Université Libre de Bruxelles

Computer Science Department

MEMO-F403 Preparatory work for the master thesis

An Efficient and Parallel Abstract Interpreter in Scala — Preparatory Work —

Olivier Pirson – opi@opimedia.be

Academic year 2016 – 2017
(first year of the master)

August 21, 2017
(Little corrections: September 12, 2019)

Vrije Universiteit Brussel

Promotors **Coen De Roover**
Wolfgang De Meuter
Advisor **Quentin Stiévenart**

All documents and LaTeX sources are available on Bitbucket:
https://bitbucket.org/OPiMedia/
efficient-parallel-abstract-interpreter-in-scala-preparatory

Abstract

The Abstracting Abstract Machine (AAM) technique is used to perform static analysis by abstract interpretation.

Abstract interpretation was created in the late 1970s by Patrick and Radhia Cousot [CC77] [Cou78] [CC79]. They were honored by several prizes for that. It is now a standard method to make static analysis of programs, both theoretically and practically. This makes it possible to prove correctness of programs or some properties of their behavior.

However, apart from specific [1] types of programming languages or programs, this kind of analysis is difficult to scale for large programs (but nevertheless usual).

The goal of the future master thesis will be to explore possibilities to parallelize an AAM developed in Scala and targeting for Scheme programs. It will be to identify slow parts of the generation of the state graph and to parallelize in order to improve it. To compare the various possible strategies and to describe how to move from a sequential AAM to an efficient parallel version.

This preparatory work focuses on the comprehension of concepts and foundations required by the author to carry out its task the next academic year.

We begin by exposing the context and introducing the fundamental concepts of abstract interpretation. After that we will define mathematical foundations. Then we will discuss about the parallelization problem and will relate existing works on parallelization of AAM. Finally, the ongoing work will be mentioned as well as remaining research.

Several lists of References (abbreviations, symbols, index, etc.) are available at the end of the document that may be useful to understand some classical notations and to search definitions.

This *orange* color will be used to mark definitions.
These gray and blue colors will indicate internal and external links.

Keywords: static analysis, abstract interpretation, AAM, lattice, Scheme, parallelization, concurrency

[1] A major example, *Astrée*, in particular developed by Patrick Cousot and used in industry [Bou08], "is a static code analyzer that proves the absence of runtime errors and invalid concurrent behavior in safety-critical software written or generated in C." [Abs]

Résumé

La technique d'Abstracting Abstract Machine (AAM) est utilisée pour effectuer des analyse statiques par interprétation abstraite.

L'interprétation abstraite a été créée à la fin des années 70 par Patrick et Radhia Cousot [CC77] [Cou78] [CC79]. Ils ont été honorés par plusieurs prix pour cela. C'est maintenant une méthode standard pour effectuer l'analyse statique de programmes, à la fois en théorie et en pratique. Ce qui permet de prouver la correction des programmes ou certaines propriétés de leur comportement.

Cependant, en dehors de types de langages de programmation ou de programmes spécifiques [2], ce type d'analyses est difficile à mettre à l'échelle pour les gros programmes (mais néanmoins usuels).

L'objectif de ce futur mémoire sera d'explorer les possibilités de paralléliser une AAM développée en Scala et ciblant les programmes Scheme. Il s'agira d'identifier les parties lentes de la génération du graphe d'états et de le paralléliser pour l'améliorer. De comparer les diverses stratégies possibles et de décrire comment passer d'une AAM séquentielle à une version parallèle efficace.

Ce travail préparatoire se concentre sur la compréhension des concepts et des fondements nécessaires à l'auteur pour mener à bien sa tâche l'année scolaire prochaine.

Nous commençons par exposer le contexte et introduire les concepts fondamentaux de l'interprétation abstraite. Après quoi nous définirons les fondements mathématiques. Ensuite nous évoquerons le problème de la parallélisation et nous évoquerons les travaux existants sur la parallélisation d'AAM. Enfin, le travail en cours sera mentionné ainsi que la recherche restante.

Plusieurs listes de References (abréviations, symboles, index, etc.) sont disponibles à la fin du document, ce qui peut être utile pour comprendre certaines notations classiques et rechercher les définitions.

Cette couleur *orange* sera utilisée pour marquer les définitions.
Ces couleurs grise et bleue indiqueront les liens internes et externes.

Mots-clés : analyse statique, interprétation abstraite, AAM, treillis, Scheme, parallélisation, programmation concurrente

[2] Un exemple majeur, *Astrée*, notamment développé par Patrick Cousot et utilisé par l'industrie [Bou08], "est un analyseur statique de code qui prouve l'absence d'erreurs à l'exécution et de comportement concurrent invalide dans les logiciels critiques écrits ou générés en C." (traduit de l'anglais) [Abs]

Acknowledgements

Thanks to Astrology (!) to have put computers on my way, very long time ago. And thanks to computers in return, for impregnating me with rationalism.

Thanks to Arabella Brayer for the craziest idea: to resume my studies!

Thanks to Sci-Hub to free the knowledge.

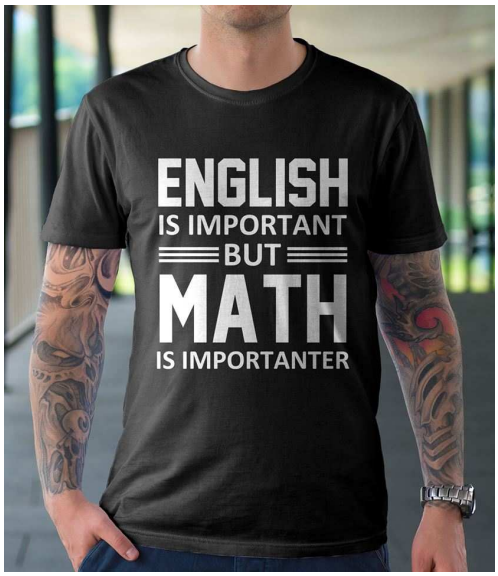
Thanks to professor Gilles Geeraerts for the bad news but especially the good ones: there is no real specialist in the domain of programming languages at the ULB, but there is the *Software Languages Lab* at the VUB.

Thanks to my promotors Coen De Roover and Wolfgang De Meuter, who accepted me in this laboratory for my future master thesis.

Many thanks to my advisor Quentin Stiévenart, who gave me the first references and answers in order to begin my work, and next for its advice and corrections. This document is largely inspired by first chapters of its master thesis.

Thanks to all (Carole, Cédric, Marion, Stefania...) those who have helped me to correct my English, at times very unlikely. It was a real challenge for me to write this work.

My apologies for the mistakes that will remain.



https://teespring.com/math-im#pid=389&cid=100029&sid=front

Table of Contents

"*A language that doesn't affect the way you think about programming, is not worth knowing.*"

— Alan J. Perlis, *Epigrams on Programming*, *SIGPLAN Notice*, September 1982

Chapter 1

Overview

Nowadays computers and programs are everywhere. For some anodyne tasks, but as well as for a lot of very critical ones. It is crucial that they run perfectly correctly.

In the classical example of the aviation domain, the least mistake can have catastrophic consequences. Other common examples are medical assistance, management of nuclear power stations, road traffic control, banking flows, stock exchange management, etc. We also want that our washing machines run correctly, and our smartphones, television sets, etc. Programs are really everywhere. And "*One Ring to rule them all*" [3], the Internet is also a gigantic masterpiece composed of a cloud of programs.

1.1 From execution on some instances to global analysis of behaviour

> "*The first moral of the story is that program testing can be used very effectively to show the presence of bugs but never to show their absence.*"
>
> — Edsger W. Dijkstra, *On the reliability of programs.* (EWD303)

To be sure that a program runs correctly on *each* instance [4] we have to be sure that it runs correctly on *all* instances. It is obvious to say, but it is a big challenge.

One way to avoid mistakes in a program is to execute it on some instances, well chosen to explore different paths on this program. There exists several techniques, named *testing*, with different degrees of performance and precision (unit testing, integration testing...). However with such techniques, one cannot know if any problematic case was missed. And except on particular programs it is impossible to cover all instances, because there is a combinatorial explosion of the number of possibilities. This kind of tests is really important, but we need something better.

The thesis of Olivier Bouissou [Bou08] summarizes the cycle of program development with different possible types of tests associated on each step. It also recalls some famous software failures, like the explosion of Ariane 5 at its first launch. During the investigation after this accident another type of verification was made

[3] J. R. R. Tolkien, *The Lord of the Rings*, 1954.

[4] In accordance with the terminology of the computational complexity theory we call one data of a problem an *instance* of this problem. And we use the same term for the input of a program.

(in addition to a manual code analysis). Instead of executing the program on some instances, it was submitted to a static analysis to study its global behaviour. This study found the reason of the accident. The bug was the conversion of a floating number too big to become an integer on the architecture used. Too late in this case. Nevertheless this example shows the utility of static analysis, if it is done at the right time, before the programs are used.

Beyond the necessity to verify the correctness of programs, it is interesting to prove properties of their behaviour. This is useful to make automatic transformations and for optimizations. This may be necessary to ensure that our correct programs are executed quickly enough.

A *dynamic analysis* of a program is made at runtime, consequently on particular instances. On the contrary a *static analysis* is made without executing the program. Moreover its goal is to be exhaustive, to *prove* some properties of the program rather than to check specific executions of the program.

To make these analysis we want to use computers themselves to help us to check that they run properly. After all the role of computers is to automate work. [5] Anyway we are incapable ourselves of doing this work without their help. Yet computers cannot do everything. Because everything is not automatable.

1.1.1 The wall of the undecidability

"Eternity is a very long time, especially towards the end."
— Woody Allen

Let's start by reminding an impossibility. A major result of computer science informs us that all interesting properties about the behavior of a program are *undecidable*, which means there does not always exist a program to always decide whether such properties are true or not.

Theorem 1 (Rice's theorem) Any non-trivial semantics property of a program is undecidable. [6]

A *trivial property* is a property such that the answer is always the same.
A *semantics property* of a program is about its behavior.

The princeps example is the famous halting problem. Its undecidability was proved by Alan Turing in 1936. It is impossible to write a program that always says if a program [7] stops, starting from one instance, or if runs forever.

[5] The French denomination of computer science (*informatique*) highlights the automatic processing of information (*traitement automatique de l'information*). While the English denomination emphasizes the computability side. Even though basically it is the same thing.

[6] The theorem proved by Henry Gordon Rice in 1953 [Ric53] should be formulated in more formal manner. We write here the result to make sense in our context.

[7] Another program, or possibly the same program. This is the nub of matter, a form of self-reference being used for the demonstration.

More precisions about these fundamental concepts of calculability are available in the book of Michael Sipser [Sip12].

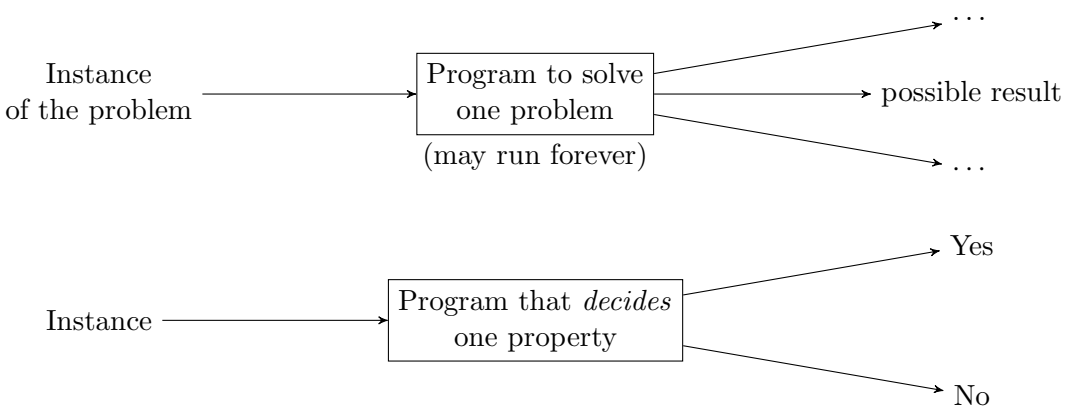


Figure 1.1 – General representation of a program to solve one general problem for one instance, and specific case of a *decider* that *always* solve one *decision problem* (i.e. a problem that accepts only the two answers *yes* or *no*) *for each* instance.

We are interested here in analyzers that deal with other programs in input to study the behaviour of these programs (and that on all possibles instances for each of these programs).

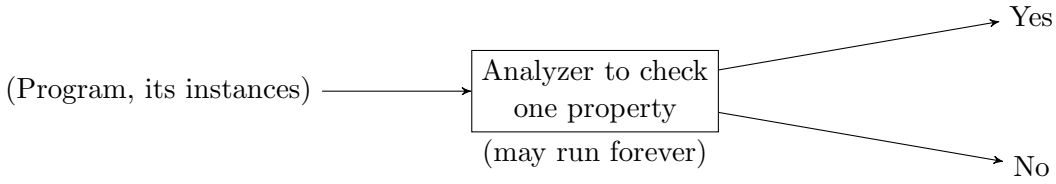


Figure 1.2 – An analyzer is a program that get a program in input.

If we still want to be able to check the correctness of our programs and prove some properties on their behaviour then we must give up some aspects of our claims. We allow the analyzer to respond "*I don't know*" [8], and so we remain in the domain of the decidable.

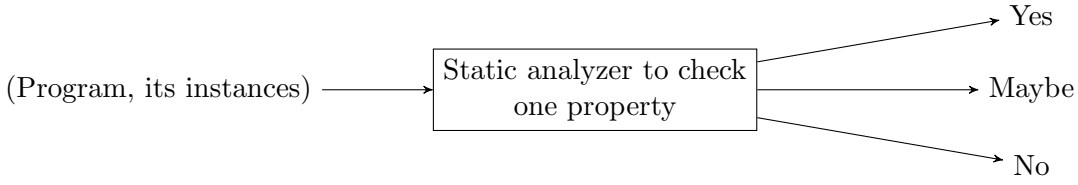


Figure 1.3 – For a static analyzer it is allowed to answer that it does not know, to avoid infinite running.

[8] It is like an absence of answer. Another point of view is to say that the analyzer can answer by the tautology: "Yes *or* no".

It is always possible, because there always exists a program that can answer "*I don't know*" for each instance. Obviously such a program is useless.

The first difficulty will be to find trade-offs between calculability and utility.

1.1.2 Soundness guarantee

Since it is impossible to have both, we must choose between soundness and completeness.

(I) *Soundness*: all properties of programs proved by the analyzer are true, i.e. the analyzer only gives true results.

(II) *Completeness*: all true properties of programs are provable by the analyzer, i.e. the analyzer is capable to prove all true results.

We want *to prove* correctness of programs or some properties of their behavior. We want to be sure that a positive result (*yes* answer) given by the analyzer to be true. We choose soundness. Consequently we accept that when it gives a negative result, this answer means *maybe not*.

In other terms, we choose to reject false positive errors and we accept to obtain false negative errors.

(I) *False positive*: when the answer given is yes while the correct answer is no.

In a context of question about the correctness of a program, this means there are undetected problems.

In a context of error detection it is a *false alarm*, which means that the program is correct although that the analysis said that no.

(II) *False negative*: when the answer given is no while the correct answer is yes.

In a context of question about the correctness of a program it is a *false alarm*, which means that the program is correct although that the analysis said that no.

In a context of error detection, this means there are undetected errors.

In practice analyzers can answer *yes*, *no* or *maybe not*. In the context of the correctness of programs, that means *yes the program is correct*, *no the program is incorrect and this bug has been identified* or *maybe the program is correct because no bug have been identified, but maybe there exist an unidentified bug.*

Testing answers to the opposite question. Instead of answering the question *my program is it correct?* testing answers to the question *is there a bug in my program?*

1.1.3 Concrete interpretation, the starting point

A *concrete interpretation* (its execution by a concrete interpreter) of one program on one instance goes successively through by states from the initial state to the

terminal state (if it does not run forever). We call this (possibly infinite) succession of states the *trace* of the program for this instance.

A interpreter is also called an *abstract machine* (*AM*). *Abstract* machine here is opposed to a physical computer.

With the *small-step* semantics [Mig11] the concrete interpretation is expressed by

- *inject* the program e into the concrete interpreter in an initial state s_0,
- with a *transition function* to move from this state to the next one,
- and so on, until reach a final state, or run forever.

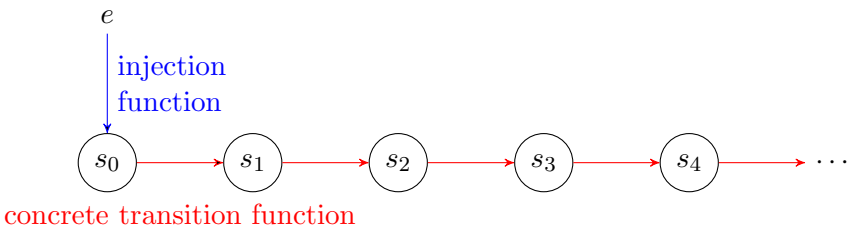


Figure 1.4 – A trace, a concrete interpretation with small-step semantics, for one instance.

This trace may be finite or infinite, depending on the program but also on the instance given to this program.

We want to prove one property of the behaviour of a program (for example its correctness), so we interested in *all* its traces. And we want check that all these traces do not cross prohibited states. As already mentioned testing cannot check all cases. Even if in practice the number of cases is finite, it is far too large to check of all them.

The left part of the figure 1.5 shows an illustration of a correct program: all its traces are acceptable.

On the right, an illustration of an incorrect program: one trace crosses prohibited state, in red.

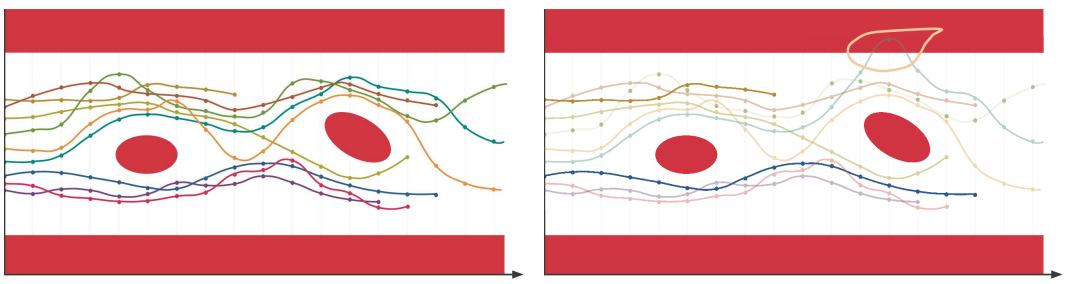


Figure 1.5 – Illustration of a correct and an incorrect program.
(These images were made from images of the conference [Cou08].)

1.1.4 Abstraction to simplify

— René Magritte, *Le Calcul Mental*, 1940 [9]

Instead of checking if all traces are in inside the authorized zone, i.e. if the semantics of the program corresponds to its specification, we will abstract the semantics. [10]

The left part of the figure 1.6 shows an illustration of a correct abstract interpretation: the abstract semantics in green has no intersection with the prohibited zone in red and all concrete traces are in the green zone, therefore all concrete traces are acceptable.

On the right, an illustration of an incorrect abstract interpretation: one concrete trace escapes to the abstract semantics, so we cannot prove the correctness of the program. And in this case, the program is incorrect because this concrete trace crosses prohibited state.

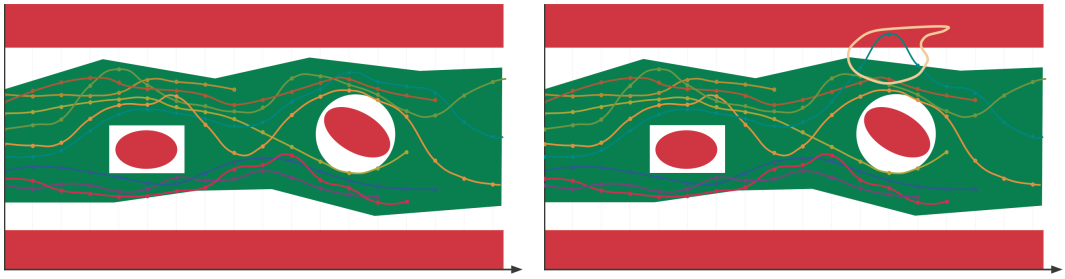


Figure 1.6 – Illustration of a correct and an incorrect abstract interpretation. (These images were made from images of the conference [Cou08].)

To obtain a static analysis by *abstract interpretation* we make two simplifications. First we reduce the concrete infinite state space to an abstract *finite* state space. The abstract interpreter will work with abstract values, which represent an over-approximation of the concrete values. In general we will use the hat symbol $\widehat{\ }$ to express that an element is an abstraction (or a function on abstractions).

Second we mimic the concrete small-step semantics on this abstract state space. But instead of going from one state to another state, the abstract transition function goes from one state to all (directly) reachable states.

One technique to build an abstract interpreter from a language semantics is called *abstracting abstract machine* (*AAM*). This is the technique that we will use.

[9] http://www.artnet.com/artists/ren%C3%A9-magritte/le-calcul-mental-oU6yWQzE-ERgxZSTgANE-g2

[10] The recording of the conference of Patrick Cousot [Cou08] is a pleasant way, in French, to have an introduction to the abstract interpretation.

A program trace is now approximated by a finite *state graph*, which may contain cycles. This state graph being finite, it ensures that the abstract interpretation is decidable and can be computed in a finite amount of time.

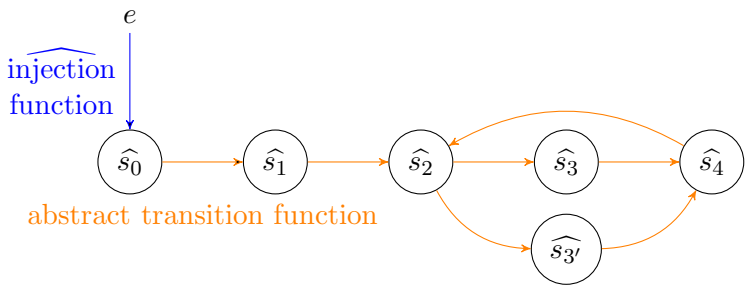


Figure 1.7 – State graph, abstract interpretation with small-step semantics, for *all* instances.

It should be noted that the abstract result makes two over-approximations of the concrete result. Abstract values over-approximate concrete values. And the state graph forgets some order of states present in the trace. Besides it summarizes the global behavior of the program, on all its instances.

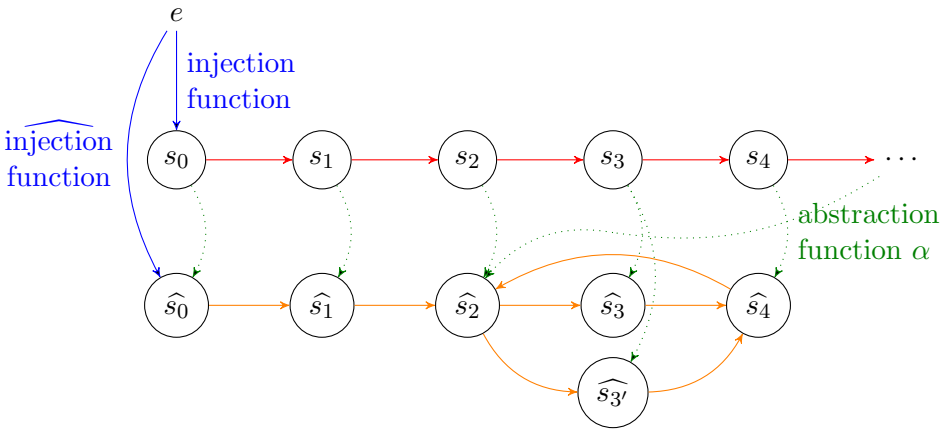


Figure 1.8 – "*The abstract simulates the concrete*" [Mig11], for *all* instances.

Sign example A classical example [Mig12]. to illustrate the principle of abstraction, we will represent integer numbers by their sign. We transform concrete values in abstract values by a *abstraction function* generally noted α: [11]

$$\mathbb{Z} \quad \xrightarrow{\alpha} \quad \widehat{\mathbb{Z}} = \mathcal{P}(\{-, 0, +\}) = \{\emptyset, \{-\}, \{0\}, \{+\}, \{-, 0\}, \{-, +\}, \{0, +\}, \{-, 0, +\}\}$$

We also define an abstract addition $\widehat{+}$ on the abstract integers which verifies in particular these equalities: [12]

[11] We work theoretically on the infinite set of values although in practice programming language implementations can only handle a finite subset of these values.

[12] Do not confuse the meaning of symbols. A same symbols may be have several meanings. Here and in the rest of this example, the meaning of the symbol $-$ may be is the sign of a concrete integer like in -42 or the meaning of a negative abstract value like in $\{-, 0\}$. It could be also the sign of

$\{-\} \widehat{+} \{-\} = \{-\}$
$\{-\} \widehat{+} \{+\} = \{-, 0, +\}$
$\{-, 0\} \widehat{+} \{-\} = \{-\}$
...

With this very simple abstraction we can study behaviour of some use of the addition + on integers. [13]

We can deduce that the sign of $-36 + -6$ is $-$, by $\alpha(-36) \widehat{+} \alpha(-6) = \{-\}$. This is precisely the correct sign of -42.

We see that this abstraction may be not precise enough to answer to all questions, we cannot deduce the sign of $-42 + 42$ because $\alpha(-42) \widehat{+} \alpha(42) = \{-, 0, +\}$.

Precision The *precision* of an abstraction characterize its capacity to analyze properties. More the abstraction is close to its original concrete model and more the abstraction can be ideally useful to study what we want.

Another classical abstractions We can use an interval to represent a number. For example 42 may be approximated by $[0, 100]$.

With intervals it is possible to have a perfect precision, like $[42, 42]$. Or less and less precision, like $[0, 100]$, $[0, +\infty]$, $[-\infty, +\infty]$.

Note that $[0, +\infty]$ represents the all naturals set $\mathbb{N}$ and $[-\infty, +\infty]$ the all integers set $\mathbb{Z}$. In the same way $[1, \infty]$ corresponds to the sign $+$.

More generally yet, we can approximate a value by a set of this type of values. And more general yet, approximation of a value by its type (e.g. Integer).

The difficulties to have both an approximation simplifying enough to be calculable in practice (to have a good complexity) and good enough to be ideally capable to check some properties (to have a good precision), leads to a large variety of type of abstractions. Polygons and polyhedrons for example allow to combine inequalities. In a similar manner it is also possible to combine congruence equalities. [Cou00]

This second difficulty, the most important in practice, is to find trade-offs between complexity and precision.

1.1.5 Fixed point search

In general, a *fixed point* of a function f is a x such that $f(x) = x$.

Programs generally contain loops or recursion, thus their state graphs contains cycles. To deal with this problem we will have to find fixed points This task may be complicated and may require new approximations.

We will see in the chapter 2 some mathematical conditions and structures that can to guarantee existence of these fixing points.

---

the concrete subtraction operator like in $42 - 36$ and the sign of the abstract subtraction operator like in $\{0\} \widehat{-} \{+\}$.

[13] Such a simple abstraction may nevertheless be useful. A very common verification is that check if an index on an array is really in this array. If the sign of the index is not negative ($-$) then we have the guarantee that the program do not try to access to a negative index.

1.1.6 The cost of the complexity

After all these approximations, we obtain a state graph which allows us to answer some questions about the behaviour of our program. Our approximated problems become decidable, and ideally solvable.

In practice our computers have limited memory, and we have limited time. In general these limitations depend on the goal (several weeks/months are acceptable for the verification of a rocket, but at most few seconds/minutes for a static analysis in an integrated development environment.)

The complexity to prove a behaviour of a program is high. Even on small programs the complexity explodes quickly. For reals programs, i.e. usually large programs, the complexity is often gigantic. However it is precisely this kind of programs that we would like to be able to analyze.

A natural idea is to believe that a weak precision can be calculated quickly. It would seem that it is not so simple.

For specific problems it is eventually possible to make a good choice of abstraction and/or optimize the implementation of the analyzer. However a real solution would be to have a general usable solution. If we want to improve the general techniques of abstract interpretation as presented before, we have to formalized them precisely. It will also be done in the chapter 2.

1.2 The *Scheme* programming language, as the target

Functional programming languages are generally not mainstream. Perhaps because of a restrictive "mathematical" aspect that repel them. And also a very refined syntax composed by a lot of nested parentheses.

Let Abelson, Sussman and Sussman, the authors of the classic *wizard book*, give an answer to that: "*If Lisp* [14] *is not a mainstream language, why are we using it as the framework for our discussion of programming? Because the language possesses unique features that make it an excellent medium for studying important programming constructs and data structures and for relating them to the linguistic features that support them. The most significant of these features is the fact that Lisp descriptions of processes, called procedures, can themselves be represented and manipulated as Lisp data. [...] As we shall discover, Lisp's flexibility in handling procedures as data makes it one of the most convenient languages in existence for exploring these techniques. The ability to represent procedures as data also makes Lisp an excellent language for writing programs that must manipulate other programs as data, such as the interpreters and compilers that support computer languages. Above and beyond these considerations, programming in Lisp is great fun.*" [ASS96]

It is relatively easy to implement a (concrete) *Lisp* interpreter in *Lisp* itself.

Linked lists are a fundamental data structure in *Lisp*. And the program itself is a such list. This equivalence between source code and data is called *homoiconicity*.

[14] Abelson, Sussman and Sussman talk about *Lisp*, nevertheless in fact all the book uses *Scheme*, a direct derived of the *Lisp* family language.

1.2.1 *Scheme* presentation

Scheme is a functional programming language directly descending of *Lisp* and retaining its minimal syntax (with some fundamental design features modifications like lexical scoping). "*It was designed to have an exceptionally clear and simple semantics and few different ways to form expressions. A wide variety of programming paradigms, including imperative, functional, and message passing styles, find convenient expression in Scheme.*" [KCR98]

Standards There are a lot of implementations of *Scheme*, some with extensions (like *Racket* [comd]). There are also several successive standards, the official IEEE Standard of 1990 [com90], and since, *Revised*^{n} *Report on the Algorithmic Language Scheme* (RnRS).

Quick enumeration of its features (several introduced for the first time by this language [KCR98]):

- homoiconicity,
- capability to manipulate mutable data (contrary to a purely functional programming language like *Haskell*),
- *strong typing*: there is no implicit conversion,
- *latent typing* (or *dynamic typing*): types are associated with values instead variables,
- *lexical scope* (or *static scope*): bindings of variables in a function are independent of the caller context,
- *closure* (or *lexical closure*, *function closure*): function associated with its environment,
- *first-class function*: functions are elements of the language like the other types,
- *continuation*: abstract representation of the program state,
- *first-class continuation*: continuations are elements of the language like the other types,
- *properly tail-recursive*: tail-recursive calls are transformed to iterations,
- delayed evaluation: allow lazy evaluation and call by need,
- shared namespace for procedures and variables,
- the order of evaluation of procedure arguments are not fixed,
- primitive numbers are arbitrary large,
- *hygienic macro*: there is no accidental capture of identifiers in macro expansion,
- ...

1.2.2 *Scheme* is a good target to experiment with static analyzers

Only the essential core of the R5RS [KCR98] will be considered here. [15] Next version of RnRS begins to complexify the language and losts its minimalist design philosophy. Nor do we consider macros.

[15] Previous master thesis from ULB and VUB studied AAM for non deterministic Scheme extensions: concurrency by Quentin Stiévenart [Sti14] and "*JavaScript-like objects and events*" by Jonas De Bleser [De 16].

Note that Quentin Stiévenart had studied sequential AAM to analyze concurrent programs. The subject here is the opposite: to study parallelized AAM to analyze sequential programs.

To build a (concrete) *Scheme* interpreter only few fundamental forms have to be necessarily implemented. The other forms of the language can be composed from these few fundamental forms. Thus only few abstract forms have to be required to build an abstract interpreter.

Despite its simplicity *Scheme* contains characteristics "*to support most of the major programming paradigms in use today*" [KCR98]. We are particularly interested by its capacity to manipulate mutable data (that causes side-effects), closures and first-class functions, because these possibilities of programming languages are difficult to statically analyze.

Its simplicity is a double advantage: exploration on Scheme is simple to implemented, and exploration on Scheme is representative for more complex programming languages. For us it is a bit of a perfect language to manipulate it.

More contextual, it is also a programming language taught at the VUB.

It is interesting to note that this simplicity had not been premeditated by the creators of the language [SS98]. They were planning something complicated, and realized that it was simple. They discovered the powerful simplicity of the λ-calculus to express programming languages. *Scheme* may be summarized in this maxim: "*Scheme is only syntactic sugar* [16] *on the* λ*-calculus.*" This mathematical model of computation will be briefly presented in chapter 2.

1.3 The *Scala* programming language, for the implementation of the AAM

Scala is the programming language chosen by Quentin Stiévenart to implement *Scala-AM* [Sti17] during his PhD. We think that this is a good choice.

Scala [comb] is also a functional programming language. One of its great forces is to associate the two major paradigms usually opposed that are the object paradigm and the functional paradigm.

Its standard implementation runs on the Java Virtual Machine (JVM), so programs are portable. Moreover it is possible to take advantage of all the rich *Java* ecosystem. By now, *Java* also integrates functional concepts, but *Scala* is a more modern programming language, directly designed to integrate this paradigm. Its contains many useful features like static and strong typing with type inference, immutability, pattern matching, first-class functions, lazy evaluation, etc.

The choice of *Scala* is more important for the future work as it is particularly well suited to parallel programming. Aleksandar Prokopec explains in its book *Learning Concurrent Programming in Scala* [Pro17] that the nice features of *Scala* allows to implements various concurrency model in frameworks as such manner that they can used like basic language features. In the future work we will explore the actor model with the *Akka* framework [coma].

[16] *syntactic sugar* is unnecessary syntactic forms added for convenience.

Scala is less widespread than *Java* but it is becoming more popular. And like *Scheme*, it is a programming language taught at the VUB. [17]

1.4 Abstracting Abstract Machines

In a Web article [Mig12] Matthew MIGHT shows how to implement an abstracting abstract machine (AAM), step by step. In this short article, *Scheme* (in fact *Racket* [comd]) is not used as the target language. It is used to implement the AAM. The language analyzed is the simple language of arithmetic expressions, by the sign abstraction illustrated in the subsection 1.1.4 of this document. [18]

The method is quite straightforward. It consists in implementing in practice which was explained in theory during this chapter.

The first step is to define the semantics of arithmetic expressions.

Then he implements a tiny interpreter, an abstract machine (AM), for this concrete semantics. [19]

After that he duplicates the interpreter and he modifies it in order to adapt it to the abstract sign semantics. The result is an AAM. In fact it is just an AM for another semantics that is the abstract semantics.

1.4.1 Scala-AM

Scala-AM [Sti17] is a framework implemented in *Scala* to build AAMs. It is both a concrete interpreter and an abstract interpreter. Its implementation was designed to be general and modular, with the aim to support different semantics and abstractions. An other aim was to facilitate the work on different aspects. It is possible for "*language designers, static analysis developers and machine abstraction experts to combine their efforts.*" [Sti+16]

The paper *Scala-AM: A Modular Static Analysis Framework* [Sti+16] illustrates that with an AAM for a static analysis of Scheme.

As this implementation was not designed with the goal to be fast, the future work will be to study this implementation and will have to identify slow parts, to parallelize it. Eventually better states representation will be explored to facilitates the parallelization.

[17] Martin ODERSKY, the designer of the *Scala* language proposes two interesting MOOCs to approach *Scala* and the functional programming: *Functional Programming Principles in Scala* [Ode16b] and *Functional Program Design in Scala* [Ode16a].

[18] In a recording [Mig14] of a lecture he shows that step by step, live.

[19] In the *wizard book* [ASS96] a (concrete) *Scheme* interpreter is implemented in *Scheme* itself.

Chapter 2

Mathematical foundation

This chapter exposes some mathematical structures to have a well defined theoretical frame of notions presented in the chapter 1.

2.1 Partially ordered sets and lattices [20]

This somewhat indigestible section summarizes basic definitions about partially ordered sets. More details and properties are available in the book of Davey and Priestley [DP02].

It is a part of the domain theory initiated by Dana Scott in the late 1960s to formalized the denotational semantics of the λ-calculus. These structures will be used to formalize abstractions and the fixed point search.

2.1.1 Partially ordered sets

Let S be a set.

Definition 1 (Relation) A *binary relation* on S is a subset of $S \times S$.

Definition 2 (Partially order) An *partially order* (or *poset*) on S is a binary relation $\sqsubseteq$ on S such that, $\forall x, y, z \in S$:

(i) $x \sqsubseteq x$ (*reflexivity*)

(ii) $(x \sqsubseteq y) \wedge (y \sqsubseteq x) \Rightarrow (x = y)$ (*antisymmetry*)

(iii) $(x \sqsubseteq y) \wedge (y \sqsubseteq z) \Rightarrow (x \sqsubseteq z)$ (*transitivity*)

An *partially ordered set* $(S, \sqsubseteq)$ is a set S with an order $\sqsubseteq$.

The idea is that x and y are not necessarily comparable, but if they are then $x \sqsubseteq y$ expresses that x is lower or equal to y.

[20] This is the order notion of lattice (*un treillis* in French) and not the geometric notion (*un réseau* in French).

The set S may be any set. However we will use these concepts with sets of sets (in particular with powersets). That is why we use the symbol $\sqsubseteq$ (and others symbols in the following) that looks like the inclusion symbol $\subseteq$.

$\forall$ set $E : (\mathcal{P}(E), \subseteq)$ is a partially ordered set.

A HASSE *diagram* is a simple way to represent a finite partially order set. Each element is represented by a vertex and each edge between two vertices x and y represents $x \sqsubseteq y$ if the edge goes upward from x to y. Self-loop from the reflexivity property and all $x \sqsubseteq y$ such that can be deduced from transitivity are not represented.

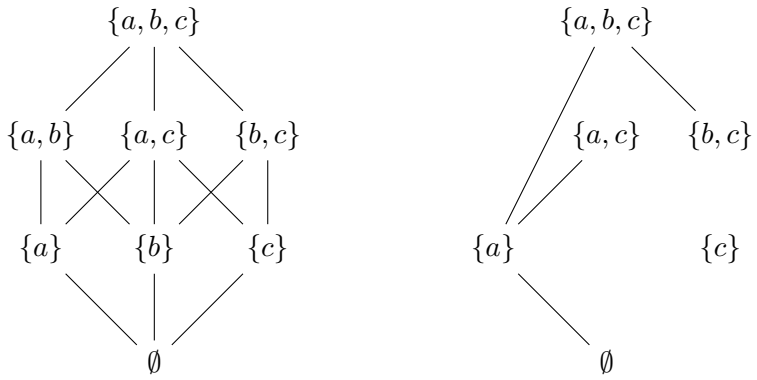


Figure 2.1 – HASSE diagrams of the partially ordered set $(\mathcal{P}(\{a, b, c\}), \subseteq)$ and of the partially ordered set $(\{\emptyset, \{a\}, \{c\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}, \sqsubseteq)$ with $\sqsubseteq = \{(\emptyset, \{a\}), (\emptyset, \{a, c\}), (\emptyset, \{a, b, c\}), (\{a\}, \{a, c\}), (\{a\}, \{a, b, c\}), (\{b, c\}, \{a, b, c\})\}$.

Let $(S, \sqsubseteq)$ be a partially ordered set.

Definition 3 (Totally ordered set) $(S, \sqsubseteq)$ is a *totally ordered set* (or a *chain*) if $\forall x, y \in S$: x and y are *comparable* (i.e. $x \sqsubseteq y$ or $y \sqsubseteq x$).

In general the order structures what we manipulate are not total. We have following properties to help us to deal with them.

Definition 4 (Upper bound)
$\forall X \subseteq S, u \in S$ is an *upper bound of* X if $\forall x \in X, x \sqsubseteq u$.

$\forall X \subseteq S, u \in S$ is the [21] *least upper bound* (or *supremum*) *of* X if u is an upper bound of X and $\forall$ upper bound u' of $X : u \sqsubseteq u'$.
We will noted it by $\sqcup X$, and we will called $\sqcup$ the *join* operator.
We will abbreviate $\sqcup\{x, y\} = x \sqcup y$.

If there exists, we called *top* the element $\top = \sqcup L$.

Dually we define,

[21] The unicity is a direct consequence of definitions and the antisymmetry property.

Definition 5 (Lower bound)
$\forall X \subseteq S, l \in S$ is a *lower bound of* X if $\forall x \in X, l \sqsubseteq x$.

$\forall X \subseteq S, l \in S$ is the *greatest lower bound* (or *infimum*) *of* X if l is an lower bound of X and $\forall$ lower bound l' of $X : l' \sqsubseteq l$.
We will noted it by $\sqcap X$, and we will called $\sqcap$ the *meet* operator.
We will abbreviate $\sqcap\{x, y\} = x \sqcap y$.

If there exists, we called *bottom* the element $\bot = \sqcap L$.

Definition 6 (Directed set) $(S, \sqsubseteq)$ is a *directed set* if $\forall x, y \in S : x \sqcup y$ exists.

Definition 7 (CPO) $(S, \sqsubseteq)$ is a *complete partially ordered set* (*CPO*) if

(i) the bottom $\bot$ element exists,

(ii) $\forall X \subseteq S : \sqcup X$ exists.

2.1.2 Lattices

Let $(L, \sqsubseteq)$ be a non-empty partially ordered set.

Definition 8 (Lattice) $(L, \sqsubseteq)$ is a *lattice* if $\forall x, y \in L : x \sqcup y$ and $x \sqcap y$ exist.

Definition 9 (Complete lattice)
$(L, \sqsubseteq)$ is a *complete lattice* if $\forall X \subseteq L : \sqcup X$ and $\sqcap X$ exist.

If L is a finite set, then every lattice $(L, \subseteq)$ is a complete lattice.

$\forall$ set $E : (\mathcal{P}(E), \subseteq)$ is a complete lattice where $\forall X \subseteq \mathcal{P}(E) : \sqcup X = \cup X$ and $\sqcap X = \cap X$.

Note that the lattice structure can be also seen as an algebraic structure, equivalent. Let $(L, \sqsubseteq)$ be a lattice.

Lemma 1 (Connecting lemma) $\forall x, y \in L :$
$(x \sqsubseteq y) \iff (x \sqcup y = x) \iff (x \sqcap y = y)$

Theorem 2 (Algebraic properties of join and meet) $\forall x, y, z \in L :$

(i) $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$ (*associative laws*)
$(x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$

(ii) $x \sqcup y = y \sqcup x$ (*commutative laws*)
$x \sqcap y = y \sqcap x$

(iii) $x \sqcup (x \sqcap y) = x$ (*absorption laws*)

$x \sqcap (x \sqcup y) = x$

(iv) $x \sqcup x = x$ (*idempotency laws*)
$x \sqcap x = x$

Theorem 3 (Lattice equivalence algebraic point of view) Let $(L, \sqcup, \sqcap)$ be a non-empty set with two binary relations that satisfy all properties of the previous theorem.
Let a binary relation $\sqsubseteq$ such that $\forall x, y \in L : (x \sqsubseteq y) \iff (x \sqcup y) = y$.
Then $(L, \sqsubseteq)$ is a lattice, such that $\sqcup$ and $\sqcap$ are the two operators defined like before.

2.1.3 Fixed points

Let $(S, \sqsubseteq)$ be a partially ordered set, let $f : S \to S$ be a function.

Definition 10 (Fixed point)
$x \in S$ is a *fixed point* of f if $f(x) = x$.

$x \in S$ is a *least fixed point* of f
if x is a fixed point and $\forall y \in S : (y \text{ is a fixed point}) \Rightarrow (x \sqsubseteq y)$.
If there exists, we note it $\mathrm{lfp}(f)$.

$x \in S$ is a *greatest fixed point* of f
if x is a fixed point and $\forall y \in S : (y \text{ is a fixed point}) \Rightarrow (y \sqsubseteq x)$.
If there exists, we note it $\mathrm{gfp}(f)$.

Let $(S, \sqsubseteq_S)$ and $(T, \sqsubseteq_T)$ be two partially ordered sets.

Definition 11 (Order-preserving function) The function $f : S \to T$ is said *order-preserving* (or *monotone*) if $\forall x, y \in S : (x \sqsubseteq_S y) \Rightarrow (f(x) \sqsubseteq_T f(y))$.

Let $(S, \sqsubseteq_S)$ and $(T, \sqsubseteq_T)$ be two CPOs.

Definition 12 (Continuous function)
The function $f : S \to T$ is said *continuous* [22] if $\forall$ directed set $X \subseteq S$:

(i) $f(X)$ is directed,

(ii) $f(\sqcup X) = \sqcup f(X)$.

This "*continuity condition can be awkward to check.*" [DP02]

Theorem 4 (Knaster–Tarski fixed point) Let $(L, \sqsubseteq)$ be a complete lattice and $f : L \to L$ an order-preserving function.
Then $\mathrm{gfp}(f) = \sqcup\{x \in L \mid x \sqsubseteq f(x)\}$ and $\mathrm{lfp}(f) = \sqcap\{x \in L \mid f(x) \sqsubseteq x\}$.

[22] The analysis notion of continuity preserves limits. The notion here of Scott continuity preserves the join of a directed set.

Theorem 5 (KLEENE fixed point) [Bou08]
Let $(L, \sqsubseteq)$ be a complete lattice with a bottom $\bot$, and $f : L \to L$ a continuous function.
Then $\text{lfp}(f) = \sqcup_{i \in \mathbb{N}}\{f^i(\bot)\}$.

With a top $\top$.
Then $\text{gfp}(f) = \sqcap_{i \in \mathbb{N}}\{f^i(\top)\}$.

2.2 Back to the notion of abstraction

> "*Abstract Interpretation, one of the most applied techniques for semantics based static analysis of software, is based on two main key-concepts: the correspondence between concrete and abstract semantics through* GALOIS *connections/insertions, and the feasibility of a fixed point computation of the abstract semantics, through the fast convergence of widening operators. The latter point is crucial to ensure the scalability of the analysis to large software systems.*"
>
> — Agostino CORTESI,
> *Widening Operators for Abstract Interpretation* [Cor08]

Let us leave the difficult problem of the widening operator, eventually for future work.

Definition 13 (GALOIS connection) A GALOIS *connection*[23] between two partially ordered sets $(A, \sqsubseteq_A)$ and $(B, \sqsubseteq_B)$ is a pair of functions $\alpha : A \to B$ and $\gamma : A \to B$ such that $\forall a \in A, b \in B : \alpha(a) \sqsubseteq_B b \iff a \sqsubseteq_A \gamma(b)$.

The concept of the GALOIS connection give us a general framework to formalized the notion of abstraction seen in the chapter 1. Let X be a concrete set, i.e. a set of values that we want analysis. $(\mathcal{P}(X), \subseteq)$ is a complete lattice.

Note $\widehat{X}$ the abstraction set of X. Suppose that $\widehat{X}$ is such that $(\widehat{X}, \sqsubseteq)$ is a partially ordered set.

Let α and γ be a GALOIS connection between $(\mathcal{P}(X), \subseteq)$ and $(\widehat{X}, \sqsubseteq)$.

$$\mathcal{P}(X) \underset{\gamma}{\overset{\alpha}{\leftrightarrows}} \widehat{X}$$

We call α the abstraction function and γ the concretization function.

Next the operations on X must be also abstracted.

Sign example Formalize the sign example seen in the subsection 1.1.4 with this framework.

$\widehat{sign} = \{\bot, +, \widehat{0}, -, \top\}$ abstracts the set of integers $\mathbb{Z}$ and form a complete lattice with the partially order characterized by the following HASSE diagram.

[23] An other possible definition sometimes used in some contexts reverses the order. Here the definition used preserve the order.

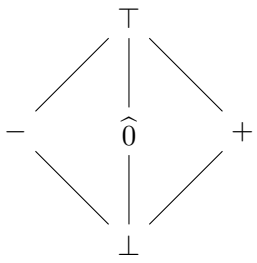


Figure 2.2 – HASSE diagram of the complete lattice of signs.

The total abstraction and concretization functions:

$$\alpha : \mathcal{P}(\mathbb{Z}) \to \widehat{sign} \qquad \gamma : \widehat{sign} \to \mathcal{P}(\mathbb{Z})$$

$$x \mapsto \begin{cases} \bot & \text{if } x = \emptyset \\ - & \text{if } x \subseteq -\mathbb{N}_* \\ \widehat{0} & \text{if } x = \{0\} \\ + & \text{if } x \subseteq \mathbb{N}_* \\ \top & \text{otherwise} \end{cases} \qquad y \mapsto \begin{cases} \emptyset & \text{if } y = \bot \\ -\mathbb{N}_* & \text{if } y = - \\ \{0\} & \text{if } y = \widehat{0} \\ \mathbb{N}_* & \text{if } y = + \\ \mathbb{Z} & \text{if } y = \top \end{cases}$$

The abstraction of the addition operation (presented only for some values):

$$\widehat{+} : \widehat{sign} \times \widehat{sign} \to \widehat{sign}$$

$$(a, b) \mapsto \begin{cases} - & \text{if } a = b = - \\ \top & \text{if } a = - \text{ and } b = + \\ \text{(The corresponding example to } \{-, 0\} \,\widehat{+}\, \{-\} = \{-\} & \\ \qquad \text{is not representable with this sign abstraction.)} & \\ \dots \end{cases}$$

If we analysis again the sign of the $-42 + 42$,
we have $\alpha(\{-42\}) = -$ and $\alpha(\{42\}) = +$.
Thus $\alpha(\{-42\}) \,\widehat{+}\, \alpha(\{42\}) = \top$
and if we come back to the concrete side: $\gamma\left(\alpha(\{-42\}) \,\widehat{+}\, \alpha(\{42\})\right) = \mathbb{Z}$.

We see that the abstraction version of the addition operation is clearly less precise that the direct concrete operation: $-42 + 42 = 0$.

Nevertheless the result is an over-approximation: $\{0\} \subset (\gamma \circ \widehat{+} \circ \alpha) = \mathbb{Z}$

Sound abstraction Let $f : X \to X$ be an operation in X. We say that the abstraction is *sound* for this operation if $\forall x \in X : \alpha(f(x)) \sqsubseteq \widehat{f}(\alpha(x))$, i.e. if the abstract operation over-approximates the concrete operation.

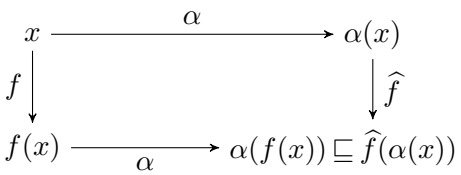


Figure 2.3 – Sound abstraction.

Future work We guess that the interest of the GALOIS connection is the good properties of the combinations of α and γ functions which allow the application of

fixed point theorems. Maybe these slides [Sut08] of the presentation *Summer School on Verification Technology, Systems & Applications — part 2* by Grégoire SUTRE will be a good first approach to learn more about that.

2.3 λ-calculus

Only few words on the *untyped λ-calculus*. It is a very simple mathematical model of computation, however equivalent in terms of calculability to the TURING machines. Consequently equivalent to all our computers and programming languages. [24] [25]

Simple description of the grammar, in the usual BACKUS–NAUR form:

Definition 14 (Untyped λ-calculus)

$$\begin{aligned} \langle \text{v} \rangle &\models \text{variable identifier} \in Var \\ \langle \text{e} \rangle &\models \langle \text{v} \rangle \mid (\lambda \langle \text{v} \rangle . \langle \text{e} \rangle) \mid (\langle \text{e} \rangle \ \langle \text{e} \rangle) \end{aligned}$$

v	variable	
$(\lambda v.e)$	abstraction	Definition of a function.
$(e\ e')$	application	Applying the function e' to the argument e.

The syntax of *Scheme* is only composed by *S-expressions*, i.e. by parenthesized lists such that the first element is the operator and the following are its arguments. This corresponds directly to the λ-calculus.

Definition 15 (S-expression)

$$\begin{aligned} \langle \text{atom} \rangle &\models \text{indivisible fundamental item} \\ \langle \text{S-expression} \rangle &\models (\langle \text{S-expression} \rangle \ . \ \langle \text{S-expression} \rangle) \end{aligned}$$

[24] In fact our computers are limited by time and memory, in contrary to a TURING machine, that is a mathematical "object". However ideal versions of computers are completely equivalent to a TURING machine.
Programming languages are also mathematical "objects", and theirs implementations have the same limitations that computers.

[25] As all known calculability models are equivalent each others (with the exception of models that are too simple), the CHURCH–TURING thesis proclaims that this notion of calculability corresponds exactly to the intuitive notion of calculability.

Chapter 3

CESK machine

This chapter outlines the machine concept such as developed by Van Horn and Might in the paper *Abstracting Abstract Machines* [VM10]. They successively transform more limited model to finally arrive to the formalism of the time-stamped CESK* machine. [26]

The result was proved sound, i.e. "*that the states reachable by the concrete transition function are also reachable by the abstract transition function, so that we can use only the abstract transition function to prove properties about the concrete transition function.*" [Sti14]

The transformations done had the object of reducing infinities to make the problem decidable as explained in chapter 2.

This machine is capable to perform both concrete and abstract interpretation. From a input program the principle is to compute a set of states by an evaluation function.

3.1 Concrete semantics

The concrete semantics contains these four elements (that formalize notions from the chapter 1):

- *State space*: set of all states of the machine,
- *Injection function*: function that defines how to inject the initial expression (the program to be executed or analyzed) to an initial state,
- *Transition function*: function that defines how to move from one state to another,
- *Evaluation function*: function that defines the set of states reachable from another.

The state space of this *CESK* machine is composed of four elements:

- *Control*: represents the expression that is currently evaluated by the machine,

[26] The paper make a difference between a CESK machine and the final version of the CESK* machine. We will note simply CESK for this final version, like in [Sti14].

- *Environment*: represents the environment in which this expression is evaluated and binds variable names to (possibly abstracted) memory addresses,
- *Store*: represents the memory of the machine, and binds memory addresses to actual values (concrete or abstract depending on the configuration of the machine),
- [K]*Continuation* pointer: represents what has to be evaluated next by the machine.

In [Sti14] Quentin Stiévenart exposes this machine and formalize the semantic of the untyped *lambda*-calculus presented here in the section 2.3. As already said this is very close of the *Scheme* programming language.

3.2 Abstract semantics

The abstract semantics is also formalized in [Sti14]. It is almost a direct transformation of the concrete semantics.

3.3 Optimizations

The master thesis [Sti14] presents some useful optimizations, depending of the context, like abstract counting, abstract garbage collection, state subsumption.

The paper *Optimizing Abstract Abstract Machines* [Joh+13] is a more complicated matter that will be read again.

3.4 Beyond this simple evocation

It is a crucial part of subject that we will have to work more seriously in the coming year. Maybe experiments with very simple languages may be useful to a full understanding of this. Or maybe to implement from scratch a CESK machine.

And of course experiment with *Scala-AM* [Sti17]. The implementation that will be parallelized.

Chapter 4

Parallelization

Parallelization is the true goal for the future master thesis. In this preparatory work, after laying the necessary foundations of the domain of abstract interpretation, we only discuss some general notions on the subject of parallelization.

The enticing idea of parallelization is to do several things at the same time to have a performance gain, contrary to the usual sequential programs, which execute the instructions one after the other in a specific order.

Two problems arise immediately. First, the fundamental necessity to identify independent parts. Because dependent parts cannot be executed at the same time. Second, it is to achieve correctly the interaction between the execution of the dependent parts.

Let us first evoke briefly this second problem. It is well known that it is very hard to develop correct concurrent programs. The indeterminism of the execution order can lead to a bug to several results. As the execution can take different paths, debugging is difficult. It is probably less known that the number of paths can be really astonishing, even for small programs.

Robert C. Martin in *Clean Code: A Handbook of Agile Software Craftsmanship* [Mar09] gives this surprising example:

```
public class X {
    private int lastIdUsed;
    public int getNextId() {
        return ++lastIdUsed;
    }
}
```

If one instance of this very simple *Java* class is shared between two threads, then the number of possible execution paths is 12,870. With `long` instead `int` the number of paths becomes 2,704,156. Most of them produce the correct result, but some of them do not. In this context, finding bugs is as like finding a needle in a haystack.

Let us return to the first problem. A program is always intrinsically composed of a certain proportion of elements depending on each other.

Let t_s be the sequential execution time of a program, sometimes called *work*. Let t_p be the execution time of the same program but on n processors, sometimes called *span*.

The *speedup factor* $S(n) = {}^{t_s}/_{t_p}$ measures the relative acceleration between the sequential execution and the parallel execution.

The *linear speedup principle* say that the maximum speedup factor with n processors is n. But in practice is less. Because each program contains parts that cannot run in the same time.

Law 1 (AMDAHL's law) Let α the fraction of necessarily sequential computations of a parallel program. Thus $(1 - \alpha)$ is the fraction of parallelizable computations. $S(n) = \frac{t_s}{\alpha t_s + (1-\alpha)\frac{t_s}{n}} = \frac{n}{1+(n-1)\alpha}$

If we get the upper bound of this expression (as if we had an infinite number of processors) we obtains: $\lim_{n\to\infty} S(n) = \frac{1}{\alpha}$.

That prove that the expected speedup is bounded by the not-parallelizable part.

With 5% of the computations necessary sequential, the *potential* maximum speedup factor is 20. Whatever the number of available processors.

4.1 Actor model of concurrency

The actor model is a conceptual model of concurrency. Instead of manipulating threads and synchronisation to run in parallel and share resources, the idea is to delegate the parallel behaviour to entities named *actor*, something like that objects in the object paradigm. But only as objects which would completely isolated from each other, with the exception to the possibility to send and receive messages.

These message can contains the references of the actors, which it make possible to contact actors initially unknown.

Each actor send and receive message sequentially. However all actors potentially can run in the same time.

Furthermore each actor can create new actors.

The design of the actor model facilitates the distributed programming, since this particular form of concurrency works without share resources.

***Akka* framework** *Akka* [coma] is a concurrency framework available for the *Scala* and *Java* languages. It gives several concurrency models, with a special attention to the actor model. For these two reasons it will be the framework used in future work.

It is a Free Software [FSF], like the standard implementation of *Scala*.

4.2 Curiosity for the computation graph model

In the first MOOC [Sar17] was invoked the *computation graph* (*CG*) model of concurrency. May we will learn more about that, with a special interest for what seems to be a link with the notion of partially ordered sets developed in chapter 2.

4.3 Beyond this simple evocation, again

This subject of concurrency is also a crucial part of the future work. However it is the essential specific subject the most neglected in this document.

Chapter 5

Future work

"*To be continued...*"

Much work remains to be done. Here a brief list of elements that which have been identified.

Explore more properties of Galois connections and see if they help to parallelize.

Read some other articles from Might and Van Horn, and specially read again the important article on the CESK machine [VM10]. It is a very important matter to develop my master thesis.

This other very important paper *A parallel abstract interpreter for JavaScript* [DKH15] also must have read again. It was difficult to understand it before have a good understanding of the concept of the abstract interpretation.

Maybe also read some historical papers from Cousot and Cousot. More out of curiosity.

Deeply experiment with *Scala-AM* [Sti17]. First to well understand it. Then to make the expected work of parallelization.

I just started the three MOOCs of the *Parallel, Concurrent, and Distributed Programming in Java Specialization* [Sar17] by Vivek Sarkar from the Rice University and I will try to follow all of them in parallel to finished them before the start of the academic year. It seems to me that is a good way in order to have a good overview on the large problematic of the parallelization and distribution domains.

Acquiring knowledge of concurrency programming will be a big part of the work to come.

During a previous MOOC on *Parallel programming* [KP17] I discovered the difficulty of making precise benchmarks on the JVM [GBE07] [GEB08].

The library *ScalaMeter* [comc] was used for that. I plan to implement its utilisation in *Scala-AM*.

During the first semester in the next academic year I plan to follow one or two of these courses by Wolfgang De Meuter from VUB: *Functional Programming* [De 17a] and *Higher-Order Programming* [De 17b]. This will be an opportunity for me to deepen my understanding of the functional paradigm.

...

References

Bibliography

[Abs] AbsInt. **Astrée Runtime Error Analyzer**. URL: https://www.absint.com/astree/ (cit. on pp. i, ii).

[ASS96] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. **Structure & Interpretation of Computer Programs 2e**. second edition. Cambridge, Mass.: MIT Press, 1996. 684 pp. ISBN: 978-0-262-51087-5. URL: https://mitpress.mit.edu/sicp/ (cit. on pp. 9, 12, 32).

[Bou08] Olivier Bouissou. **Analyse statique par interprétation abstraite de systèmes hybrides**. PhD thesis. École Polytechnique X, Sept. 23, 2008. URL: https://pastel.archives-ouvertes.fr/pastel-00004412/document (cit. on pp. i, ii, 1, 17).

[CC77] 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*. POPL '77. New York, NY, USA: ACM, 1977, pp. 238–252. DOI: 10.1145/512950.512973. URL: http://doi.acm.org/10.1145/512950.512973 (cit. on pp. i, ii).

[CC79] 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*. POPL '79. New York, NY, USA: ACM, 1979, pp. 269–282. DOI: 10.1145/567752.567778. URL: http://doi.acm.org/10.1145/567752.567778 (cit. on pp. i, ii).

[coma] Apache License community. **Akka**. URL: http://akka.io/ (cit. on pp. 11, 23).

[comb] BSD License community. **Scala**. The Scala Programming Language. URL: https://www.scala-lang.org/ (cit. on p. 11).

[comc] BSD License community. **ScalaMeter**. URL: https://scalameter.github.io/ (cit. on p. 25).

[comd] LGPL License community. **Racket**. URL: http://www.racket-lang.org/ (cit. on pp. 10, 12).

[com90] 1178 WG COMMUNITY. **IEEE SA - 1178-1990 - IEEE Standard for the Scheme Programming Language**. 1990. URL: https://standards.ieee.org/findstds/standard/1178-1990.html (cit. on p. 10).

[Cor08] Agostino CORTESI. **Widening Operators for Abstract Interpretation**. In: *2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*. 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods. Nov. 2008, pp. 31–40. DOI: 10.1109/SEFM.2008.20 (cit. on p. 17).

[Cou00] Patrick COUSOT. **Interprétation abstraite**. In: *Technique et science informatique* 19.1 (Jan. 2000), pp. 155–164. URL: http://www.di.ens.fr/~cousot/COUSOTpapers/TSI00.shtml (cit. on p. 8).

[Cou08] Patrick COUSOT. **La vérification des programmes par interprétation abstraite**. Collège de France, Feb. 22, 208. URL: http://www.college-de-france.fr/site/gerard-berry/seminar-2008-02-22-11h30.htm (cit. on pp. 5, 6).

[Cou78] Patrick COUSOT. **Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes**. thesis. Institut National Polytechnique de Grenoble - INPG ; Université Joseph-Fourier - Grenoble I, Mar. 21, 1978. URL: https://tel.archives-ouvertes.fr/tel-00288657/document (cit. on pp. i, ii).

[De 16] Jonas DE BLESER. **Static Taint Analysis of Event-driven Programs: An Abstracting Abstract Machines Approach**. Master's thesis. Vrije Universiteit Brussel, June 2016 (cit. on p. 10).

[De 17a] Wolfgang DE MEUTER. **Functional Programming**. Course from Vrije Universiteit Brussel. 2017. URL: http://www.vub.ac.be/en/study/fiches/54625/functional-programming (cit. on p. 26).

[De 17b] Wolfgang DE MEUTER. **Higher-Order Programming**. Course from Vrije Universiteit Brussel. 2017. URL: http://www.vub.ac.be/en/study/fiches/55234/higher-order-programming (cit. on p. 26).

[DKH15] Kyle DEWEY, Vineeth KASHYAP, and Ben HARDEKOPF. **A parallel abstract interpreter for JavaScript**. In: *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). Feb. 7, 2015, pp. 34–45. DOI: 10.1109/CGO.2015.7054185 (cit. on p. 25).

[DP02] Brian A. DAVEY and Hilary A. PRIESTLEY. **Introduction to Lattices and Order**. 2nd ed. Cambridge, UK ; New York, NY: Cambridge University Press, 2002. 312 pp. ISBN: 978-0-521-78451-1 (cit. on pp. 13, 16).

[Fel01] Matthias FELLEISEN. **How to Design Programs – An Introduction to Programming & Computing**. Cambridge, Mass: MIT Press, 2001. 724 pp. ISBN: 978-0-262-06218-3. URL: http://www.htdp.org/ (cit. on p. 32).

[FSF] FSF. **What is free software?** URL: https://www.gnu.org/philosophy/free-sw.en.html (cit. on p. 23).

[GBE07] Andy Georges, Dries Buytaert, and Lieven Eeckhout. **Statistically Rigorous Java Performance Evaluation**. In: *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*. OOPSLA '07. New York, NY, USA: ACM, 2007, pp. 57–76. ISBN: 978-1-59593-786-5. DOI: 10.1145/1297027.1297033. URL: http://doi.acm.org/10.1145/1297027.1297033 (cit. on p. 25).

[GEB08] Andy Georges, Lieven Eeckhout, and Dries Buytaert. **Java Performance Evaluation Through Rigorous Replay Compilation**. In: *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*. OOPSLA '08. New York, NY, USA: ACM, 2008, pp. 367–384. ISBN: 978-1-60558-215-3. DOI: 10.1145/1449764.1449794. URL: http://doi.acm.org/10.1145/1449764.1449794 (cit. on p. 25).

[Joh+13] J. Ian Johnson, Nicholas Labich, Matthew Might, and David Van Horn. **Optimizing Abstract Abstract Machines**. In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP '13. New York, NY, USA: ACM, 2013, pp. 443–454. ISBN: 978-1-4503-2326-0. DOI: 10.1145/2500365.2500604. URL: http://doi.acm.org/10.1145/2500365.2500604 (cit. on p. 21).

[KCR98] Richard Kelsey, William Clinger, and Jonathan Rees. **Revised5 Report on the Algorithmic Language Scheme**. Feb. 20, 1998. URL: http://www.schemers.org/Documents/Standards/R5RS/ (cit. on pp. 10, 11, 32).

[KP17] Viktor Kuncak and Aleksandar Prokopec. **Parallel programming**. MOOC of Coursera from École Polytechnique Fédérale de Lausanne. July 2017. URL: https://www.coursera.org/learn/parprog1 (cit. on p. 25).

[Mar09] Robert C. Martin. **Clean Code: A Handbook of Agile Software Craftsmanship**. Pearson Education, 2009 (cit. on p. 22).

[Mig11] Matthew Might. **Tutorial: Small-step CFA**. NII Shonan Meeting on Higher-Order Program Analysis. Shonan Village, Japan, Sept. 23, 2011. URL: http://matt.might.net/papers/might2011cfa-talk.pdf (cit. on pp. 5, 7).

[Mig12] Matthew Might. **What is static analysis?** Matt Might. 2012. URL: http://matt.might.net/articles/intro-static-analysis/ (cit. on pp. 7, 12).

[Mig14] Matthew Might. **What is static snalysis?** Lambda Lounge, Salt Lake City, Utah, Aug. 8, 2014. URL: https://www.youtube.com/watch?v=POvX4hYIoxg (cit. on p. 12).

[Ode16a] Martin Odersky. **Functional Program Design in Scala**. MOOC of Coursera from École Polytechnique Fédérale de Lausanne. 2016. URL: https://www.coursera.org/learn/progfun2 (cit. on p. 12).

[Ode16b] Martin Odersky. **Functional Programming Principles in Scala**. MOOC of Coursera from École Polytechnique Fédérale de Lausanne. 2016. URL: https://www.coursera.org/learn/progfun1 (cit. on p. 12).

[Pro17] Aleksandar Prokopec. **Learning Concurrent Programming in Scala**. Google-Books-ID: D1QoDwAAQBAJ. Packt Publishing Ltd, Feb. 22, 2017. 426 pp. ISBN: 978-1-78646-214-5. URL: https://books.google.be/books?id=D1QoDwAAQBAJ (cit. on p. 11).

[Ric53] Henry Gordon Rice. **Classes of Recursively Enumerable Sets and Their Decision Problems**. In: *Transactions of the American Mathematical Society* 74.2 (1953), pp. 358–366. ISSN: 0002-9947. DOI: 10.2307/1990888. URL: http://www.jstor.org/stable/1990888 (cit. on p. 2).

[Sar17] Vivek Sarkar. **Parallel, Concurrent, and Distributed Programming in Java Specialization**. MOOCs of Coursera from the Rice University. 2017. URL: https://www.coursera.org/specializations/pcdp (cit. on pp. 23, 25).

[Sip12] Michael Sipser. **Introduction to the Theory of Computation**. Google-Books-ID: 1aMKAAAAQBAJ. Cengage Learning, June 27, 2012. 482 pp. ISBN: 978-1-285-40106-5. URL: https://books.google.be/books?id=1aMKAAAAQBAJ (cit. on p. 3).

[SS98] Gerald Jay Sussman and Guy Lewis Steele Jr. **The First Report on Scheme Revisited**. In: *Higher-Order and Symbolic Computation* 11.4 (Dec. 1, 1998), pp. 399–404. ISSN: 1388-3690, 1573-0557. DOI: 10.1023/A:1010079421970. URL: https://link.springer.com/article/10.1023/A:1010079421970 (cit. on p. 11).

[Sti+16] Quentin Stiévenart, Maarten Vandercammen, Wolfgang De Meuter, and Coen De Roover. **Scala-AM: A Modular Static Analysis Framework**. In: *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM). Oct. 2, 2016, pp. 85–90. DOI: 10.1109/SCAM.2016.14 (cit. on p. 12).

[Sti14] Quentin Stiévenart. **Static Analysis of Concurrency Constructs in Higher-Order Programs**. Master's thesis. Université Libre de Bruxelles, 2014. URL: https://pdfs.semanticscholar.org/1359/031f84ae96b4e3a5bc1cc3faf3a3ed5dc8bc.pdf (cit. on pp. 10, 20, 21).

[Sti17] Quentin Stievenart. **scala-am: (Abstract) Abstract Machine Experiments using Scala**. original-date: 2014-12-17T14:14:02Z. June 8, 2017. URL: https://github.com/acieroid/scala-am (cit. on pp. 11, 12, 21, 25).

[Sut08] Grégoire Sutre. **Summer School on Verification Technology, Systems & Applications — part 2**. Saarbrücken, Sept. 2008. URL: http://resources.mpi-inf.mpg.de/departments/rg1/conferences/vtsa08/slides/sutre2.pdf (cit. on p. 19).

[VM10] David Van Horn and Matthew Might. **Abstracting Abstract Machines**. In: *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. ICFP '10. New York, NY, USA: ACM, 2010, pp. 51–62. ISBN: 978-1-60558-794-3. DOI: 10.1145/1863543.1863553. URL: http://doi.acm.org/10.1145/1863543.1863553 (cit. on pp. 20, 25).

Abbreviations [27]

AAC Abstracting Abstract Control

AAM Abstracting Abstract Machine

AM Abstract Machine

ASM Abstract State Machine

BNF Backus–Naur form

CESK Control, Environment, Store, [K]Continuation

CFA Control Flow Analysis

CFG Control Flow Graph

CG Computation Graph

CPO Complete Partial Order

CPS Continuation-Passing Style

DAG Directed Acyclic Graph

HtDP *How to Design Programs* [Fel01]

IEEE Institute of Electrical and Electronics Engineers

gfp Greatest Fixed Point

JVM Java Virtual Machine

lfp Least Fixed Point

MOOC Massive Open Online Course

poset Partially Ordered Set

R5RS Revised^5 Report on the Algorithmic Language Scheme [KCR98]

RnRS Revised^n Report on the Algorithmic Language Scheme

redex Reducible Expression

RTE Running Time Error

SICP *Structure and Interpretation of Computer Programs* [ASS96]

SRFI Scheme Requests for Implementation, https://srfi.schemers.org/

STS State Transition System

ULB Université Libre de Bruxelles

VUB Vrije Universiteit Brussel

[27] Abbreviations mentioned in this document or used in the literature.

Shortlist of common Symbols

$<$, $>$	Strict inequalities.
$\leq$, $\geq$	Inequalities.
$\subset$, $\supset$	Strict inclusions.
$\subseteq$, $\supseteq$	Inclusions.
$\mathbb{N}$	the set of *natural numbers* $\{0, 1, 2, 3, \ldots\}$.
$\mathbb{N}_*$	$\{1, 2, 3, \ldots\}$, natural numbers without 0.
$\mathcal{P}(X)$	the *powerset* of X, i.e. the set of all subsets of X.
$\mathbb{Z}$	the set of *integer numbers* $\{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$.
$f : X \to Y$	Function f (in the general sense, not necessarily total) from the *domain* X to the *codomain* Y.
$\mathrm{dom}(f)$	*Domain of definition* of a function f.
$\mathrm{img}(f)$	*Image* of a function f, i.e. $\mathrm{img}(f) = f(\mathrm{dom}(f))$.
$f^i(x)$	ith iterate of the f function, i.e. $f^0(x) = x$ and $f^{i+1}(x) = f(f^i(x))$.

List of Figures

List of Definitions

List of Theorems

Index

"*Beware of bugs in the above code; I have only proved it correct, not tried it.*"

— Donald E. Knuth,
Notes on the van Emde Boas *construction of priority deques: An instructive use of recursion*, March 1977