2020).

Appendix A

# Proof of Correctness of the Abstract Interpretation

Let $\mathbb{S}$ be a skeletal semantics, ProgTypes the set of program types, and **prog** a value of a program type.

## A.1 A Relation Between Concrete and Abstract Values

Let $v \in V^{\texttt{ppoint}}(\tau)$, $v^\sharp \in V^\sharp(\tau)$ and $\mathcal{A}$ an AI-state. Both values form ASTs, and at the leaves of these ASTs, some of them have unspecified types. We define the relation $v \sim_{\mathcal{A}} v^\sharp$, if $v$ and $v^\sharp$ have an AST of the same shape, and that a leaf in $v$ of unspecified type $\tau_u$ is in the concretisation of the corresponding leaf in $v^\sharp$, in abstract state $\mathcal{A}$.

**Definition 29** *Let $\mathcal{A}$ be an AI-state of the abstract interpretation.*

$$\frac{v \sim_{\mathcal{A}} v^\sharp}{C\ v \sim_{\mathcal{A}} C\ v^\sharp}\ \textsc{Alg} \qquad \frac{}{\text{pp} \sim_{\mathcal{A}} \text{pp}}\ PP$$

$$\frac{\exists (v_1^\sharp, \ldots, v_n^\sharp) \in t^\sharp, \quad \forall 1 \leq i \leq n,\ v_i \sim_{\mathcal{A}} v_i^\sharp}{(v_1, \ldots, v_n) \sim_{\mathcal{A}} t^\sharp}\ \textsc{Tuple} \qquad \frac{f \in F_2^\sharp}{f \sim_{\mathcal{A}} (F_1^\sharp, F_2^\sharp)}\ \textsc{NClos}$$

$$\frac{\exists (\Gamma, p, S, E^\sharp) \in F_1^\sharp,\ E \sim_{\mathcal{A}} E^\sharp}{(\Gamma, p, S, E) \sim_{\mathcal{A}} (F_1^\sharp, F_2^\sharp)}\ \textsc{Clos}$$

$$\frac{v \in V^{ppoint}(\tau) \qquad \tau\ \textit{unspecified} \qquad v \in \gamma(\mathcal{A}, v^\sharp)}{v \sim_{\mathcal{A}} v^\sharp}\ \textsc{Unspec}$$

$$\frac{\Gamma \vDash E \qquad \Gamma \vDash E^\sharp \qquad \forall x \in \operatorname{dom} E,\ E(x) \sim_{\mathcal{A}} E^\sharp(x)}{E \sim_{\mathcal{A}} E^\sharp}\ \textsc{Env}$$

*We write $v \sim_{\mathcal{A}} v^\sharp$ when there exists a finite derivation with conclusion $v \sim_{\mathcal{A}} v^\sharp$*

**Lemma 15** *Let $v \in V^{ppoint}(\tau)$ and $v^\sharp \in V^\sharp(\tau)$, and $\mathcal{A}$ an AI-state. We suppose that the abstract instantiations of the unspecified terms are correct approximations of the concrete instantiations of the unspecified terms. Then, it is true that:*

$$v \sim_{\mathcal{A}} v^\sharp \implies v \in \gamma(\mathcal{A}, v^\sharp)$$

**Proof 10** *Suppose $v \in V^{ppoint}(\tau)$, $v^\sharp \in V^\sharp(\tau)$, $\mathcal{A}$ is an AI-state. We proceed by induction of the derivation tree of $v \sim_{\mathcal{A}} v^\sharp$.*
*To prove the theorem, we proceed by induction on $\pi$*

- *The conclusion rule of the tree of $v \sim_{\mathcal{A}} v^\sharp$ is Const. Therefore, there is a constructor $C$ such that $v = C\ w$, $v^\sharp = C\ w^\sharp$, and $w \sim_{\mathcal{A}} w^\sharp$. Using the induction hypothesis, we get $w \in \gamma(\mathcal{A}, w^\sharp)$. And, by the definition of $\gamma$, one can conclude that $C\ w \in \gamma(\mathcal{A}, C\ w^\sharp)$.*

- *The conclusion rule of the tree of $v \sim_{\mathcal{A}} v^\sharp$ is PP. Therefore, $v = \text{pp}$ and $v^\sharp = \text{pp}$. By definition of $\gamma$, $\text{pp} \in \gamma(\mathcal{A}, \text{pp})$.*

- *The conclusion rule of the tree of* $v \sim_{\mathcal{A}} v^\sharp$ *is* Tuple. *Therefore,* $v = (v_1, \ldots, v_n)$ *and* $v^\sharp = t^\sharp$ *where* $t^\sharp$ *is a set of tuple of abstract values.* $\exists (v_1^\sharp, \cdots, v_n^\sharp) \in t^\sharp$, *such that* $\forall 1 \leq i \leq n,\ v_i \sim_{\mathcal{A}} v_i^\sharp$. *Using the induction hypothesis, we get* $\forall 1 \leq i \leq n,\ v_i \in \gamma(\mathcal{A}, v_i^\sharp)$. *By the definition of* $\gamma$, $(v_1, \ldots, v_n) \in \gamma(\mathcal{A}, t^\sharp)$

- *The conclusion rule of the tree of* $v \sim_{\mathcal{A}} v^\sharp$ *is* NClos. *Therefore* $v = f$ *and* $v^\sharp = (F_1^\sharp, F_2^\sharp)$ *and* $f \in F_2^\sharp$. *By definition of* $\gamma$, *it comes that* $f \in \gamma(\mathcal{A}, v^\sharp)$

- *The conclusion rule of the tree of* $v \sim_{\mathcal{A}} v^\sharp$ *is* Clos. *Therefore* $v = (\Gamma, p, S, E)$ *and* $v^\sharp = (F_1^\sharp, F_2^\sharp)$ *such that* $\exists E^\sharp$, $(\Gamma, p, S, E^\sharp) \in F_1^\sharp$ *such that* $E \sim_{\mathcal{A}} E^\sharp$. *By definition of* $\gamma$, *it comes that* $(\Gamma, p, S, E) \in \gamma(\mathcal{A}, v^\sharp)$.

- *The conclusion rule of the tree of* $v \sim_{\mathcal{A}} v^\sharp$ *is* Unspec. *Because the abstract instantiations of the unspecified terms are correct approximation of the concrete instantiations, by definition* $v \in \gamma(\mathcal{A}, v^\sharp)$.

- *The conclusion rule of the tree of* $v \sim_{\mathcal{A}} v^\sharp$ *is* Env. *Therefore* $\exists \Gamma$ *a typing environment such that* $\Gamma \vDash E$ *and* $\Gamma \vDash E^\sharp$, *and* $\forall x \in \operatorname{dom} E,\ E \sim_{\mathcal{A}} E^\sharp$. *Using the induction hypothesis,* $\forall x \in \operatorname{dom} E,\ E(x) \in \gamma(\mathcal{A}, E^\sharp(x))$. *By the definition of* $\gamma$, *it comes that* $E \in \gamma_{env}(\mathcal{A}, E^\sharp)$.

*This concludes the proof.*

**Lemma 16** $\sim$ *is monotonic in the abstract state and second argument:*

$$\left.\begin{array}{l} v \sim_{\mathcal{A}} v^\sharp \\ v^\sharp \sqsubseteq^\sharp v'^\sharp \\ \mathcal{A} \sqsubseteq^\sharp \mathcal{A}' \end{array}\right\} \implies v \sim_{\mathcal{A}'} v'^\sharp$$

**Proof 11** *By induction on the derivation tree of* $v \sim_{\mathcal{A}} v^\sharp$.

**Lemma 17** *Suppose* $\mathcal{A}$ *is an AI-state, and* ProgTypes *is the set of program types. Suppose* $E \sim_{\mathcal{A}} E^\sharp$ *and there are* $v \in V^{\mathit{ppoint}}(\tau)$ *and* $v^\sharp \in V^\sharp(\tau)$ *such that* $v \sim_{\mathcal{A}} v^\sharp$, *and* $\xi$ *is a set of abstract environments such that* $E^\sharp \in \xi$

$$\left.\begin{array}{l} \text{ProgTypes}, \boldsymbol{prog} \vdash E + p \mapsto v \rightsquigarrow E' \\ \text{ProgTypes}, \boldsymbol{prog} \vdash \xi^\sharp + p \mapsto^\sharp v^\sharp \rightsquigarrow \left\{E_1^\sharp, \ldots, E_n^\sharp\right\} \end{array}\right. \implies \exists 1 \leq i \leq n,\ E' \sim_{\mathcal{A}} E_i^\sharp$$

**Proof 12** *Suppose $\mathcal{A}$, an AI-state, $\tau$ a type, $v$ and $v^\sharp$ such that $v \in V^{ppoint}(\tau)$ and $v^\sharp \in V^\sharp(\tau)$. Suppose $E$ and $E^\sharp$ a concrete environment and an abstract environment respectively such that $E \sim_{\mathcal{A}} E^\sharp$, and $\xi$ is a set of abstract environments that contains $E^\sharp$.*

*Suppose* ProgTypes, $\boldsymbol{prog} \vdash E + p \mapsto v \rightsquigarrow E'$ *and* ProgTypes, $\boldsymbol{prog} \vdash \xi + p \mapsto^\sharp v^\sharp \rightsquigarrow \left\{E_1^\sharp, \ldots, E_n^\sharp\right\}$.

*We proceed by induction on the derivation tree of* ProgTypes, $\boldsymbol{prog} \vdash E + p \mapsto v \rightsquigarrow E'$.

- *The conclusion rule of the tree of* ProgTypes, $\boldsymbol{prog} \vdash E + p \mapsto v \rightsquigarrow E'$ *is* Asn-Wildcard. *Therefore $p = \_$ and $E' = E$. Then, the conclusion rule of* ProgTypes, $\boldsymbol{prog} \vdash \xi^\sharp + p \mapsto^\sharp v^\sharp \rightsquigarrow \left\{E_1^\sharp, \ldots, E_n^\sharp\right\}$ *is A-*Asn-Wildcard *(it is the only rule that handles wildcard). Therefore, it comes that* ProgTypes, $\boldsymbol{prog} \vdash \xi^\sharp + \_ \mapsto^\sharp v^\sharp \rightsquigarrow \xi$. *It comes that $\xi = \left\{E_1^\sharp, \ldots, E_n^\sharp\right\}$, therefore $E^\sharp \in \left\{E_1^\sharp, \ldots, E_n^\sharp\right\}$ and by definition, $E \sim_{\mathcal{A}} E^\sharp$.*

- *The conclusion rule of the tree of* ProgTypes, $\boldsymbol{prog} \vdash E + p \mapsto v \rightsquigarrow E'$ *is* Asn-Var. *Therefore, $p = x$ and $E' = E + p \mapsto v$. Then, the conclusion rule of* ProgTypes, $\boldsymbol{prog} \vdash \xi + x \mapsto^\sharp v^\sharp \rightsquigarrow \left\{E_1^\sharp, \ldots, E_n^\sharp\right\}$ *is A-*Asn-Var *(it is the only rule that handles variables). Therefore, it comes that:* ProgTypes, $\boldsymbol{prog} \vdash \xi + x \mapsto^\sharp v^\sharp \rightsquigarrow \left\{ E'^\sharp + x \mapsto^\sharp v^\sharp \;\middle|\; E'^\sharp \in \xi \right\}$. *Because $E^\sharp \in \xi$, then $\exists E_i^\sharp$ such that $E^\sharp + x \mapsto^\sharp v^\sharp = E_i^\sharp$. By definition of $E \sim_{\mathcal{A}} E^\sharp$,* $\operatorname{dom} E = \operatorname{dom} E^\sharp$. *Therefore,* $\operatorname{dom} E + x \mapsto v = \operatorname{dom} E^\sharp + x \mapsto^\sharp v^\sharp$. *Take $y \in \operatorname{dom} E + x \mapsto v$*

  - *If $y = x$*
    *Then $E + x \mapsto v(x) = v$ and $E_i^\sharp(x) = v^\sharp$. By hypothesis, $v \sim_{\mathcal{A}} v^\sharp$ so $E + x \mapsto v(x) \sim_{\mathcal{A}} E_i^\sharp(x)$*
  - *If $y \neq x$*
    *$E + x \mapsto v(y) = E(y)$, and $E_i^\sharp(y) = E^\sharp(y)$. Because $E \sim_{\mathcal{A}} E^\sharp$, it comes that $E(y) \sim_{\mathcal{A}} E^\sharp(y)$*

  *This proves that $E + x \mapsto v \sim_{\mathcal{A}} E_i^\sharp$*

- *The conclusion rule of the tree of* ProgTypes, $\boldsymbol{prog} \vdash E + p \mapsto v \rightsquigarrow E'$ *is* Asn-Constr. *Therefore, $p = C\ p'$, $v = C\ v'$ and* ProgTypes, $\boldsymbol{prog} \vdash E + p' \mapsto v' \rightsquigarrow E'$. *Because $v = C\ v'$, by definition of $\sim$, $v^\sharp = C\ v'^\sharp$. Then, only rule that applies to* ProgTypes, $\boldsymbol{prog} \vdash \xi + p \mapsto^\sharp v^\sharp \rightsquigarrow \left\{E_1^\sharp, \ldots, E_n^\sharp\right\}$ *is A-*Asn-Constr. *Thus it comes that* ProgTypes, $\boldsymbol{prog} \vdash \xi + p' \mapsto^\sharp v'^\sharp \rightsquigarrow \left\{E_1^\sharp, \ldots, E_n^\sharp\right\}$. *By definition of $C\ v' \sim_{\mathcal{A}} C\ v'^\sharp$, it comes that $v' \sim_{\mathcal{A}} v'^\sharp$. Using the Induction Hypothesis, $\exists E_i^\sharp \in \left\{E_1^\sharp, \ldots, E_n^\sharp\right\}$ such that $E' \sim_{\mathcal{A}} E_i^\sharp$.*

- *The conclusion rule of the tree of* $\text{ProgTypes}, \boldsymbol{prog} \vdash E + p \mapsto v \rightsquigarrow E'$ *is* Asn-Tuple. *Thus* $p = (p_1, \ldots, p_m)$, *and* $v = (v_1, \cdots, v_m)$. *Moreover, the conclusion in the abstract must be A-*Asn-Tuple. *By the definition of* $\sim$, *it comes that* $v^\sharp = t^\sharp$ *where* $t^\sharp$ *is an abstract tuple, and* $\exists (v_1^\sharp, \ldots, v_m^\sharp) \in t^\sharp$, *such that* $\forall 1 \leq i \leq m$, $v_i \sim_{\mathcal{A}} v_i^\sharp$. *By the definition of A-*Asn-Tuple, $\exists \xi_{v_1^\sharp, \ldots, v_m^\sharp}$ *such that:*
  $\text{ProgTypes}, \boldsymbol{prog} \vdash E^\sharp + (p_1, \ldots, p_m) \mapsto^\sharp (v_1^\sharp, \cdots, v_m^\sharp) \rightsquigarrow \xi_{v_1^\sharp, \ldots, v_m^\sharp}$.
  *By induction on* $m$, *it comes that* $\exists E'^\sharp \in \xi_{v_1^\sharp, \ldots, v_m^\sharp}$ *such that* $E' \sim_{\mathcal{A}} E'^\sharp$. *And because* $\xi_{v_1^\sharp, \ldots, v_n^\sharp} \subseteq \bigcup_{(v_1^\sharp, \cdots, v_n^\sharp) \in v^\sharp} \xi_{v_1^\sharp, \ldots v_n^\sharp}$, *this concludes this case.*
- *The conclusion rule of the tree of* $\text{ProgTypes}, \boldsymbol{prog} \vdash E + p \mapsto v \rightsquigarrow E'$ *is* Asn-Unfold. *Thus* $p = C\ p'$, *and* $v = \text{pp}$. *Moreover,* $\text{ProgTypes}, \boldsymbol{prog} \vdash E + p \mapsto (v_0, .., v_{n-1}) \rightsquigarrow E'$ *with* $\boldsymbol{prog}@\text{pp} = C\,(v'_0, .., v'_{n-1})$, $C : (\tau_0 \times .. \times \tau_{n-1}, \tau)$ *and* $v_j = \boldsymbol{if}\ \tau_j \in \text{ProgTypes}\ \boldsymbol{then}\ \text{pp} \cdot \underline{j}\ \boldsymbol{else}\ v'_j$. *Because* $v = \text{pp}$ *and* $v \sim_{\mathcal{A}} v^\sharp$, *then* $v^\sharp = \text{pp}$. *The conclusion rule of the abstract extension of environment is therefore A-Asn-Unfold with* $\text{ProgTypes}, \boldsymbol{prog} \vdash \xi + p \mapsto^\sharp (v_0^\sharp, .., v_{n-1}^\sharp) \rightsquigarrow \left\{E_1^\sharp, \ldots, E_n^\sharp\right\}$ *with* $v_j^\sharp = \boldsymbol{if}\ \tau_j \in \text{ProgTypes}\ \boldsymbol{then}\ \text{pp} \cdot \underline{j}\ \boldsymbol{else}\ v'_j$. *Because* $v_j = v_j^\sharp$, *using the induction hypothesis, it comes that* $\exists E'^\sharp \in \left\{E_1^\sharp, \ldots, E_n^\sharp\right\}$ *such that* $E' \sim_{\mathcal{A}} E'^\sharp$.

*This concludes the induction and the proof.*

## A.2 Intermediate Abstract Interpretation

The issue with our abstract interpretation is that the Spec-Loop returns a locally wrong result: $\bot$. The rule is used to stop an infinite derivation, and to globally compute a fixpoint.

$$\dfrac{\dfrac{\pi', \mathcal{A}, f\ v^\sharp \Downarrow_S^\sharp \bot}{\cdots}\ \textsc{Spec-Loop}}{\pi, \mathcal{A}, f\ v^\sharp \Downarrow_S^\sharp w^\sharp}\ \textsc{Spec}$$

The abstract interpretation of $f\ v^\sharp$ is obviously not $\bot$ as shown in the rule Spec-Loop, but $w^\sharp$ of the Spec rule. This prevents us to do an induction of the derivation tree of the natural semantics of Skel, because we will not be able to conclude on recursive call to a specified function. We introduce a slightly modified abstract interpretation that is locally correct and we prove that it is globally equivalent to our abstract interpretation.

We suppose specified functions do not return values of inductive types. Take an abstract derivation tree $\mathcal{T}^\sharp$. When in $\mathcal{T}^\sharp$, there is a derivation of the form:

$$\dfrac{\dfrac{}{\pi', \mathcal{A}, f\ v^\sharp \Downarrow_S^\sharp \bot}\ \textsc{Spec-Loop}}{\dfrac{\cdots}{\pi, \mathcal{A}, f\ v^\sharp \Downarrow_S^\sharp w^\sharp}\ \textsc{Spec}}$$

We replace it with:

$$\dfrac{\dfrac{}{\pi', \mathcal{A}, f\ v^\sharp \Downarrow_S^\sharp w^\sharp}\ \textsc{Spec-Loop}}{\dfrac{\cdots}{\pi, \mathcal{A}, f\ v^\sharp \Downarrow_S^\sharp w^\sharp}\ \textsc{Spec}}$$

Only $\bot$ has been changed. Therefore we obtain a new derivation tree $\mathcal{T}'^\sharp$ that is equivalent to $\mathcal{T}^\sharp$ because we have reached a fixpoint for the function $f$ on input $v^\sharp$. This change only works when specified functions do not return values with inductive type. Indeed, suppose the Spec-Loop rule returns a symbolic variable $\alpha$ that is traceable. Then the derivation becomes:

$$\dfrac{\dfrac{}{\pi', \mathcal{A}, f\ v^\sharp \Downarrow_S^\sharp \alpha}\ \textsc{Spec-Loop}}{\dfrac{\cdots}{\pi, \mathcal{A}, f\ v^\sharp \Downarrow_S^\sharp \phi(\alpha)}\ \textsc{Spec}}$$

The $\phi$ is some function of the symbolic variable, and we want a solution to the equation $\alpha = \phi(\alpha)$. For non-inductive type, $\phi(\alpha)$ is either of the form $\alpha \sqcup v^\sharp$ or the constant function $v^\sharp$. Either way, $\phi(\bot)$ is a solution to the equation. This explains why Spec-Loop return $\bot$ and returns a correct result globally. If $\phi$ is more complex, this can happen with inductive type, then we do not know how to solve the equation. We did not try to handle the abstract interpretation with inductive type as it was unnecessary for the analyses we wanted to derive. One solution could be to define an abstract interpretation that returns symbolic variables and solve the equation at the end. If the equation are too hard, $\alpha = \top$ would give a correct result to the abstract interpretation.

## A.3 Correctness of the Abstract Interpretation of Terms

**Lemma 18**

$$\begin{array}{l} \Gamma \vDash E, \quad \Gamma \vDash E^\sharp, \quad \Gamma \vdash t : \tau \\ E, t \Downarrow_t v, \quad E^\sharp, t \Downarrow_t^\sharp v^\sharp \qquad\qquad \Longrightarrow\ v \in \gamma(\mathcal{A}, v^\sharp) \wedge v \sim_{\mathcal{A}} v^\sharp \\ E \sim_{\mathcal{A}} E^\sharp \end{array}$$

**Proof 13** *We do an induction on the derivation tree of* $E, t \Downarrow_t v$.

- *The conclusion rule of* $E, t \Downarrow_t v$ *is* VAR.
  *Therefore,* $t = x$, *and* $v = E(x)$. *Therefore,* $v^\sharp = E^\sharp(x)$. *Because* $E \sim_{\mathcal{A}} E^\sharp$, *it comes that* $E(x) \sim_{\mathcal{A}} E^\sharp(x)$.

- *The conclusion rule of* $E, t \Downarrow_t v$ *is* TERMCLOS.
  *Therefore, it comes that* $t = f$, ***val*** $f : \tau_1 \to \tau_2 [= t_0]$ *with* $v = f$. *Moreover, it comes that* $E^\sharp, f \Downarrow_t^\sharp (\{\}, \{f\})$ *and* $v^\sharp = (\{\}, \{f\})$. *Because* $f \in \{f\}$, *we conclude that* $v \sim_{\mathcal{A}} v^\sharp$.

- *The conclusion rule of* $E, t \Downarrow_t v$ *is* TERMUNSPEC.
  *Therefore, it comes that* $t = x$, ***val*** $x : \tau \in \mathbb{S}$ *and* $v \in [\![x]\!]$. *Moreover,* $v^\sharp = [\![x]\!]^\sharp$
  *Because the abstract instantiations are sound over-approximation of concrete instantiation, it comes that* $[\![x]\!]^{ppoint} \sim_{\mathcal{A}} [\![x]\!]^\sharp$ *and* $v \sim_{\mathcal{A}} [\![x]\!]^\sharp$.

- *The conclusion rule of* $E, t \Downarrow_t v$ *is* TERMSPEC.
  *Therefore, it comes that* $t = x$ *and* ***val*** $x : \tau = t_0 \in \mathbb{S}$. *Therefore, we have* $E, t_0 \Downarrow_t v$ *and* $E^\sharp, t_0 \Downarrow_t v^\sharp$ *with* $E \sim_{\mathcal{A}} E^\sharp$. *Using the induction hypothesis, it comes that* $v \sim_{\mathcal{A}} v^\sharp$.

- *The conclusion rule of* $E, t \Downarrow_t v$ *is* ALG.
  *Therefore,* $t = C\ t_0$, *and* $v = C\ v'$, *with* $E, C\ t_0 \Downarrow_t v'$.
  *Moreover,* $E^\sharp, t_0 \Downarrow_t^\sharp v'^\sharp$ *with* $v^\sharp = C\ v'^\sharp$. *Using the induction hypothesis, it comes that* $v' \sim_{\mathcal{A}} v'^\sharp$, *and by the definition of* $\sim$, *we conclude that* $v \sim_{\mathcal{A}} v^\sharp$.

- *The conclusion rule of* $E, t \Downarrow_t v$ *is* TUPLE.
  *Therefore,* $t = (t_1, \ldots, t_n)$ *and* $E, (t_1, \ldots, t_n) \Downarrow_t (v_1, \cdots, v_n)$ *such that* $E, t_i \Downarrow_t v_i$
  *Moreover, by the definition of* $E^\sharp, (t_1, \ldots, t_n) \Downarrow_t^\sharp v^\sharp$, *it comes that* $v^\sharp = \left\{(v_1^\sharp, \ldots, v_n^\sharp)\right\}$ *with* $E^\sharp, t_i \Downarrow_S^\sharp v_i^\sharp$. *By the induction hypothesis, it comes that* $\forall 1 \leq i \leq n$, $v_i \sim_{\mathcal{A}} v_i^\sharp$.
  *By definition of* $\sim$, *it comes that* $v \sim_{\mathcal{A}} v^\sharp$.

- *The conclusion rule of* $E, t \Downarrow_t v$ *is* Clos.
  *Therefore,* $t = (\lambda p \cdot S)$ *and* $v = (\Gamma, p, S, E)$.
  *Because* $E^\sharp, (\lambda p \cdot S) \Downarrow_t^\sharp (\{(\Gamma, p, S, E^\sharp)\}, \{\})$, *it comes that* $v^\sharp = \{(\Gamma, p, S, E^\sharp)\}$. *Because* $E \sim_{\mathcal{A}} E^\sharp$, *we can conclude that* $(\Gamma, p, S, E) \sim_{\mathcal{A}} \{(\Gamma, p, S, E^\sharp)\}$

*This concludes the proof.*

## A.4 Correctness of the Abstract Interpretation

The $\Downarrow_{app}$ and $\Downarrow_S^\sharp$ are defined mutually recursively, thus we should do one induction to prove they are correct. To make it simpler, we split the correctness of the $\Downarrow_{app}$ and $\Downarrow_S^\sharp$ relations in two theorems.

**Theorem 6** *Soundness of the Abstract Interpretation of application*
*If* $\mathcal{A}_0$ *is an AI-State*

$$\begin{array}{l} v_0\ v_1 \Downarrow_{app} w \\ \pi, \mathcal{A}_0, v_0^\sharp\ v_1^\sharp \Downarrow_{app} w^\sharp, \mathcal{A} \\ \forall 1 \leq i \leq 2, \quad v_i \sim_{\mathcal{A}_0} v_i^\sharp \\ \textit{Specified functions do not return values of inductive types} \end{array} \implies \begin{array}{l} \mathcal{A}_0 \sqsubseteq^\sharp \mathcal{A} \\ w \sim_{\mathcal{A}} w^\sharp \end{array}$$

**Proof 14** *Let* $\mathcal{T}$ *be the derivation tree of* $v_0\ v_1 \Downarrow_{app} w$ *and* $\mathcal{T}^\sharp$ *be the derivation tree of* $\pi, \mathcal{A}_0, v_0^\sharp\ v_1^\sharp \Downarrow_{app} w^\sharp, \mathcal{A}$ *We proceed by induction on* $\mathcal{T}$.

- *The conclusion rule of* $v_0\ v_1 \Downarrow_{app} w$ *is* Clos.
  *Therefore* $v_0 = (\Gamma, p, S, E)$. *Because* $v_0 \sim_{\mathcal{A}_0} v_0^\sharp$, $v_0^\sharp = (\Gamma, p, S, E^\sharp)$ *such that* $E \sim_{\mathcal{A}_0} E^\sharp$.
  *The* Clos *rule says that:*

$$\frac{E' = E + p \mapsto v_1 \qquad \dfrac{\mathcal{T}}{E', S \Downarrow_S w}}{(\Gamma, p, S, E)\ v_1 \Downarrow_{app} w}$$

*Moreover, because* $v_0^\sharp = (\Gamma, p, S, E^\sharp)$, *the conclusion rule of the abstract derivation*

*is the rule* Clos*:*

$$\frac{\text{ProgTypes}, \textbf{\textit{prog}} \vdash \left\{E^\sharp\right\} + p \mapsto^\sharp v_1^\sharp \rightsquigarrow \left\{E_1^\sharp, \cdots, E_m^\sharp\right\} \qquad \forall E_i^\sharp \in \left\{E_1^\sharp, \cdots, E_m^\sharp\right\} \qquad \overline{\pi, \mathcal{A}_0, E_i^\sharp, S \Downarrow_S w_i^\sharp, \mathcal{A}_i}^{\mathcal{T}^\sharp}}{\pi, \mathcal{A}_0, (p, S, E^\sharp)\ v_1^\sharp \Downarrow_{app} \sqcup^\sharp u_i^\sharp, \sqcup^\sharp \mathcal{A}_i}\ \textsc{Clos}$$

*With* $\sqcup^\sharp u_i^\sharp = w^\sharp$ *and* $\mathcal{A} = \sqcup^\sharp \mathcal{A}_i$

*Using Lemma 17,* $\exists 1 \leq i_0 \leq m$, $E' \sim_{\mathcal{A}} E_{i_0}^\sharp$*. Using the induction hypothesis, it is true that* $w \sim_{\mathcal{A}_{i_0}} w_{i_0}^\sharp$*, and* $\mathcal{A}_0 \sqsubseteq \mathcal{A}_{i_0}$*. By monotonicity of* $\sim$*,* $w \sim_{\sqcup_i \mathcal{A}_i} \sqcup_i u_i^\sharp$*, and by the property of the abstract union,* $\mathcal{A}_0 \sqsubseteq \sqcup_i \mathcal{A}_i = \mathcal{A}$

- *The conclusion rule of* $v_0\ v_1 \Downarrow_{app} w$ *is* Unspec*.*

$$\frac{\textbf{\textit{val}}\ f : \tau_1 \to \tau_2 \to \tau \in \mathbb{S} \qquad w \in [\![f]\!](v_1)}{f\ v_1 \Downarrow_{app} w}\ \textsc{Unspec}$$

*Therefore,* $v_0 = f$*. Because* $v_0 \sim_{\mathcal{A}} v_0^\sharp$*, then* $v_0^\sharp = f$

$$\frac{\textbf{\textit{val}}\ f : \tau_1 \to \tau_2 \in \mathbb{S} \qquad [\![f]\!]^\sharp(\mathcal{A}_0, v_1^\sharp) = w^\sharp, \mathcal{A}}{\pi, \mathcal{A}_0, f\ v_1^\sharp \Downarrow_{app} w^\sharp, \mathcal{A}}\ \textsc{Unspec}$$

*Because* $[\![f]\!]^\sharp$ *is a sound approximation of* $[\![f]\!]$*, one can conclude that* $w \sim_{\mathcal{A}'} w^\sharp$*, and* $\mathcal{A}\prime \sqsubseteq \mathcal{A}$ *by definition*

- *The conclusion rule of* $v_0\ v_1 \Downarrow_{app} w$ *is* Spec*.*
  *Therefore,* $v_0 = f$*.*

$$\frac{\textbf{\textit{val}}\ f : \tau_1 \to \tau_2 = t \in \mathbb{S} \qquad \emptyset, t \Downarrow_t w \qquad w\ v_1 \Downarrow_{app} v}{f\ v_1 \Downarrow_{app} v}\ \textsc{Spec}$$

*Moreover,* $v_0 \sim_{\mathcal{A}} v_0^\sharp$*, thereby* $v_0^\sharp = (F_1, F_2)$ *and* $f \in F_2$ *The conclusion rule in the abstract derivation tree is either* Spec *or* Spec-Loop

  - *The conclusion rule in the abstract derivation tree is* Spec

*By monotonicity of the update functions and by applying the Induction Hypothesis, the property is true.*

– *The conclusion rule in the abstract derivation tree is* Spec-Loop
*By definition of the abstract tree* $\mathcal{T}^\sharp$*, the result of* $\pi, \mathcal{A}_0, v_0^\sharp\ v_1^\sharp \Downarrow_{app} w^\sharp, \mathcal{A}$ *is equal to the result of* $\varepsilon, \mathcal{A}_0, v_0^\sharp\ v_1^\sharp \Downarrow_{app} w^\sharp, \mathcal{A}$*. Thus, we continue the induction with this new derivation tree and we fallback on the* Spec *case.*

**Theorem 7** *Soundness of the Abstract Interpretation of Skeletons*
*Let* $\mathbb{S}$ *be a Skeletal Semantics with unspecified terms* $Te$ *and unspecified types* $Ty$*, and let* $E$ *and* $E^\sharp$ *be a concrete and abstract environment, respectively. Suppose*

- $\forall x \in Te$*,* $[\![x]\!]^\sharp$ *is a sound approximation of* $[\![x]\!]^{\texttt{ppoint}}$*.*
- $\forall \tau \in Ty$*,* $\gamma_\tau$ *is monotonic.*
- $\boldsymbol{update}^{in}$ *and* $\boldsymbol{update}^{out}$ *are monotonic.*
- *Specified functions do not return values of inductive types*

*Then:*

$$\left.\begin{array}{c} E \in \gamma_\Gamma(\mathcal{A}_0, E^\sharp) \\ E, S \Downarrow_S v \\ \varepsilon, \mathcal{A}_0, E^\sharp, S \Downarrow_S^\sharp v^\sharp, \mathcal{A} \end{array}\right\} \implies v \in \gamma(\mathcal{A}, v^\sharp)$$

**Proof 15** *Proof by induction on the derivation tree of* $E, S \Downarrow_S v$

- *The conclusion rule of* $E, S \Downarrow_S v$ *is* Branch
*Then* $S = (S_1, \cdots, S_n)$*, therefore*

$$\frac{E, S_i \Downarrow_S v}{E, (S_1, .., S_n) \Downarrow_S v}\ \textsc{Branch}$$

*Therefore, the conclusion rule of* $\pi, \mathcal{A}_0, E^\sharp, S \Downarrow_S^\sharp v^\sharp, \mathcal{A}$ *is* Branch*:*

$$\frac{\pi, \mathcal{A}_0, E^\sharp, S_i \Downarrow_S^\sharp v_i^\sharp, \mathcal{A}_i}{\pi, \mathcal{A}_0, E^\sharp, (S_1 \cdots S_n) \Downarrow_S^\sharp \sqcup_i^\sharp v_i^\sharp, \sqcup_i^\sharp \mathcal{A}_i}\ \textsc{Branch}$$

*Using the Induction Hypothesis,* $v_i \sim_{\mathcal{A}_i} v_i^\sharp$*, and by the monotonicity of* $\sim$*:* $v_i \sim_{\sqcup_i^\sharp \mathcal{A}_i} \sqcup_i^\sharp v_i^\sharp$

- *The conclusion rule of* $E, S \Downarrow_S v$ *is* LetIn
  *Then* $S = \textbf{\textit{let}}\ p\ =\ S_1\ \textbf{\textit{in}}\ S_2$*, therefore*

$$\frac{E, S_1 \Downarrow_S v \qquad \vdash E + p \mapsto v \rightsquigarrow E' \qquad E', S_2 \Downarrow_S w}{E, \text{let}\, p = S_1 \,\text{in}\, S_2 \Downarrow_S w}\ \textsc{LetIn}$$

  *Therefore, the conclusion rule of* $\pi, \mathcal{A}_0, E^\sharp, S \Downarrow_S^\sharp v^\sharp, \mathcal{A}$ *is* LetIn*:*

$$\frac{\begin{array}{c}\pi, \mathcal{A}, E^\sharp, S_1 \Downarrow_S^\sharp v^\sharp, \mathcal{A}' \\ \text{ProgTypes}, \textbf{\textit{prog}} \vdash \left\{E^\sharp\right\} + p \mapsto v^\sharp \rightsquigarrow \left\{E_1^\sharp, \cdots, E_n^\sharp\right\} \\ \pi, \mathcal{A}', E_i^\sharp, S_2 \Downarrow_S^\sharp w_i^\sharp, \mathcal{A}_i\end{array}}{\pi, \mathcal{A}, E^\sharp, \text{let}\, p = S_1 \,\text{in}\, S_2 \Downarrow_S^\sharp \sqcup^\sharp w_i^\sharp, \sqcup^\sharp \mathcal{A}_i}\ \textsc{LetIn}$$

  *Using Lemma 17,* $\exists 1 \leq i \leq m$*,* $E' \sim_{\mathcal{A}_i} E_i$*. Therefore, using the Induction Hypothesis,* $v \sim_{\mathcal{A}_i} w_i^\sharp$ *and by the monotonicity of* $\sim$*:* $v \sim_{\sqcup_i^\sharp \mathcal{A}_i} \sqcup_i^\sharp w_i^\sharp$*.*