Automated Approximate Recurrence Solving applied to Static Analysis of Energy Consumption

Louis Rustenholz, supervised by Manuel Hermenegildo,
Pedro López-García and José F. Morales,
CLIP Lab, IMDEA Software Institute, Madrid

13 August 2022

The general context

With IT's share of global carbon footprint around 2.5% in 2013 and 5% in 2022, it is one of the sectors with the fastest growing energy and carbon usage. While IT solutions have the potential to avoid emissions in other sectors, developing new analysis and optimisation tools of IT's carbon emissions is a worthwhile pursuit, as such tools are currently in their infancy.

While the energy consumption of *programs* represents around a third of the energy use of the sector, it is often overlooked by carbon audit experts, mostly by lack of tools and data.

The scientific field of program analysis has some prior experience in the study of *resource consumption* of programs. Note that it is a rich subject from a theoretical viewpoint: we are interested in the study of *non-functional properties.*

Historically, the resource that has received the most attention is *time.* Rather mature worst-case execution time (WCET) tools exist, such as AbsInt's `aiT`. This type of static analysis tools however suffer from a major drawback: *developers must add assertions giving constant bounds on the number of executions of each loop.* In other words, very little control-flow analysis (CFA) is performed.

CiaoPP, a program processing tool designed for logic programs, contains analysis modules for *parametric* resource consumption. It is possible to study time, memory, energy, or any user-defined resource. Moreover, CiaoPP proposes to completely automate CFA by reducing the resource consumption problem to the resolution of recurrence equations.

Many of CiaoPP's analyses are based on the theory of *abstract interpretation*, started by Patrick Cousot in the 70s, and which gave birth to many static analyser usually focused on more *functional* properties.

The research problem

CiaoPP's cost analysis idea may be framed as the following: to a large extent, systems of recurrence equations may be viewed as programs stripped from information irrelevant for resource consumption analysis.

A central question studied during this internship is thus to improve recurrence equation solvers to make them more powerful in the context of resource consumption analysis.

Available solvers (`Mathematica`, `PURRS`, etc.) mostly work by implementing known methods for specific classes of recurrence equations. Given an equation to solve, they check whether it appears in a dictionary of known shapes. If it does, the exact solution is outputted, otherwise nothing is done. This leaves the road open for at least two types of original contributions: *approximate solving* – obtain bounds on the solution rather than the exact solution – and *domain-specific solving* – solve equations common in cost analysis but uncommon otherwise.

Those approaches are not entirely new and have been explored in other theses. However, our lattice-theoretical ideas are novel and could bring new methods to the field. Moreover, in the imperative energy analysis context, many max operators appear in the equations, and we believe that our experience with order theory and tropical geometry may provide inspiration.

Improving recurrence solving has far-reaching consequences on cost analysis. Indeed, precision and generality of the recurrence solver are currently the main bottlenecks in CiaoPP's cost analysis. Moreover, expanding the class of solvable equations also allows the other modules to extract more challenging but precise equations from the programs.

More generally, during this internship, we reviewed the entire pipeline for energy consumption analysis of imperative programs, identified shortcomings, and proposed directions of improvement.

Your contribution

General approximate recurrence resolution

- An order-theoretical framework for *monotone* recurrence equations is proposed.
- Pre-/post-fixpoints are presented as *easily checkable bounds* on solutions. This provides a new manual proof technique. To automate it, a custom guess-and-check algorithm is sketched, and divide-and-conquer ideas are studied.
- A generic algorithm based on abstract interpretation is proposed.
  A few abstract domains are presented and tested on paper.

Implementation

- Complex numbers and second/third order linear recurrence solving are added to CiaoPP.
- An analysis+rewriting fixpoint algorithm is added to remove irrelevant variables.
- Small updates and bugfixes.

Possible directions for future work at multiple stages of the pipeline are also proposed.

Arguments supporting its validity

Implementations are tested on various examples. New algorithm ideas are proven sound by general order-theoretical theorems, and tested on paper.

Monotonicity of the recurrence equations is a strong requirement, but is an assumption already very present in CiaoPP's cost analysis. A sound but imprecise way to monotonise equations is proposed to add generality.

Summary and future work

On the implementation side, the author learned the Ciao language, got familiar with the CiaoPP codebase, and implemented a few new functionalities to the recurrence solver.

On the theoretical side, a new order-theoretical viewpoint on recurrence solving is introduced, as well as algorithm ideas stemming from it. It is quite general, and may be applied to other use cases. One of the next steps is to implement those techniques and test them on realistic benchmarks. Some lines of research for future work are listed in the following bullet points. Multiple other ideas are mentioned in the body of the report.

- Generating benchmarks displaying "the recurrence structure of real-world programs" is a challenge and an interesting scientific question in itself. Debugging tools have been developed during this internship, but more work is required to ensure that precise equations are generated from imperative programs (ranking function inference, better treatment of conditionals, etc.).
- Rewriting techniques of recurrence equations and their connection with rewriting of programs, having in mind an abstract lens on program transformation and optimisation [1].
- Generation of energy models, applications of program analysis to hardware description languages, and other low-level aspects such as cache analysis.
- Finally, we would like to apply those ideas. Contacts have been taken with IT carbon audit professionals to get to know industrial challenges. It would also be interesting to apply formal methods to other aspects of IT, e.g. to detect and monitor bad practices related to energy consumption.

Acknowledgements

First and foremost, I would like to deeply thank my advisors, Manuel Hermenegildo, Pedro Lopez Garcia and José F. Morales, for their support, their advice and for the great opportunity to work on such a motivating subject. More generally, I want to thank all of IMDEA Software's researchers, students and support staff for their warm welcome to Madrid and for making my stay so enjoyable and enriching. The many discussions and activities organised taught me a lot, and I am glad to start my – in-person – research life here.

I would also like to thank all of my abstract interpretation professors for what they shared during my semester at MPRI, with a particular thought for Antoine Miné who advised me to contact Manuel Hermenegildo for this internship and allowed me to prepare for it as part of the course's final exam, as well as for Jérôme Feret for giving his thoughts on static analysis applied to energy consumption. Additionally, I would like to thank Patrick Cousot for visiting us at IMDEA and for his ideas on my topic.

Finally, my thanks go to my MPRI jury for the time dedicated to this report.

Abstract

To a large extent, systems of recurrence equations may be viewed as programs stripped from information irrelevant for resource consumption analysis. However, most available recurrence solvers are based on lookups in dictionaries containing expert knowledge, and lack the generality required for program analysis.

In this work, we introduce new techniques for approximate recurrence solving, based on abstract interpretation and order theory.

CiaoPP, the program processor framework of the Ciao language, already contains multiple functionalities for program analysis and transformation. It includes a cost analysis module, which extracts relevant recurrence equations from programs and attempts to solve them. In recent work, using Ciao as an intermediate language, this has been applied to energy consumption analysis of programs written in common imperative languages.

In this internship report, we study those analysis pipelines, discuss their shortcomings, and propose possible improvements.

A focus is put on improving the recurrence solving module. Before implementing our new order-theoretical algorithms, we add some functionalities to the recurrence solving module, as an implementation exercise. Adding complex numbers to CiaoPP allows us to expend the dictionary of known recurrence solutions. Moreover, an analyse+rewrite pass is implemented to remove irrelevant variables from equations.

Contents

1 Background

1.1 Scientific background

Cost Analysis, Recurrence Equations, Undecidability. Complexity analysis – the study of the resource consumption of programs in terms of time, memory, but also energy and other resources – is a common concern in computer science, and multiple reasoning tools are available to *manually* study *asymptotic* complexities. However, it is much harder to *automate* the discovery of the *exact* cost of programs, necessary for tasks such as automated optimisation. In fact, *this problem is undecidable*, since the halting problem reduces to it. In a 1975 prototype applied to `Lisp` programs, Wegbreit underlined how the cost analysis problem may be reduced to automated recurrence solving [2]. Unsurprisingly, *the recurrence solving problem is also non-computable in general*, and undecidable equations may be created even with only basic arithmetic expressions [3].

Nevertheless, the field of formal methods has a way to answer seemingly impossible questions: we have to aim for *sound*, *approximate* answers. For a decision problem, rather than a Yes/No answer, output Yes/No/*I don't know*, and aim to minimise the size of "I don't know" space. Similarly, for cost analysis, rather than returning an exact value, give upper and lower bounds, and aim for them to be "reasonably close".

Abstract Interpretation, Logic Programming, Ciao. The theory of *abstract interpretation*, started by Cousot and Cousot in the 70s [4], provides a systematic way to escape the consequences of Rice's theorem, by formally constructing abstract semantics of programs. Executing such programs in the *abstract* domain provides information about their *concrete* semantics, whose exact properties are undecidable. This theory is based on order-theory: semantic domains appear as lattices, and semantics of programs are defined as fixed point of a well-chosen operator. It lead to multiple analysis tools, such as the now industrial `Astrée` [5] or the modular multilingual `MOPSA` [6].

As abstract interpretation became mature, it was applied to new programming paradigms, including *logic programming* [7, 8]. This declarative paradigm is exemplified by `Prolog` [9], in which programs are written using first-order logical formulas called *Horn clauses*, and executed via an automated proof technique such as *SLD-resolution*. It allows for non-determinism and a form of reversibility: programs are defined as relations rather than functions. In the 90s, a `Prolog` dialect called `Ciao` appeared, with analysis, verification, debugging and optimisation at its core, thanks to its program processor `CiaoPP` enabled by abstract interpretation. Thanks to its modular design, CiaoPP has gained many functionalities since then, including new abstract domains, and an improved fixpoint algorithm [10].

CiaoPP's cost analysis, HCIR, Application. CiaoPP has an important line of work on automated cost analysis and its applications. The foundational technique presented in 1990 [11] and implemented in `Caslog` [12] was initially motivated by automated parallelisation. In the next few years, it received multiple improvements, including the possibility to compute lower bounds of the complexity rather than only upper bounds [13]. In the same paper, it was noticed that complexity analysis often encounters domain-specific properties of recurrence equations, such as non-determinism. In 2007, the cost analysis module was made parametric with respect to the type of *bound* (upper or lower), *resource* (time, memory, energy, user-defined, etc.) but also *cost* (worst-case, probabilistic [14], accumulated [15], etc.). In 2013, the module gained in precision by being completely reframed as an abstract domain [16], and new techniques for verification of cost assertions via automated comparison of functions were introduced in 2018 [17].

Thanks to Horn clauses' simplicity and expressivity, they may be used as an intermediate language (HCIR) for the analysis of programs written in other paradigms. This idea was used in related tools such as the LLVM analyser `SeaHorn` [18]. More generally, the use of Horn clauses in verification has grown in the last decades and they have been applied to multiple fields, including cryptographic protocols with `ProVerif` [19].

In the recent years, CiaoPP has thus been applied to the energy analysis of imperative programs, such as mobile applications in `Java` [20] and embedded software in `C` (at the C, LLVM and ISA levels) [21, 22, 23]. Applications to *gas* consumption of blockchain have also been proposed [24].

Energy, Hardware. Hardware researchers and engineers, unlike software scientists, have had a long interest in energy consumption, as it governs heat emission, and has thus an important impact on hardware's scale and speed. An interesting course on energy analysis at low-levels can be found at [25]. Physical parameters are identified, but bridging the gap between them and code execution is a hard problem.

Pure testing is possible, but misses much information about possible execution traces. To use static analysers, energy models are required, and a suitable level of abstraction must be chosen.

At a level closest to bare metal, the amount of gate switching has been identified as a great approximation of dynamic energy consumption [26]. At this level, simulation is possible, using VHDL descriptions if available [26, 27].

To make models closer to the code, one idea might be to assign an energy cost – or an interval – to each low-level instruction – or pair of instruction, to account for some switching cost. An overview of what is done in CiaoPP is presented in Umer Liqat's PhD thesis [28]. This is reasonable [29], but instruction cost is quite data-dependent in practice [30]: the cost of a subtraction may double when the result is negative, because of two's complement representation.

Moreover, modern processors have many functionalities, such as cache memories, which have an enormous impact on cost. Some abstract interpretation techniques are able to deal with it [31], which is used by `aiT` [32] in the case of time, but many difficulties arise, as cache models are rarely available.

Note that `aiT`, a tool used for WCET analysis, requires the user to input a constant bound on the number of executions of each loop. The missing piece is more control-flow analysis, which is what we propose to do in our work, by reduction to recurrence solving.

Recurrence Equations. Among computer algebra systems (CAS), state of the art recurrence solvers which may be used with CiaoPP include `Mathematica` [33] and `PURRS` [34]. The latter contains a few specialised techniques for complexity analysis, e.g. to deal with non-deterministic divide-and-conquer equations. More generally, the book $A = B$ by Marko Petkovšek et al. [35] presents many state of the art techniques for the exact resolution of known classes of equations which still have to be implemented in CiaoPP. Focus is given to holonomic sequences, of which hypergeometric sequences are a particular case.

Solutions to classes of equations containing max operators are presented in Maximiliano Klemen's PhD thesis [36], as well as preliminary work on guess-and-check methods for more general recurrence solving using Lasso regression and SMT solvers.

The COSTA group also worked on recurrence solving specialised for cost analysis, dealing with what they call *cost relations*: non-deterministic recurrence equations with various types of guards. Their solver `PUBS` [37] implements some specialised methods. For example, it infers ranking functions and invariants to bound cost in evaluation trees, and performs partial evaluation to simplify equations. Other techniques are presented in Antonio Flores' thesis [38].

Interestingly, previous work on abstract domains of sequences exist. The only example we know of is the domain of arithmetico-geometric sequences by Jérôme Feret [39], initially applied to numerical filters in avionic software.

Cost analysis of Functional Programs. Related work has been done for functional programs, such as Wegbreit's `METRIC` – applied to Lisp, which extract recurrence equations to be solved by CAS, but isn't able to deal with mutual recursion – or David Le Métayer's `ACE` [40], which focuses on asymptotic complexity. Recent work often leverages type systems [41, 42], but have trouble with either higher-order or recursion and are not always applicable to programs written in languages not specialised for cost analysis.

1.2 Some orders of magnitude

To curb and control an industrial sector's carbon footprint, it is necessary to have in mind relevant orders of magnitude, in order to focus on significant aspects. To this end, we include a small number of figures extracted from reports published in the last 5 years by academics, associations and government officials. Of course this is only a first glimpse, as robust data collection would require more than a few paragraphs.

IT's share of global carbon footprint has risen from around 2.5% in 2013 to 5% in 2022. This footprint can be entirely attributed to energy use, either in the form of electricity or directly as petrol to extract required materials. At a macroscopic level, this footprint can be split between a low number of industrial equipment whose operation requires lots of electricity, and a very high number of consumer equipment whose individual operation requires less electricity. Globally, equipment production represents $\sim$45% of IT's energy usage, and restraint in this sector is a major carbon savings opportunity.

In our tool, we focus more directly on energy used for the operation of IT equipment, which represents $\sim$55% of IT's consumption, although our resource analysis techniques may also be used to temper the amount of equipment used. We check that energy usage attributed to data processing is significant with regard to idle consumption and cooling by going through a few case studies on datacenters, which tend to get more optimisation attention than personal equipment, although they don't consume more globally.

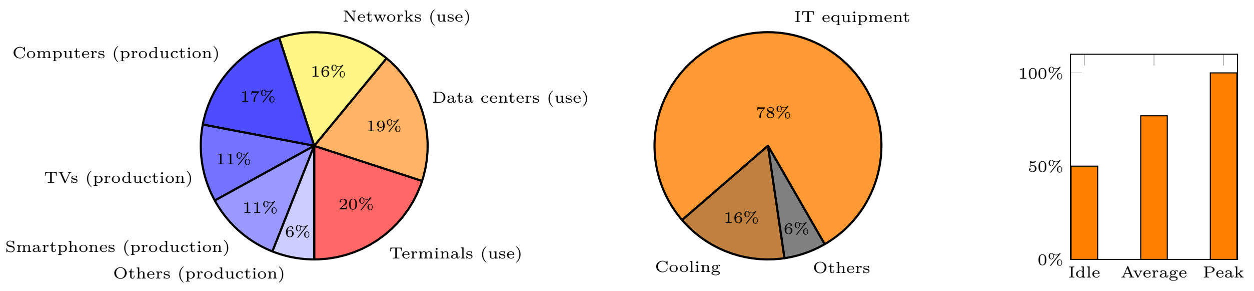

Figure 1: Energy consumption of IT's subsectors [43, 44] (left). Energy consumption of a datacenter with reasonably efficient cooling architecture (PUE 1.38) [45, 46] (center). Approximate idle, average and peak power usage of a server in case study [47] as a fraction of peak power (right).

2 CiaoPP's energy consumption analysis pipeline

In this section, we describe the current state of CiaoPP's energy analysis pipeline, and propose some improvement ideas. Here, the goal of energy consumption analysis is to provide, for each function in a program, bounds on its energy cost that are parametric in the size of its inputs, such as the value of a number or the length of a list. For example, consider the following C program.

```c
int fact(int n){
    if (n <= 0) return 1;
    return n * fact(n - 1);
}
```

The analysis may output that the execution cost of the program is between $4.1n + 3.8\,\mu J$ and $5.1n + 4.2\,\mu J$, when the execution architecture is set to be XMOS XS1 with XS1-A16-128-FB217 processors. In this case, the deviation with experimental measures is between -11.7% and 7% (cf. chapter 3.5 of [28]).

CiaoPP current pipeline for energy consumption analysis of imperative programs, e.g. C programs, may be split into four parts, which we describe in the next sections.

- First, a cost semantic must be provided via an energy model. This model must be designed, and the level of abstraction chosen has an impact on the precision of the analysis. Depending on the method, this part may or may not be program independent.
- Then, the core of the program-dependent analysis starts, and a C program is transformed into Ciao, viewed as HCIR. This transformation ideally preserves functional and cost semantics up to some observational equivalence. In practice, we may already start to focus on cost semantics.
- To analyse Ciao programs, recurrence equations are extracted. Those recurrence equations are of two kinds: *size* and *cost* relations. Interestingly, those equations may be seen as equivalent to a Ciao program where data is abstracted to its *size*.
- Finally, those recurrence equations must be solved. We describe current techniques in Section 2.4.

This pipeline may be mapped out as in Figure 3, where we emphasise that the pipeline is built on ideas of abstract interpretation. The general idea of abstract interpretation is presented in Figure 2.

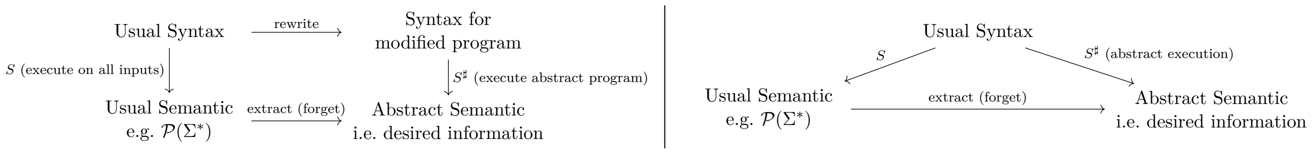
Usual Syntax
rewrite
Syntax for modified program
S (execute on all inputs)
$S^\sharp$ (execute abstract program)
Usual Semantic e.g. $\mathcal{P}(\Sigma^*)$
extract (forget)
Abstract Semantic i.e. desired information
Usual Syntax
S
$S^\sharp$ (abstract execution)
Usual Semantic e.g. $\mathcal{P}(\Sigma^*)$
extract (forget)
Abstract Semantic i.e. desired information

Figure 2: Idea of abstract interpretation.
Rather than doing "execute then abstract", do "abstract then execute" (left), or even "execute abstractly" (right).

The following paragraph is a commented version of Figure 2.

Suppose there is some information we want about the semantic of a program. The most straightforward way to get it is to execute the program on all possible inputs (going top left to bottom left), thus computing its full semantic (often a set of traces), and then extract this information (going from bottom left to bottom right). However, this is not practical: there is an infinite number of inputs and each trace may be extremely long. Abstract interpretation notices that it is a waste to compute an expensive object – the full semantic – only to throw out some parts after that, and that we would be better off forgetting irrelevant information *before* executing the program, either by rewriting it to a simpler program or by simplifying the execution semantic. The goal is then to make the diagrams "as commutative as possible".

$$\begin{tikzcd}
\texttt{C} \arrow[r] \arrow[d, red] & \texttt{HCIR} \arrow[r, dashed, "\alpha_{\mathrm{Sz}}"] \arrow[d, red] \arrow[dr, red] & \texttt{HCIR}^{(\texttt{Size})} \arrow[r, "\text{extract}"] \arrow[d, red] & \texttt{RecEq}(\mathrm{Sz} + \mathrm{Cost}) \arrow[dd, "\text{solve}"] \\
\mathcal{T}_{races}(\mathrm{Mem}^{(C)}, \mathrm{Cost}) \arrow[r, phantom, "\approx"] \arrow[dr, blue] & \begin{array}{c}\mathcal{T}_{rees}(\mathrm{Mem}^{(HC)}) \\ + \text{ Cost model for builtins}\end{array} \arrow[r, dashed, "\overline{\alpha_{\mathrm{Sz}}}"] \arrow[d, blue] & \begin{array}{c}\mathcal{T}_{rees}(\mathrm{Mem}^{(HC)}_{\mathrm{Size}}) \\ + \text{ Cost model for builtins}\end{array} \arrow[d, blue] & \\
& \mathcal{F}_{\mathcal{P}} + \mathrm{Cost}_{\mathcal{P}} \arrow[r, "\alpha^{\circ}_{\mathrm{Sz}}"] & \mathrm{Sz}_{\mathcal{P}} + \mathrm{Cost}_{\mathrm{Sz},\mathcal{P}} \arrow[r, "\alpha^{\circ}_{\mathcal{I}}"] & \mathrm{Sz}_{\mathcal{I}} + \mathrm{Cost}_{\mathrm{Sz},\mathcal{I}}
\end{tikzcd}$$

Figure 3: Energy analysis pipeline as the top path of an abstraction diagram.
Red arrows execute code (on all inputs), and blue arrows extract functions from traces and trees. Section 2.1 focuses on the cost models enabling the bottom left triangle. Section 2.2 corresponds to the top left square. Section 2.3 explains the middle square and the "extract" arrow. Finally, Section 2.4 describes the "solve" arrow.

In the diagram presented in Figure 3, note that execution trees are the equivalent for logic programs of execution traces for imperative programs, and correspond to concrete code semantics. Mem stands for "memory state". The arrows in our C execution traces are weighted, to allow cost semantics.

The bottom row contains the desired functional and cost semantics. Note that while we are only interested in cost semantics, some level of functional information is required to be able to compose functions, which will be explained in Section 2.3. $\mathcal{F}_{\mathcal{P}}$ is a full functional semantic, which assign to each function in the program a function $\texttt{Input} \to \mathcal{P}(\texttt{Output})$ recording all possible outputs for each input. $\mathrm{Cost}_{\mathcal{P}}$ is a full cost semantic, which assign to each function in the program a function $\texttt{Input} \to \mathcal{P}(\mathbb{R}_+)$ recording all possible costs for each input. $\mathrm{Sz}_{\mathcal{P}}$ and $\mathrm{Cost}_{\mathrm{Sz},\mathcal{P}}$ are abstracted versions of $\mathcal{F}_{\mathcal{P}}$ and $\mathrm{Cost}_{\mathcal{P}}$ respectively, where input/output *values* are replaced with input/output *sizes*. $\mathrm{Sz}_{\mathcal{I}}$ and $\mathrm{Cost}_{\mathrm{Sz},\mathcal{I}}$ are further abstractions, where rather than collecting all possible output sizes/costs, we provide an interval in which those output *may* appear, i.e. bounds on those outputs.

2.1 Energy models

In order to compute energy consumption at the source code level, we must provide energy models, which relate execution traces with energy consumption. The more fine-grained the model, the most precision can be obtained, but this also means that more information must be collected in traces and their abstractions.

For example, an energy model might assert that some low-level operation on XMOS architecture always costs 1210759 fJ to be executed, and write so in CiaoPP's assertion language.

```
:- trust pred sub_3r2(X)
    : var(X) => (num(X), rsize(X,num(A,B)))
    + (is_det, not_fails, cardinality(1,1), costb(energy,1210759,1210759)).
```

To have a complete model, the same must be done for every low-level instruction, e.g. via experimental measures, where we execute multiple times the same instruction, with random input data and initial state.

Such a constant model is however very simplistic. In practice, there is some variability, and we need to use bounds on the consumption rather than exact values. Moreover, focusing on instructions only might be too coarse. To account for some gate-switching, we can gain precision by providing the energy cost for all *pairs* of instructions. Moreover, random input data might gave bad estimates of the worst-case, as energy consumption might be very data-dependent [30] (data-switching accounts for ∼ 30% of dynamic consumption in XMOS experiments).

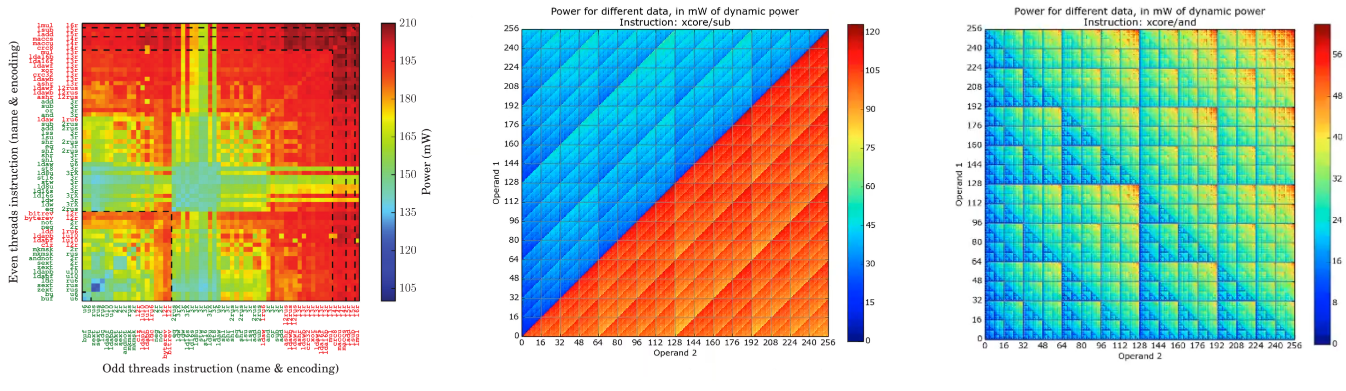

Figure 4: Measures presented by Kerstin Eder [48] on an XMOS architecture, for the average consumption of pairs of instructions on 32-bit data (left), and data-dependent consumption for the `sub` (middle) and and instructions (right).

More generally, to take architecture into account, it is possible to create models which parameterise the consumption by a list of previous instructions, or by the state of the architecture (e.g. cache state). Note that in some modern architectures, such as this NVIDIA GPU for supercomputers [49], fetching operands takes more energy than computing on them, and there may be a 100-fold consumption difference between a cache miss and cache hit.

In general, modern hardware's complex "runtime policies" are an important limit to the precision of analyses, in particular when safe upper bounds are critical ("Worst-Case Energy Consumption", e.g. for embedded software). Currently, one must choose depending on the application whether some lack of safety is acceptable, e.g. to focus on average consumption to optimise for complex architecture, or not, in which case we are restricted to very simple architectures (common in embedded software) or very large bounds.

For the moment, we talked about models at the ISA level. Another possibility is to work at a higher level, such as at the level of LLVM or even C source. This tends to lessen energy models' precision, but to improve analyses, as dataflow information becomes more structured. C level is mainly used for coarse complexity analysis (for example, a unit cost might be assigned to every instruction). At the LLVM level, two main techniques have been explored.

- Compile LLVM to the ISA level, relate the two representations, and map back to the LLVM program using the ISA energy model [50].
- Execute each LLVM basic block on the target machine (multiple times, with random initial values and initial states), and measure the consumption. This makes the analysis program-dependent, but can still be quite fast, as all recursive structure of the program is ignored. Moreover, the user can now perform the analysis on his architecture independently of what is provided by developers of the tool. However, the architecture must be available for execution.

We note a few propositions for future work.

- Make a working prototype of this idea of measuring the consumption of each LLVM basic block.
- Explore automated creation of models using hardware's VHDL, to gain in generality, as creating a model for a single chip is currently extremely expensive and time-consuming. Note that VHDL are not always provided by constructors.
- Explore the impact and possibility to analyse low-level aspects such as cache behaviour using known abstract interpretation methods. Note that cache models are often not provided by constructors. One way to mitigate it would be to use retroengineering tools such as those used in side-channel cryptanalysis.

2.2 Ciao as HCIR, C/LLVM/ISA $\to$ Ciao

In order to make our analysis multilingual, an intermediate representation is chosen, corresponding to horn clauses (HCIR) and written here in Ciao. The first obvious advantage is that it makes it possible to apply analyses present in CiaoPP to other languages. A deeper advantage of HCIR is that it uses *function call* as the only control structure, with no side-effect. Thus, it makes analyses simpler and focuses on the core problem: recursion.

Transformations of imperative programs to HCIR are based on the discovery of basic blocks in the control flow graph and on the translation of any higher-level control structure (cf. chapter 2 of [28]). Figure 5 gives an example of imperative code converted to HCIR.

Assertion files must be provided in order to give functional and cost semantic to each builtin construct of the source language. Note however that a perfect functional semantic is not necessary, and it can be sufficient to describe only basic properties of those constructs.

Current translations are prototypes and can be refined. Some future work ideas are listed in the following.

- Much information about data structures is lost and discarded, in particular in LLVM types. Properties of arrays, or complex structs, are mostly forgotten – we can hope to reconstruct those data structures in Ciao, perhaps using some amount of shape analysis, to gain precision. Another example is the size of integers, where 8 bits and 32 bits are currently treated the same by our HCIR, even though there often is at least a 2-fold energy consumption difference.
- Additionally, control structures are often translated in a way that makes analysis difficult. The most striking example is `for` loops with increasing argument where cost analysis fails to infer precise bounds. A quick fix would be to translate common `for` loops differently, and a more interesting solution would be to use some automated inference of ranking functions.

```
int fact(int n){
    if(n <= 0)
        return 1;
    return n*fact(n-1);
}
```

```
<fact>:
001: entsp  0x2
002: stw    r0, sp[0x1]
003: ldw    r1, sp[0x1]
004: ldc    r0, 0x0
005: lss    r0, r1
006: bf     <008>

007: bu     <010>
010: ldw    r0, sp[0x1]
011: sub    r0, r0, 0x1
012: bl     <fact>

013: ldw    r1, sp[0x1]
014: mul    r0, r1, r0
015: retsp  0x2

008: mkmsk  r0, 0x1
009: retsp  0x2
```

```
fact(R0, R0_3) :-
    entsp(0x2),
    stw(R0, Sp0x1),
    ldw(R1, Sp0x1),
    ldc(R0_1, 0x0),
    lss(R0_2, R0_1, R1),
    bf(R0_2, 0x8) ,
    fact_aux(R0_2, Sp0x1, R0_3, R1_1).

fact_aux(1, Sp0x1, R0_4, R1) :-
    bu(0x0A),
    ldw(R0_1, Sp0x1),
    sub(R0_2, R0_1, 0x1),
    bl(fact),
    fact(R0_2, R0_3),
    ldw(R1, Sp0x1),
    mul(R0_4, R1, R0_3),
    retsp(0x2).

fact_aux(0, Sp0x1, R0, R1) :-
    mkmsk(R0, 0x1),
```

Figure 5: A C program (left) translated into ISA level (middle) and HCIR from ISA (right).

2.3 HCIR → `RecEq` ∼ `Size program`

In this section, we explain how recurrence equations are set up in the current pipeline. The details are not a core part of the internship, but do represent both a foundation and a bottleneck of the tool. We present some core notions and sketch their inference algorithms. We then propose a more abstract viewpoint and ideas for future work.

Core standard intuitions are the following.

- A *size metric* m is some chosen measure of the "size" of data, such as the length of a list, the value of a number, or the depth of a term.
- *Size relations* relate sizes of multiple arguments (of literals in the program). In practice, this leads to *size functions* $\mathrm{Sz}_p^{(i)}$ that take as argument the sizes of input arguments of the predicate p, and return bounds on the size of p's i-th argument.
- *Cost relations* relate the cost of executing a predicate with other predicates' cost and arguments' sizes. This leads to *cost functions* Cost_p that take as argument the sizes input arguments of p, and return bounds on the cost of executing it. This cost is measured according to some chosen measure, which could be time, energy, number of resolutions, number of unifications, or some user-defined resource.

If we want cost functions, size functions are necessary for composition. Schematically, if $p(n) := f(g(n))$ in functional notation, $\mathrm{Cost}_p(n) = \mathrm{Cost}_f(\mathrm{Sz}_g(n)) + \mathrm{Cost}_g(n)$.

To read the rest of this section, it is useful to remember basic facts about the `Prolog` language[1]. To exemplify, consider the following quicksort program. When analysing with `entry qsort(L,Ls):list(num)*var`, modes (i.e. input/output arguments) and types are inferred. Here, the output arguments are the second of `qsort`, third and fourth of `partition`, and third of `append`.

```
qsort([], []).
qsort([F|L], Ls) :-
    partition(L, F, L1, L2),
    qsort(L1, L1s), qsort(L2, L2s),
    append(L1s, [F|L2s], Ls).

append([], L, L).
append([H|L1], L2, [H|R]) :- append(L1, L2, R).
```

```
partition([], F, [], []).
partition([X|Y], F, [X|Y1], Y2) :-
    X =< F,
    partition(Y, F, Y1, Y2).
partition([X|Y], F, Y1, [X|Y2]) :-
    X > F,
    partition(Y, F, Y1, Y2).
```

- In practice, a *size metric* m is just a partial function $\mathcal{H} \to \mathbb{N}$. Here $m = \texttt{len}$ might be chosen on all lists. In CiaoPP, metrics can be user-defined and are partially inferred, either using types or directly term structures [16]. A metric can be extended to a function $\texttt{size}_m$ on all terms, by setting $\texttt{size}_m = n$ whenever $m(\theta(t)) = n$ for all substitutions θ making t ground. For example, $\texttt{size}_{\texttt{len}}([X, Y]) = 2$ even if the list contains free variables. Additionally, a function $\texttt{diff}_m$ is defined on pairs of terms such that $\texttt{diff}_m(t_1, t_2) = d$ whenever t_2 is a subterm of t_1 and $m(\theta(t_2)) - m(\theta(t_1)) = d$ for all θ making both terms ground. For example, $\texttt{diff}_{\texttt{len}}([A, B|L], L) = -2$.

[1]Data in Prolog (and in Ciao) is represented by first-order terms, which may contain both free variables and ground atoms. Compound terms are defined using "functors". Data is a priori untyped, although functors may help to define datatypes, inferred by CiaoPP. The Herbrand universe $\mathcal{H}$ is the set of ground terms, and a substitution θ is a (partial) function from variables to terms.

- To give examples of *size relations*, we denote $\mathrm{sz}(p_i)$ the size of the i-th argument of a predicate p. The analysis may be able to infer that, at predicates' success, $\mathrm{sz}(\mathtt{append}_3) = \mathrm{sz}(\mathtt{append}_1) + \mathrm{sz}(\mathtt{append}_2)$, that $\mathrm{sz}(\mathtt{partition}_3) + \mathrm{sz}(\mathtt{partition}_4) = \mathrm{sz}(\mathtt{partition}_1)$ and that $\mathrm{sz}(\mathtt{qsort}_2) = \mathrm{sz}(\mathtt{qsort}_1)$.
  In practice, the analysis is currently only able to infer input/output relations, and not output/output relations like the one on `partition`, and can thus only get $\mathrm{sz}(\mathtt{partition}_3) \in [0, \mathrm{sz}(\mathtt{partition}_1)]$, which is why our size functions $\mathrm{Sz}_p^{(i)}$ are only parameterised by *input* sizes.
  The algorithm that infers those size relations works by repeatedly applying the `size` and `diff` functions in the directed acyclic graph of literal dependencies, and doing substitutions to simplify the relations. In the case of recursive or mutually recursive predicates, recurrence relations are generated and handed to the solver. In the `qsort` example, the upper bound analysis might infer the following relations, where $a \in \{3, 4\}$, after doing a max on output size of `partition`'s two recursive clauses without using the conditions on `F`.

$$\begin{aligned}
&\mathrm{Sz}^{(3)}_{\mathtt{append}}(0, m) = m && \mathrm{Sz}^{(3)}_{\mathtt{append}}(n+1, m) = 1 + \mathrm{Sz}^{(3)}_{\mathtt{append}}(n, m)\\
&\mathrm{Sz}^{(a)}_{\mathtt{partition}}(0, f) = 0 && \mathrm{Sz}^{(a)}_{\mathtt{partition}}(n+1, f) \le 1 + \mathrm{Sz}^{(a)}_{\mathtt{partition}}(n, f)\\
&\mathrm{Sz}^{(2)}_{\mathtt{qsort}}(0) = 0 && \mathrm{Sz}^{(2)}_{\mathtt{qsort}}(n+1) = \mathrm{Sz}^{(3)}_{\mathtt{append}}\Big(\mathrm{Sz}^{(2)}_{\mathtt{qsort}}\big(\mathrm{Sz}^{(3)}_{\mathtt{partition}}(n, \_)\big), 1 + \mathrm{Sz}^{(2)}_{\mathtt{qsort}}\big(\mathrm{Sz}^{(4)}_{\mathtt{partition}}(n, \_)\big)\Big)
\end{aligned}$$

  After some recurrence solving and substitutions, this leads to the exact $\mathrm{Sz}^{(3)}_{\mathtt{append}}(n, m) = n + m$, but only $\mathrm{Sz}^{(a)}_{\mathtt{partition}}(n, f) \le n$ and $\mathrm{Sz}^{(2)}_{\mathtt{qsort}}(n+1) \le 1 + 2 \times \mathrm{Sz}^{(2)}_{\mathtt{qsort}}(n)$, and thus an exponential bound on `qsort`. Note that the precision would be greatly improved with output/output relations.
- To set up *cost relations* and *functions*, the analyser must then compute the resource consumption of each predicate, split into the consumption of each clause, split into the consumption in the head and in each body literal. In this example, the predicate *qsort* has two clauses for empty or non-empty list, unification in the head of the second clause must unify an argument with $[F|L]$ structure, and the body literals of the second clause use all three predicates defined in this code. The way those different levels are combined is resource-dependent. For a deterministic non-failing program, we may sum the costs of body literals and take the maximum of the cost of each clause. Possibility of failure must also be taken into account, and – as logic programs are non-deterministic – the number of solutions of a call may also have to be used and estimated. More details may be found in [12, 13, 51, 16].

  In the example above, for a resource related to time such as number of resolution steps, we could get the following equations, where the last line is obtained after some substitution and recurrence solving, with imprecision in output sizes of `qsort` and `append` leading to a $O(n2^n)$ bound on $\mathrm{Cost}_{\mathtt{qsort}}$.

$$\begin{aligned}
&\mathrm{Cost}_{\mathtt{append}}(0, m) = 1 && \mathrm{Cost}_{\mathtt{append}}(n+1, m) = 1 + \mathrm{Cost}_{\mathtt{append}}(n, m)\\
&\mathrm{Cost}_{\mathtt{partition}}(0, f) = 1 && \mathrm{Cost}_{\mathtt{partition}}(n+1, f) = \max(1 + \mathrm{Cost}_{\mathtt{partition}}(n, f), 1 + \mathrm{Cost}_{\mathtt{partition}}(n, f))\\
&\mathrm{Cost}_{\mathtt{qsort}}(0) = 0
\end{aligned}$$

$$\begin{aligned}
\mathrm{Cost}_{\mathtt{qsort}}(n+1) &= 1 + \mathrm{Cost}_{\mathtt{partition}}(n, \_) + \mathrm{Cost}_{\mathtt{qsort}}(\mathrm{Sz}^{(3)}_{\mathtt{partition}}(n, \_)) + \mathrm{Cost}_{\mathtt{qsort}}(\mathrm{Sz}^{(4)}_{\mathtt{partition}}(n, \_))\\
&\quad + \mathrm{Cost}_{\mathtt{append}}\Big(\mathrm{Sz}^{(2)}_{\mathtt{qsort}}\big(\mathrm{Sz}^{(3)}_{\mathtt{partition}}(n, \_)\big), 1 + \mathrm{Sz}^{(2)}_{\mathtt{qsort}}\big(\mathrm{Sz}^{(4)}_{\mathtt{partition}}(n, \_)\big)\Big)\\
&\le 1 + n + 2^n + 2 \times \mathrm{Cost}_{\mathtt{qsort}}(n).
\end{aligned}$$

Some ideas for future work are listed in the following bullet points.

- Although this is not exactly what happens here, an ideal notion of *size* might be viewed as the coarsest approximation of *data* that preserves cost semantics on a particular program.

  Studying the discrepancy between this ideal notion and the implementation could lead us to new ideas to design and infer measures, as well as a better understanding of their limits.
- If we replace a Ciao program by a similar program in which data is replaced with its size measure, we obtain something which is extremely similar to the recurrence equations inferred. Note that recurrence equations might be viewed as recursive "`RecEq`" programs (acting on size data – note that this is different from Section 4 where equations are viewed as sequence operators).

  Studying precisely this analogy, we could complete the right half of Figure 3 with a stronger connection between $\mathtt{HCIR}^{\mathrm{Size}}$ and `RecEq`. We will need to describe precisely the operational semantic of `RecEq`.

  We hope that this sheds a new light on the connection between rewriting of programs and rewriting of recurrence equations, hence giving new ways of simplifying equations extracted from programs.
- One instance of rewriting technique that we want to implement (to improve analysis of imperative loops) is related to automated inference of ranking functions. Other rewriting techniques coming from recurrence equations are changes of variables.

- An improvement to the size analysis framework would be to infer some output/output relations. Ideas towards that are proposed in [13, 52, 53].
- It might be interesting to analyse higher-order programs, e.g. for analysis of functional code, which would probably require a sized type of functions, using ideas encountered in [42]: the size of a function would be some function relating the size of its inputs to the size of its outputs.

2.4 Recurrence equation solver

The current recurrence solver is mostly based on lookup in a table of known solutions. Given an equation, it checks whether it has a known shape, such a low order linear equation on one variable with constant coefficients or some divide-and-conquer equation. It has a few capabilities to solve equations one multiple variables if only one actually changes in recursive calls, and can compute the closed-form of some sums and products (e.g. to notice a factorial).

It can also discharge equations to other tools such as `Mathematica`.

In sections 3 and 4, we propose multiple ideas to improve this solver.

3 Implementation exercises and solver extensions

The static analysis techniques designed at CLIP Lab, including the resource consumption analysis and recurrence solving tools developed during this internship, are implemented in CiaoPP.

One of the goal of this internship was to get used to coding in the Ciao language, and to get familiar with the codebase of CiaoPP by implementing a few functionalities.

Small extensions to the "table-lookup" recurrence solver are described in Section 3.1. A more interesting algorithm for irrelevant variables analysis, developed and implemented during the internship, is presented in 3.2. Finally, a large number of small bugs have been discovered during experiments, and some of them fixed: a short list is found in 3.3.

3.1 Complex numbers, 2nd and 3rd order linear recurrence equations

We added complex numbers to CiaoPP and introduced them to the cost module's algebraic expressions. We modified the functions that compute with algebraic expressions, and used safe overapproximations for cases we didn't want to deal with yet, e.g. comparison of expressions.

Thanks to this, it was possible to expand the solver by completing the resolution of second order linear recurrence equations with constant coefficients: it was previously impossible to deal with equations of negative discriminant, e.g. $f(n) = f(n-1) - f(n-2) + n$.

We also added a solution for third order linear recurrence equations with constant coefficients, using Cardano's method. Complex numbers were necessary for third order: general arguments from Galois theory may be used, but notice that even the simple equation $f(n) = f(n-3)$ leads to the polynomial $X^3 - 2$ which has complex roots. Consider the following example.

```
:- entry p/2 : list(gnd) * var.
p([], []).
p([X], [X]).
p([X,Y], [X,Y]).
p([_,Y,Z|R], T) :-
    p([Y,Z|R], L1), p([Z|R], L2), p(R, L3),
    append(L1, L2, L4), append(L4, L3, T).

append([], X, X).
append([H|X], Y, [H|Z]) :- append(X, Y, Z).
```

The analyser easily infers that $C_{\text{append}}(x, y) = x + y$, using list length as size measure. Moreover,

$$\begin{cases} \mathrm{Sz}_{\mathrm{p}}(0) = 0,\ \mathrm{Sz}_{\mathrm{p}}(1) = 1,\ \mathrm{Sz}_{\mathrm{p}}(2) = 2, \\ \mathrm{Sz}_{\mathrm{p}}(n) = \mathrm{Sz}_{\mathrm{p}}(n-1) + \mathrm{Sz}_{\mathrm{p}}(n-2) + \mathrm{Sz}_{\mathrm{p}}(n-3) \text{ when } n \geq 3. \end{cases}$$

Since there is no additional term, we can focus on the homogeneous solution – otherwise, a particular solution would be found. The corresponding polynomial is $X^3 - X^2 - X - 1$. Its roots are $r_1 \approx 1.8393$, $r_2 \approx -0.41964 - 0.60629i$ and $r_3 \approx -0.41964 + 0.60629i$, and are found using Cardano's method. The solution to the equation is thus of the shape $\mathrm{Sz}_{\mathrm{p}}(n) = a_1 r_1^n + a_2 r_2^n + a_3 r_3^n$. The solver then computes the coefficients using the initial conditions, and outputs

```
:- true pred p(_A,T) : [...] => ([...],
    size(ub,length,T,
     ccomplex(-0.25951582498161835,-0.1422223907459299)*ccomplex(-0.4196433776070804,-0.6062907292071993)**length(_A)
     +ccomplex(-0.25951582498161824,0.14222239074592993)*ccomplex(-0.41964337760708065,0.6062907292071993)**length(_A)
     +ccomplex(0.5190316499632366,-1.7214688391203412e-17)*1.8392867552141612**length(_A))),
```

and the same function for the lower bound (`lb`). Notice that a numerical error occurred, with $-1.72 \cdot 10^{-17}$ appearing instead of 0. Dealing with this is left for future work, and two solutions may be considered. The first is to use a domain for floating-point computation which tracks errors and provides an interval in which the exact value is found. An issue is that for large exponents, intervals become extremely wide as errors accumulate quickly. Another solution may be to use some amount of symbolic computation and to convert between cartesian and polar representation of complex numbers when suitable.

3.2 Irrelevant variables analysis

In order to simplify equations, an analyse+rewrite pass is implemented to remove irrelevant variables. More analyse+rewrite passes may be implemented in the future.

Consider the following program, which uses an accumulating variable to compute the factorial.

```
:- entry fact(N,R) : num * var.
fact(N,R) :- fact_aux(N,1,R).

fact_aux(N,A,R) :-
    N > 0, N1 is N - 1, A1 is N1 * A,
    fact_aux(N1,A1,R).

fact_aux(N,A,R) :-
    N =< 0,
    R = A.
```

Suppose that we want to compute the cost of executing `fact` measured as the number of arithmetic operations performed. We get $C_{\text{fact}}(n) = C_{\text{fact\_aux}}(n, 1)$ and

$$\begin{cases} C_{\text{fact\_aux}}(n,a) = 2 + C_{\text{fact\_aux}}(n-1,(n-1)\times a) & \text{if } n > 0, \\ C_{\text{fact\_aux}}(n,a) = 0 & \text{if } n \leq 0. \end{cases}$$

Clearly, $C_{\text{fact\_aux}}(n,a) = 2n$ and the second index of $C_{\text{fact\_aux}}$ is irrelevant here. We could write anything instead of $(n-1) \times a$ and still get the solution: the equation may be seen as a single variable equation on n.

$$\begin{cases} \tilde{C}_{\text{fact\_aux}}(n) = 2 + \tilde{C}_{\text{fact\_aux}}(n-1) & \text{if } n > 0, \\ \tilde{C}_{\text{fact\_aux}}(n) = 0 & \text{if } n \leq 0. \end{cases}$$

The current version of CiaoPP was unable to notice that, and failed to obtain any information on the solution: the output was $0 \leq C_{\text{fact\_aux}}(n,a) \leq +\infty$. However, such irrelevant indices are common in cost analysis of imperative programs, where many variables may appear in a basic block without being used.

To solve this, we propose a simple fixpoint algorithm which overapproximates the set of relevant indices.

For the sake of rigour, for $1 \leq i \leq k$, let us define an operation

$$\begin{aligned} \text{Trim}_i : (\mathbb{N}^k \to \mathbb{N}) &\to (\mathbb{N}^{k-1} \to \mathbb{N}) \\ f &\mapsto \big((x_1, ..., x_k) \mapsto f(x_1, ..., x_{i-1}, 0, x_{i+1}, ...x_k)\big), \end{aligned}$$

which creates a new function with one less index, and a dual operation

$$\begin{aligned} \text{Expand}_i : (\mathbb{N}^{k-1} \to \mathbb{N}) &\to (\mathbb{N}^k \to \mathbb{N}) \\ f &\mapsto \big((x_1, ..., x_k) \mapsto f(x_1, ..., x_{i-1}, x_{i+1}, ...x_k)\big). \end{aligned}$$

Definition 3.1. We say that an index i is *irrelevant* in the equation $S : (\mathbb{N}^k \to \mathbb{N}) \to (\mathbb{N}^k \to \mathbb{N})$ whenever, for every solution f_{sol} of $f_{\text{sol}} = S f_{\text{sol}}$, we have $f_{\text{sol}} = \text{Expand}_i \circ \text{Trim}_i(f_{\text{sol}})$, making the reduced function $\text{Trim}_i(f_{\text{sol}})$ a solution of the reduced equation $\tilde{S} : \tilde{f} \mapsto \text{Trim}_i \circ S \circ \text{Expand}_i(\tilde{f})$.

Our goal is to overapproximate the set $I \subset \mathcal{P}(\llbracket 1, k \rrbracket)$ of *relevant* variables in an equation S presented in the following form, where Ψ_j are expressions with $f, \overrightarrow{n}$ as free variables, and ϕ_j are a finite number of conditions such that the $\phi_j^{-1}(\texttt{true})$ cover $\mathbb{N}^k$ (or partition it if we want to force unicity of the solution).

$$S(f) = \overrightarrow{n} \mapsto \begin{cases} \dots \\ \Psi_j(f, \overrightarrow{n}) & \text{if } \phi_j(\overrightarrow{n}), \\ \dots \end{cases}$$

Note that in the case of the cost module we studied, ϕ_j are always boundary conditions (such as $n_i = 0$).

Our algorithm is based on the following fact.

Proposition 3.2. *The least fixed point of the following operator, which is monotone for inclusion, is an overapproximation of the set of relevant variables, i.e.* $\operatorname{lfp} F$ *contains the set of relevant variables.*

$$\begin{aligned} F : \mathcal{P}(\llbracket 1,k \rrbracket) &\to \mathcal{P}(\llbracket 1,k \rrbracket) \\ I &\mapsto I \cup \{ i \,|\, n_i \textit{ appears in a condition } \phi_j \} \\ &\quad \cup \{ i \,|\, n_i \textit{ appears in an expression } \Psi_j \textit{ via a path going only through } f \textit{ via indices } i' \in I \}. \end{aligned}$$

Proof. (Sketch) When evaluation $f_{\text{sol}}(\overrightarrow{n})$ by unrolling $(S f_{\text{sol}})(\overrightarrow{n})$, an n_i may only be used in the result in the following three cases. It may appear in condition, and thus impact the result. It may appear in an expression outside recursive calls, and thus be used directly in the result. Finally, it may be used in a recursive call to define an index whose value impacts the result. □

Thanks to the domain's small height, we can simply iterate F from $\varnothing$ to compute $\operatorname{lfp} F$ in time $O(k)$.

Example 3.3. Consider the following equation on $f : \mathbb{N}^5 \to \mathbb{N}$.

$$\begin{cases} f(n_1,n_2,n_3,n_4,n_5) = 0 & \text{if } n_1 \le 0 \\ f(n_1,n_2,n_3,n_4,n_5) = 1 + f(n_1 - n_3 - 1, n_1 \times n_2, n_2 \times n_3, n_3 \times n_4, n_4 \times n_5) & \text{if } n_1 > 0. \end{cases}$$

- Initially, we set $I = \varnothing$.
- In the next step, we add 1 as n_1 appears in a boundary condition.
- Then, we add 3, as n_3 is required to compute the value in the first index of f in the recursive call.
- Finally, we add 2, as n_2 is required to compute the value in the third index of f in the recursive call. A fixed point is reached, as no new index is required to compute n_2. We get $\operatorname{lfp} F = \{1,2,3\}$, and we can simplify the equation to

$$\begin{cases} \tilde{f}(n_1,n_2,n_3) = 0 & \text{if } n_1 \le 0 \\ \tilde{f}(n_1,n_2,n_3) = 1 + \tilde{f}(n_1 - n_3 - 1, n_1 \times n_2, n_2 \times n_3) & \text{if } n_1 > 0. \end{cases}.$$

This algorithm was implemented in CiaoPP, in the Ciao language, in around 250 lines of code.

It is easy to expand this algorithm to work with sets of mutually recursive equations.

3.3 Other updates and bug discovery

While doing experiments on CiaoPP's cost analysis, we had the opportunity to notice a number of small bugs or outdated modules. Some of those observations are listed here.

- The LLVM pass (written in C++) used to convert LLVM code into Ciao was not compatible with LLVM13 due to a change of pass manager. A simple change in options was enough to let the pass work again.
- In a number of places, when the cost analysis module fails while trying to compute lower bounds, in order to safely say "I don't know", it outputs the constant function $+\infty$. However, this is only sound in the case of upper bounds. This was changed to 0.
- There are errors in a section of code which compares sequences together.

  While most of the code adopts the convention that `max`(f,g) means an *overapproximation* of $n \mapsto \max(f(n), g(n))$ when we compute a upper bound, and an underapproximation when we compute a lower bound, and similarly for `min`, this is forgotten in a portion of code computing minimum of expressions. Moreover, it forgets that two sequences might be non-comparable (e.g. $n \mapsto 10$ and $n \mapsto n$).

```
min_approximation(S1, S2, Sol) :-
    max_expr(S1, S2, Sol1),
    (
        (Sol1 == inf ; Sol1 == bot) ->
        normal_form(0, Sol)
    ;
        (
            Sol1 == S1 -> Sol = S2 ; Sol = S1
        )).
```

This leads to unsoundness errors. For example, the module might assert $x \leq \min(x, y)$ when x and y are free variables, or similarly that $x \leq \min(x, 2)$.

Moreover, we notice that, when computing the maximum of f and g, the code resorts to the overapproximation $f + g$ even in cases where it would have been easy to be more precise (in some cases, it does only $\max(x+1, y+1) \leq x + y + 2$, or $\max(n+3, 2) \leq n+3$ but only $\max(n+3, 4) \leq n+7$).

We plan to fix those modules, and also to give them a bigger role by using more determinacy analysis and max/min expressions, which are particularly useful when analysing imperative programs.

- Several abstract domains were not running anymore. An example is the numerical domain of polyhedra, which we wanted to test as it is often used for the analysis of imperative programs. The bug was fixed by a PhD student in the team, but further experiments revealed several limitations already present in the domain. For example, in the case of the following program, the domain infers nothing about R, whereas the information R $= 3 \times$ N would be inferred by a domain as weak as intervals.

```
p(N,R) :-
    K is 3,
    R is K*N.
```

This is caused by the inability of CiaoPP's polyhedra domain to deal with "non-linear" multiplications.

One easy solution is to write a reduced product with the domain of intervals (via an inclusion of polyhedra in hypercubes), either using CiaoPP's functionalities for reduced products or directly in the code of the polyhedra domain when a non-linear operation is performed.

We plan to use polyhedra to infer more precise recurrence equations, in particular for imperative programs.

4 Algorithmic ideas and theory: recurrence solving via order theory and abstract interpretation

In this section, we present new ideas on recurrence solving based on order theory. Important facts about this theory and abstract interpretation are recalled in Annex B.

The general framework is introduced is Section 4.1. Recurrence equations will be viewed as fixpoint equations for a sequence operator, and we will be particularly interested in the case where the operator is monotone. Then, in Section 4.2, we will see how pre- and post-fixpoints appear as "easily checkable bounds" on an equation's solution, and we explore how to use this fact for resolution via guess-and-check and divide-and-conquer methods. Finally, Section 4.3 describes how we apply abstract interpretation to recurrence solving, via abstract domains of bounds on sequences.

4.1 Order-theoretical framework

We apply the theory of abstract interpretation to recurrence equations – which correspond to programs – in order to compute, or rather approximate, their solution – viewed as their semantic. Everything starts from the following core observation: just like a program's semantic can be described as a fixpoint of some operator, a recurrence equation can always be viewed as a fixpoint equation on the domain of sequences.

Consider the following nested recurrence equation, on $(a, f) \in \mathbb{N} \times (\mathbb{N} \to \mathbb{N})$.

$$\begin{cases} f(0) & = a \\ f(n) & = f(f(n-1)) + 1, \ \forall n \in \mathbb{N}^*. \end{cases}$$

Clearly, a sequence f is solution of the equation if and only if it is a fixed point of the following operator, where $a \in \mathbb{N}$ is a chosen constant,

$$\begin{aligned} S : (\mathbb{N} \to \mathbb{N}) &\to (\mathbb{N} \to \mathbb{N}) \\ g &\mapsto \left(n \mapsto \begin{cases} a & \text{if } n = 0 \\ g(g(n-1)) + 1 & \text{otherwise} \end{cases} \right). \end{aligned}$$

Playing with this equation by hand, it is not too hard to prove that $(n \mapsto n, 0)$ is the only solution. However, it is harder to automate the resolution of this kind of equations in general. Adding an order to $\mathbb{N} \to \mathbb{N}$, the characterisation of f_{sol} as $\operatorname{lfp} S$ hints towards (several) methods of resolution.

For example, iterating from $\bot = (n \mapsto 0)$ just like before, we get the sequences of sequences

$$\begin{aligned} &0^* \mapsto 01^* \mapsto 012^* \mapsto ... \mapsto (n \mapsto n) && \text{when a} = 0, \\ &0^* \mapsto a1^* \mapsto a2^* \mapsto a3^* \mapsto ... && \text{when a} > 0, \end{aligned}$$

where the sequences are written in regex form, the first sequence converges to the solution, and the second sequence diverges, indicating the absence of solution for $a > 0$.

This is interesting: this suggests that we can compute the solutions of recurrence equations by iterating the operator corresponding to the equation.

It makes sense: according to Cousot and Cousot's theorem, for a *monotone* operator S in a *complete lattice*, the chain $(S^\alpha(\bot))_{\alpha\in\mathbf{Ord}}$ converges – perhaps in transfinitely many steps – to the least fixed point $\operatorname{lfp} S$. Of course, here, since we can diverge, $\mathbb{N} \to \mathbb{N}$ is not a domain. Moreover, the operators corresponding to recurrence equations will not always be monotone.

To fix those pitfalls, we use $\mathbb{N}_\infty \triangleq \mathbb{N} \cup \{\infty\}$, which extends $\mathbb{N}$ with a top element[2], and we restrict ourselves to monotone recurrence equations (to deal with the general case, we introduce methods to "monotonise" equations). We thus work on the following *domain of sequences*[3].

$$\begin{aligned} &D \triangleq \mathbb{N}_\infty \to \mathbb{N}_\infty, && \bigsqcup_i f_i = (n \mapsto \max_i f_i(n)), && \bigsqcap_i f_i = (n \mapsto \min_i f_i(n)), \\ &f \leq_D g \overset{\Delta}{\iff} \forall n \in \mathbb{N}_\infty,\, f(n) \leq_{\mathbb{N}_\infty} g(n), && \bot_D = (n \mapsto 0), && \top_D = (n \mapsto \infty) \end{aligned}$$

In the next two sections, we will describe ideas on solving recurrence equation by using this fixpoint characterisation of their solution.

In Section 4.2, we will use the notions of *pre- and post-fixpoint*, which instantiate in our case to *easily checkable bounds*, used to get approximate solutions by both divide-and-conquer and guess-and-check methods, either manually or with some level of automation.

Then, in Section 4.3, we will present some ideas on approximate recurrence solving by *abstract interpretation*, creating abstract domains $D^\sharp$ of bounds on sequences and abstract operators $S^\sharp$ used to perform *abstract iteration.*

4.2 A simple method for proving bounds

Consider a monotone[4] recurrence equation in one variable[5], and let $S : D \to D$ be the corresponding operator on sequences. Suppose that this equation has a single solution $f_{\text{sol}} = \operatorname{lfp} S = \operatorname{gfp} S$. We can make the following observation about pre- and post-fixpoints.

Proposition 4.1. *Let $g \in D$ be a sequence.*

If $Sg \leq g$, then $f_{sol} \leq g$. Similarly, if $g \leq Sg$, $g \leq f_{sol}$.

In other words, postfixpoints (resp. prefixpoints) can be viewed as easily checked upper (resp. lower) bounds of the solution.

Note that the converse is false: being a pre/postfixpoint is a stronger property than being a lower/upper bound of the solution.

4.2.1 A technique for humans

The simple observation above gives a simple procedure for checking if a sequence chosen as a candidate bound on the solution of a recurrence equation is indeed a bound.

Example 4.2. Consider the following equation, written as a sequence operator. Such equation may arise while doing worst-case analysis of a quicksort program.

$$\begin{aligned} &S : D \to D \\ &f \mapsto n \mapsto \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \max\limits_{1\leq k\leq n-1} f(k) + f(n-k) + n & \text{otherwise} \end{cases} \end{aligned}$$

[2]This also accommodates non-terminating programs and infinite sizes.

[3]$\mathbb{N}_\infty$ is the ordinal $\omega + 1$ and enjoys the descending chain condition (DCC). D is a complete lattice of infinite height (we can have large well-founded chains, e.g. of length $\omega^2 + \omega + 1$), and doesn't enjoy the DCC (consider $1^* \to 01^* \to 001^* \to ...$).

[4]i.e. such that the corresponding operator is monotone

[5]We could easily extend the results of this report to multiple variables, but we keep a single dimension for ease of presentation.

The operator is indeed monotone. Let $g : n \mapsto n^2$ be a candidate upper bound on f_{sol}.

We have $(Sg)(0) = 0$, $(Sg)(1) = 1$, and $(Sg)(n) = 1^2 + (n-1)^2 + n = n^2 - n + 2$ when $n \geq 2$, thus $(Sg)(n) \leq g(n)$ for all n.

Hence, g is a postfixpoint of S, and thus an upper bound on the solution: $\forall n$, $f_{\text{sol}}(n) \leq n^2$.

There is something remarkable and counterintuitive about this proof process: if only skimmed through, the proof may appear like the following. "Let g be a candidate upper bound on f_{sol}. We would have $f_{\text{sol}} \leq g$, thus $f_{\text{sol}} = Sf_{\text{sol}} \leq Sg$ by monotonicity of S. In this case, we observe $Sg \leq g$, i.e. we have deduced a stronger upper bound from a candidate upper bound. *Thus*, g is indeed an upper bound on f_{sol}".

The process is surprising, and this last *thus* seems like a reasoning mistake: "suppose a thing, deduce a stronger thing, thus the thing is actually true". Nevertheless, the proof is correct – this strongly relies on the monotonicity of S and the properties of pre- / post-fixpoints.

Note that the simplicity and generality of this proof makes it much more amenable to automation than most proofs of the same fact that could be found in undergraduate textbooks. We only need to be able to perform two operations: *compute* Sg, and *compare* Sg *with* g. Those order-based proofs are also shorter than standard proofs, at least once theory is established.

Example 4.3. Similarly, consider the equation $f(0) = 0, f(1) = 1, f(n) = 2f(\lfloor \frac{n}{2} \rfloor) + n$.

For $n \in 0, 1$, $S(n \mapsto n^2)(n) = n^2$. For $n \geq 2$, we have $S(n \mapsto n^2)(n) = 2\lfloor \frac{n}{2} \rfloor^2 + n \leq \frac{1}{2}n^2 + n \leq n^2$. Thus, $f_{\text{sol}} \leq (n \mapsto n^2)$. With some more computation, with $g \overset{\Delta}{=} n \mapsto 2n\lfloor \log_2 n \rfloor + 1$ we get $Sg \leq g$ hence $f_{\text{sol}} \leq g$.

Example 4.4. Consider an equation of the shape $f(n) = u_n$ for $n < k$ and $f(n) = \sum_{i=1}^{k} c_i f(n-i)$ with k, u_n and c_i chosen positive constants.

Using the same arguments as before extended to sequences of reals, if α is a real such that $u_n \leq \alpha^n$ for each $n < k$ and $\sum_{i=1}^{k} c_i \alpha^{k-i} \leq \alpha^k$, then we have $f_{\text{sol}}(n) \leq \alpha^n$ for all $n \in \mathbb{N}$.

4.2.2 Usage in guess-and-check methods

Of course, in our analysis, we would like to automatically generate bounds on the solutions, and not have to input them by hand.

One possibility is to *execute* an equation S to obtain the exact values of f_{sol} at a few indices, and use them to try to *guess* the solution via heuristics, before checking that it actually is a solution.

This generic idea was explored by Maximiliano Klemen in the 5th chapter of his PhD thesis [36]. There, he proposes to do the following.

1. *Train.* Generate a random set X_{train} of indices and compute $f_{\text{sol}}(x)$ for all $x \in X_{\text{train}}$.
2. *Guess.* Fit the curve by using quadratic regression with Lasso regularisation (i.e. minimise the ℓ_2 distance while penalising the ℓ_1 norm), using cross-validation for the regularisation coefficient.
   Do it using a given set of candidate functions, representative of the most common complexity orders, e.g. $T = \{\lambda x.x, \lambda x.x^2, \lambda x.x^3, \lambda x.\lceil \log_2(x) \rceil, \lambda x.2^x, \lambda x.x\lceil \log_2(x) \rceil\}$.
   We obtain a candidate $f_{\text{candidate}} = \sum_{g \in T} \alpha_g g$ and remove coefficients $|\alpha_g| < \epsilon$.
3. *Check.* If $f_{\text{candidate}}$ coincides with f_{sol} on all $x \in X_{\text{train}}$, use a SMT-solver to verify the solution, i.e. that the expressions $(f_{\text{candidate}})(n)$ and $(Sf_{\text{candidate}})(n)$ are equal for all n.

One of the shortcomings of this method is that the *guessing* step has to produce the *exact* value of the solution, which is clearly quite hard.

We propose to relax this condition, by trying to generate only *bounds* on the solution, rather than its exact value, thus providing a much larger chance of success. Two difficulties arise.

1. While it is reasonably easy to check if $f_{\text{candidate}} = f_{\text{sol}}$ using only the equation (f_{sol} is unknown), there is a priori no systematic way of checking whether $f_{\text{sol}} \leq f_{\text{candidate}}$ without knowing f_{sol} (most proofs methods would use an induction, which is difficult to automate).
2. In the guessing step, the cost function $\beta \mapsto \|y - X\beta\|_2^2 + \lambda\|\beta\|_1$ is not suited to generating strict upper or lower bounds.

To solve the first issue, we propose to use our pre-/post-fixpoint technique. If $Sf_{\text{candidate}} \leq f_{\text{candidate}}$, than the candidate is an upper bound. It is still much easier to guess a postfixpoint than to guess the exact solution, and automating the comparison between $Sf_{\text{candidate}}$ and $f_{\text{candidate}}$ is a reasonable problem, at least for simple candidate functions. For the second issue, we propose to use common techniques for optimisation under (hard or soft) constraints. Experiments will have to be performed to see what works best in our case.

This guess-and-check method may be used in conjunction with divide-and-conquer heuristics in order to produce more precise bounds on the solution.

4.2.3 Ideas towards divide-and-conquer methods

Usually, in the context of abstract interpretation, we are interested in computing a fixed point $\operatorname{lfp} S$, and we propose to overapproximate it via an abstract operator $S^\sharp$ working in a simpler domain than S.

However, when solving recurrence equations, we can take advantage of the fact that the solution is usually unique, i.e. there is a unique fixpoint $\operatorname{lfp} S = \operatorname{gfp} S$. Thanks to that, specialised technique can be designed, using the pre-/post-fixpoints properties presented above.

In the last section, we explained how to use the notion of "easily checkable bound" in guess-and-check methods. To go a bit further, notice that, once a bound is found by the guess-and-check method of the last section, we can keep looking for tighter bounds, with the search space now reduced.

In this section, we mention ideas to build upon those dichotomy-like ideas, with the hope of building, not only heuristics, but algorithms that always compute good approximations of the solution, or even converge towards it. The situation of our lattice among related work can be framed as the following.

- In a total order, e.g. $L_t = [\![1, n^d]\!]$, it is easy to use dichotomy to compute a fixpoint of any monotone $f : L_t \to L_t$ in $O(d \log n)$.
- In the last three years, ideas have been proposed to compute fixpoints in logarithmic time in other lattices, where points may be non-comparable, thus making dichotomy trickier. More precisely, [54] and [55] propose algorithms to compute fixpoints of monotone functions in the lattice $L_c = [\![1, n]\!]^d$ – equipped with the product order – in time $O(\log^d n)$ and $O(\log^{2\lceil d/3\rceil} n)$ respectively.
- The case we are interested in is somehow intermediate between those two kinds of lattices. For example, consider the lattice $L = \{\sum_{i=0}^{d-1} a_i X^i \mid 1 \le a_i \le n\}$ of polynomials with bounded degree and bounded natural coefficients, ordered by $P \le Q \iff \forall n \in \mathbb{N},\ P(n) \le Q(n)$.

  Clearly, L_c is a wide sublattice of L: if $\forall i,\ P_i \le Q_i$, then $P \le Q$. Moreover, L is not total (X^2 and $X+1$ are not comparable). However, L has many more relations than L_c: we have $X \le X^2$, $9X \le 20 + X^2$, $6X + 6X^3 \le 11X^2 + X^4$...

We thus have two lattices in which logarithmic algorithms are known to solve the fixpoint problem, while only linear algorithms are currently known in an intermediate case: this calls for exploration!

This is work in progress, and multiple questions will have to be solved.

- The lattice D of sequences is too large, and we must deal with an abstraction step to a $D^\sharp$ such as the lattice of polynomials.
- Those algorithms apply to finite lattices: to use them, we need to first find an upper bound on $\operatorname{lfp} S$, and testing all values in a fast growing chain such towards the top (such as $1, 2(1+X), 3(1+X+X^2), ...$) is not guaranteed to provide postfixpoints in all lattices.
- The algorithms of [54, 55] strongly use the recursive structure of $[\![1, n]\!]^d$, and it is not clear yet how to deal with this obstruction. Moreover, they strongly use the fact that recursive calls compute a exact fixpoint, so new ideas have to be found if we want to use only bounds.

Nevertheless, we are convinced that using the "unique fixed point" properties of recurrence equations are a promising research direction, either using guess-and-check heuristics or converging algorithms,

In the next section, we will present the "more classic" method of computing $\operatorname{lfp} S$ by abstract iteration, without assumptions on the unicity of the fixed point.

4.3 Abstract iteration – Solving equations by abstract interpretation

In this section, we propose to overapproximate the solution of recurrence equations by *abstractly executing* the equations, viewed as sequence operators. To do so, we introduce abstract domains of sequences and bounds on sequences. The general framework is presented in Section 4.3.2, while examples are given in sections 4.3.1 and 4.3.3. Finally, 4.3.4 provides perspective on extensions and research lines explored during the internship. Additional detail can be found in Annexes C and D.

4.3.1 Introductory example

Consider the operator $S : D \to D$ corresponding to the equation $f(0) = 0$ and $\forall n > 0$, $f(n) = f(f(n-1)) + 1$. We use a domain A of affine sequences and a domain $D^\sharp = (\mathcal{P}(A), \supseteq)$ of affine upper bounds on sequences. The goal is to compute $f_{\text{sol}} = \operatorname{lfp} S$, and we overapproximate it by computing $\operatorname{lfp} S^\sharp$ where $S^\sharp : D^\sharp \to D^\sharp$. In the Galois connection $(D, \leq_D) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{P}(A), \supseteq)$, the abstraction $\alpha(f)$ is the set of all affine upper bounds of a sequence f, and the concretisation $\gamma(U)$ is the highest sequence that verifies all upper bounds in U.

We start from $U_0 := \bot_{D^\sharp} = A = \uparrow\{\bot_A\} = \uparrow\{n \mapsto 0n + 0\}$, which means[6] that, initially, all affine sequences are considered candidate upper bounds on the solution. At each step, we will then remove candidates until a fixpoint is reached, where we will only have actual bounds on f_{sol}.

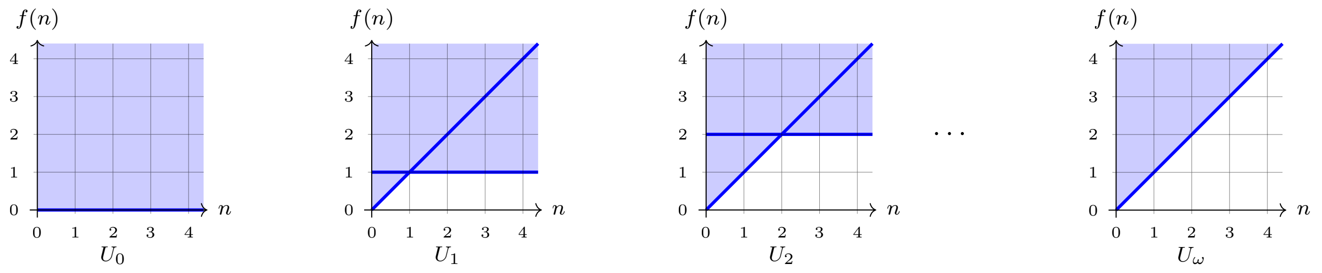

Figure 6: Candidate upper bounds (light blue), are progressively removed by applying S on all of them. It is sufficient to only apply S on extremal bounds (dark blue). Between U_1 and U_2, $(n \mapsto n)$ is fixed, which shows that it is a correct upper bound on f_{sol}. When the fixpoint U_ω is reached, only correct upper bounds are left.

For the first step, we apply S on the candidate upper bound, and we get[7] $S(n \mapsto 0n + 0) = 01^*$, hence the candidate upper bounds $U_1 = \alpha(01^*) = \uparrow\{n \mapsto 1n + 0, n \mapsto 0n + 1\}$.

Then, we once again apply S on the candidates, getting $S(n \mapsto 1n + 0) = (n \mapsto 1n + 0)$ and $S(1^*) = 02^*$. At this step, the analysis discovers that $f_{\text{sol}} \leq (n \mapsto 1n + 0)$. However, it doesn't know whether the other candidate can help refine the bound, and may try to keep going, setting $U_2 = \alpha(n \mapsto 1n + 0) \cup \alpha(02^*) = \uparrow\{n \mapsto 1n + 0, n \mapsto 2n + 0, n \mapsto 0n + 2\} = \uparrow\{n \mapsto 1n + 0, n \mapsto 0n + 2\}$ where a simplification step is taken for the last equality.

If the analysis keeps going, it will get $U_k = \uparrow\{n \mapsto 1n + 0, n \mapsto 0n + k\}$ for $3 \leq k < \omega$, progressively removing all constant candidates, before finally reaching $U_\omega = \uparrow\{n \mapsto 1n + 0, n \mapsto 0n + \infty\} = \uparrow\{n \mapsto 1n + 0\}$, which is a fixpoint of the abstract operator. Note that widenings could have been used to get there in a finite number of steps.

In the end, the analysis correctly inferred that $f_{\text{sol}} \leq (n \mapsto 1n + 0)$.

We could do the same for lower bounds, and reach $(n \mapsto 1n + 0) \leq f_{\text{sol}}$.

4.3.2 Abstract framework

In this section, we give more precise definitions on our framework.

Domain of abstract sequences. The main reason for which abstraction is required is that sequences, i.e. elements of D, cannot be finitely represented in general. Hence, we start with an abstract domain A, whose elements represent sequences and admit a finite representation, with $\phi : A \to D$ a concretisation operator, and $f^\sharp \leq_A g^\sharp \iff \phi(f^\sharp) \leq_D \phi(g^\sharp)$ For example, we could choose for A to be *the domain of affine sequences*, with $\phi((a, b)) \triangleq (n \mapsto an + b)$. Unfortunately, this doesn't lead to a Galois connection.

Proposition 4.5 (No easy connections). *With A the domain of affine sequences and ϕ the concretisation operator above, there are no Galois connections $D \underset{\alpha}{\overset{\phi}{\leftrightarrows}} A$, and neither in the other direction.*

Proof. (Sketch) The element $12^* \in D$ doesn't admit a best abstraction in A. In fact, it admits the two incomparable minimal upper bounds $(n \mapsto 1n + 1)$ and $(n \mapsto 0n + 2)$. □

To fix this, we can use the powerset trick, and record *all* true bounds rather than trying to choose one.

Definition 4.6 (Domain of abstract bounds). We call $D^\sharp \triangleq (\mathcal{P}(A), \sqsubseteq)$ the domain of abstract bounds. Note that since the inclusion order is reversed, $\sqcup^\sharp = \cap$, $\sqcap^\sharp = \cup$, $\bot^\sharp = A = \uparrow\{\bot_A\}$, and $\top^\sharp = \varnothing$.

[6]Given a domain $(L, \leq)$ and a subset $S \subseteq L$, $\uparrow S \triangleq \cup_{x \in S}\{u \in L \mid x \leq u\}$ is called the *upper closure* of S
[7]Sequences are often written in regex shape.

Proposition 4.7. *We have a Galois connection* $D \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} D^\sharp$, *defined by*

$$\begin{aligned} \alpha &: D \to D^\sharp & \gamma &: D^\sharp \to D \\ f &\mapsto \{f^\sharp \in A \,|\, f \leq_D \phi(f^\sharp)\}, & U &\mapsto (n \mapsto \min_{f^\sharp \in U} \phi(f^\sharp)(n)). \end{aligned}$$

Moreover, for any U, $\gamma \circ \alpha(U) = \uparrow U$. *In particular,* $\alpha(U) = \alpha(\uparrow U)$.

The $U \in D^\sharp$ may still contain an infinite number of elements, which make it seems like we have not made progress. However, most of the $\uparrow U$ are finitely representable, using their *generators*, or *extremal bounds*. For example, 12^* has an infinite number of affine upper bounds, but $\alpha(12^*) = \uparrow \{(1,1),(0,2)\}$ can be represented with only 4 numbers.

Proposition 4.8 (**Extremal bound**)**.** *If* A *verifies the descending chain condition, then for any* $U \in \mathcal{P}(A)$, *there exists a unique* U_{extr} *minimal for inclusion such that* $\uparrow U = \uparrow U_{extr}$, *called* set of extremal bounds of U.

Moreover, U_{extr} *is an antichain of* A. *In particular,* U_{extr} *is always finite if all antichains are finite.*

Note that both hypotheses are verified for all domains studied here.

In implementations, we introduce a `simplify` operation which removes redundant from bounds from representations. For example, for affine bounds, $\texttt{simplify}(\{(1,1),(0,2),(2,1)\}) = \{(1,1),(0,2)\}$.

Interestingly, this operator is the $\gamma_{\mathfrak{A}}$ of a Galois connection $(\mathfrak{A}(A), \sqsubseteq_{\mathfrak{A}}) \underset{\alpha_{\mathfrak{A}}}{\overset{\gamma_{\mathfrak{A}}}{\leftrightarrows}} (\mathcal{P}(A), \supseteq)$ where $\mathfrak{A}(A)$ is the set of antichains of A, with $\alpha_{\mathfrak{A}}(U) := \uparrow U$ and $\sqsubseteq_{\mathfrak{A}}$ chosen to make $\alpha_{\mathfrak{A}}$ and $\gamma_{\mathfrak{A}}$ monotone.

Abstract operators and transfer functions. We now explain how we obtain a sound abstraction $S^\sharp : D^\sharp \to D^\sharp$ given a concrete sequence operator $S : D \to D$ representing a recurrence equation.

Ideally, we would choose the best abstraction $S^\sharp = \alpha \circ S \circ \gamma$, but this is not computable.

We use the common solution which gives its name to abstract interpretation. The idea is to notice that $S = [\![\mathrm{Com}_1]\!] \circ ... \circ [\![\mathrm{Com}_k]\!]$, where Com_i are constructs in the syntax of the language used to define S, with $[\![\cdot]\!]$ their semantic. We can then set $S^\sharp = [\![\mathrm{Com}_1]\!]^\sharp \circ ... \circ [\![\mathrm{Com}_k]\!]^\sharp$, where $[\![\mathrm{Com}_i]\!]^\sharp$ are sound (or even best) abstractions of $[\![\mathrm{Com}_i]\!]$ which can be computed by hand, giving rise to an abstract semantic of the language.

To do so in a clear way, we define the syntax and semantic of a *sequence operator language* defined in Annex C, which can be used to describe recurrence equations.

Additionally, we decide that it is too complicated to define directly abstract operators on sets of bounds, so we just define them on each extremal bound before making a union, i.e. $[\![\mathrm{Com}]\!]^\sharp : A \to D^\sharp$, extended for $U \in D^\sharp$ to $[\![\mathrm{Com}]\!]^\sharp(U) := \cup_{f^\sharp \in \texttt{simplify}(U)} [\![\mathrm{Com}]\!]^\sharp(f^\sharp)$. It is easy to prove that this is sound.

4.3.3 Abstract domains and transfer functions applied to equations

To design good abstract interpretation frameworks and domains, much of the work is in defining good transfer functions, i.e. $[\![\mathrm{Com}]\!]^\sharp$ that compose well and are not too expensive to compute.

Abstract domains designed during this internship include a domain of *affine sequences*, a domain of *arithmetico-geometric sequences* and a domain of *polynomial sequences*. Additionally, a domain modifier which allows for *initial exactness*, i.e. to remember exactly the first few values of sequences, has been created. Other experiments were performed on “cheap” polynomial sequences and on a domain allowing for both logarithmic, polynomial and exponential sequences, but were mostly inconclusive for the moment because of lack of precision in either composition or shifting arguments. Annexes being already large, we didn’t include all abstract domains in them, but precise abstract semantics of the domains of affine sequences and initial exactness can be found in Annex D.

In this section, we develop another example of recurrence solving by abstract interpretation, with a higher level of details than in Section 4.3.1. Consider the following nested recurrence equation.

$$\begin{cases} f(0) &= 0 \\ f(n) &= f\left(f\left(\left\lfloor \frac{n}{2} \right\rfloor\right)\right) + 1 \text{ when } n \neq 0 \end{cases}$$

We parse it into a combination of basic operators in the following way. We first have self composition, then add the constant 1, then scale n into $n/2$, then set the initial value to 0. In the language presented in Annex C, this is the operator $\mathrm{Set}_0\ 0\ (\mathrm{Div}_{in}\ 2\ (+\ (\mathrm{Cst}\ 1)\ (\circ(\texttt{f})(\texttt{f}))))$.

The exact solution of this equation is $f_{\mathrm{sol}} = 01223^*$. Abstract iteration with the domain A of affine sequences returns the bound $f_{\mathrm{sol}} \leq (n \mapsto 1n + 0)$. Abstract iteration with the domain A_5 of affine sequences extended to remember the exact value of the first 5 indices returns the bound $f_{\mathrm{sol}} \leq 01223^*$.

Example 4.9 (Computation with affine sequences)**.** Results are the same as in Figure 6.

- We start with $U_0 = \uparrow\{\bot_A\}$, where $\bot_A = (0,0)$, which we will write $0n+0$ for the sake of readability.
- To compute U_1, we do the following. First, self composition gives $[\![\circ]\!]^\sharp_A(0n+0, 0n+0) = \{0n+0\}$. We add Cst 1, which is $\{0n+1\}$, giving the result $\{0n+1\}$. Then, we apply Set_0 0 (Div_{in} 2 $\cdot$). Since $0 \le (1-0)\times 2$ and $0 \le 1 \le 0 + (1-0)\times 2 - 0$, we enter the special case, and output $U_1 = \{0n+1,\ 1n+0\}$.
- We compute U_2. First, we use the candidate $0n+1$. Self composition fixes it, adding 1 gives $0n+2$, and Set_0 0 (Div_{in} 2 $\cdot$) enters the special case to give $\{0n+2, 2n+0\}$. Then, we use the candidate $1n+0$. Self composition fixes it, adding 1 gives $1n+1$, and Set_0 0 (Div_{in} 2 $\cdot$) enters the special case to give $\{1n+1, 1n+0\}$. $1n+0$ is discovered to be a correct bound on f_{sol}. We then output $U_2 = \texttt{simplify}(\{0n+2, 2n+0\} \cup \{1n+1, 1n+0\}) = \{0n+2, 1n+0\}$.
- In all future steps, $U_k = \{0n+k, 1n+0\}$ for $3 \le k < \omega$, and $U_\omega = \{1n+0\}$ is fixed. No new bound is discovered after step 2.

  We have obtained $f_{\text{sol}} \le (n \mapsto n)$ which is a good upper bound, but is not the best of affine bounds on the solution, which would be $\uparrow\{1n+0, 0n+3\}$.

Example 4.10 (Computation with initally exact affine sequences)**.** We represent the elements of A_5 as pairs, with the first part corresponding to the exact values of the 5 first indices in a sequence, and the second part corresponding to the remaining (shifted) affine behaviour, i.e. $\phi_{A_5}(\overrightarrow{u}, an+b)(n) = u_n$ if $n \le 4$ and $\phi_{A_5}(\overrightarrow{u}, an+b)(n) = a(n-5)+b$ otherwise.

Doing all computations by hand, we obtain the results $U_0 = \uparrow\{(00000, 0n+0)\}$, $U_1 = \uparrow\{(01111, 0n+1)\}$, $U_2 = \uparrow\{(01222, 0n+2)\}$ and $U_3 = \uparrow\{(01223, 0n+3)\}$, which is fixed.

The main subtleties in the computations are that we have to decide which part of the sequence (exact or affine) to use in self-composition, and we have to send some of the exact part to the affine part we replacing n by $n/2$. This is described in more details in Annex D.

We thus conclude that the solution to the equation is smaller than $\phi_{A_5}(01223, 0n+3) = 01223^*$, which is both true and optimal.

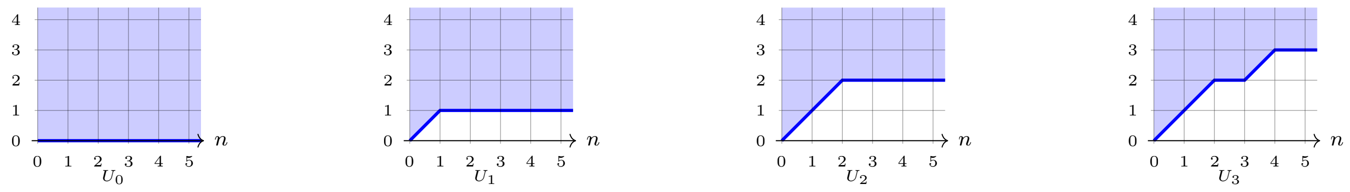

Figure 7: Abstract iteration removing candidate upper bounds, using a domain with initial exactness.

4.3.4 Extensions and perspectives

This short section lists other ideas on abstract iteration studied during this internship, as well as some lines of research for future work.

- Equations may be monotonised by a purely syntactic transformation, using the assumption $0 \le f_{\text{sol}} \le \infty$. We obtain different results depending on whether we are looking for an upper bound or a lower bound (S_{up} or S_{down}). Precision tends to be better for upper bounds.
- We can analyse for both upper and lower bounds at the same time, running Kleene sequences in parallel for all of $\text{lfp}\, S_{\text{up}}$, $\text{gfp}\, S_{\text{up}}$, $\text{lfp}\, S_{\text{down}}$ and $\text{gfp}\, S_{\text{down}}$. As sound bounds are discovered, they can be used to improve both transfer functions and monotonisations, either online or by restarting the Kleene sequences.
- When we studied manual proofs on nested recurrences such as $C(n) = C(n - C(n-1)) + C(n-1-C(n-2))$, we noticed that some proofs of the “slow growth” property (i.e. $\forall n,\ C(n) - C(n-1) \in \{0,1\}$) could be partly automatised with the help of an SMT-solver (studying inequality constraints).

  It would be interesting to see to what extent we could use the discovery of such properties to help the abstract solver. More generally, we are interested to see how to use assertions inside abstract iteration, or conversely if abstract iteration ideas can provide insight on the qualitative properties of solutions.
- Experiments showed that running a classical numerical domain on the numerical programs corresponding to recurrence equations leads to useful bounds on its solutions.

  For example, polyhedral analysis is very related to our domain of affine sequences, and both method infer the correct solution to $f(0) = 0$, $f(n) = f(f(n-1)) + 1$. However, among CiaoPP's domains, only our initially exact affine domain infers the correct solution for $f(n) = f(f(\lfloor n/2 \rfloor)) + 1$.

  Additionally, some experiments suggest that polyhedra can help to infer output/output size relations.

  This leaves open an exciting research line: study the relationships between numerical and sequence domains, as well as between representations of recurrence equations as numerical programs and sequence operators.

A References

[1] Patrick Cousot and Radhia Cousot. Systematic design of program transformation frameworks by abstract interpretation. *ACM SIGPLAN Notices*, 37(1):178–190, jan 2002.

[2] Ben Wegbreit. Mechanical program analysis. *Communications of the ACM*, 18(9):528–539, sep 1975.

[3] Marcel Celaya and Frank Ruskey. An undecidable nested recurrence relation. In S. Barry Cooper, Anuj Dawar, and Benedikt Löwe, editors, *How the World Computes*, pages 107–117, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[4] Patrick Cousot and Radhia Cousot. Abstract interpretation. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '77*. ACM Press, 1977.

[5] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTRÉE analyzer. volume 3444, pages 21–30, 10 2005.

[6] Matthieu Journault, Antoine Miné, Raphaël Monat, and Abdelraouf Ouadjaout. Combinations of reusable abstract domains for a multilingual static analyzer. In *Working Conference on Verified Software: Theories, Tools, and Experiments*, pages 1–18. Springer, 2019.

[7] Maurice Bruynooghe. A practical framework for theabstract interpretation of logic programs. *The Journal of Logic Programming*, 10(2):91–124, feb 1991.

[8] K. Muthukumar and M. Hermenegildo. Compile-time derivation of variable dependency using abstract interpretation. *The Journal of Logic Programming*, 13(2-3):315–347, jul 1992.

[9] Philipp Koerner, Michael Leuschel, Joao Barbosa, Vitor Santos Costa, Verónica Dahl, Manuel V Hermenegildo, Jose F Morales, Jan Wielemaker, Daniel Diaz, Salvador Abreu, et al. Fifty Years of Prolog and Beyond. *Theory and Practice of Logic Programming*, pages 1–83, 2022.

[10] Isabel García Contreras. *A scalable static analysis framework for reliable program development exploiting incrementality and modularity.* PhD thesis, June 2021.

[11] Saumya K. Debray, Nai-Wei Lin, and Manuel Hermnegildo. Task granularity analysis in logic programs. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, page 174–188, New York, NY, USA, 1990. Association for Computing Machinery.

[12] Saumya K. Debray and Nai-Wei Lin. Cost analysis of logic programs. *ACM Trans. Program. Lang. Syst.*, 15(5):826–875, nov 1993.

[13] Saumya Debray, Pedro López-García, Manuel Hermenegildo, and Nai-Wei Lin. Lower bound cost estimation for logic programs. In *Proceedings of the 1997 International Symposium on Logic Programming*, ILPS '97, page 291–305, Cambridge, MA, USA, 1997. MIT Press.

[14] H. Soza, M. Carro, and P. Lopez-Garcia. Probabilistic Cost Analysis of Logic Programs: A First Case Study. In *XXXII Latin-American Conference on Informatics*, August 2006.

[15] Pedro Lopez-Garcia, Maximiliano Klemen, Umer Liqat, and Manuel V. Hermenegildo. A general framework for static profiling of parametric resource usage, 2016.

[16] A. Serrano, P. Lopez-Garcia, F. Bueno, and M. V. Hermenegildo. Sized Type Analysis for Logic Programs (technical communication). In T. Swift and E. Lamma, editors, *Theory and Practice of Logic Programming, 29th Int'l. Conference on Logic Programming (ICLP'13) Special Issue, On-line Supplement*, volume 13, pages 1–14. Cambridge U. Press, August 2013.

[17] P. Lopez-Garcia, L. Darmawan, M. Klemen, U. Liqat, F. Bueno, and M. V. Hermenegildo. Interval-based Resource Usage Verification by Translation into Horn Clauses and an Application to Energy Consumption. *Theory and Practice of Logic Programming, Special Issue on Computational Logic for Verification*, 18:167–223, March 2018. arXiv:1803.04451.

[18] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The seahorn verification framework. pages 343–361, 2015.

[19] Bruno Blanchet. Using horn clauses for analyzing security protocols. *Cryptology and Information Security Series*, 5, 01 2011.

[20] J. Navas, M. Méndez-Lojo, and M. Hermenegildo. Safe Upper-bounds Inference of Energy Consumption for Java Bytecode Applications. In *The Sixth NASA Langley Formal Methods Workshop (LFM 08)*, pages 29–32, April 2008. Extended Abstract.

[21] Umer Liqat, Steve Kerrison, Alejandro Serrano, Kyriakos Georgiou, Pedro López-García, Neville Grech, Manuel V. Hermenegildo, and Kerstin Eder. Energy consumption analysis of programs based on xmos isa-level models. *23rd International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'13)*, pages 72–90, 01 2013.

[22] U. Liqat, Z. Banković, P. Lopez-Garcia, and M. V. Hermenegildo. Inferring Energy Bounds via Static Program Analysis and Evolutionary Modeling of Basic Blocks. In *Pre-proceedings of the 27th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'17)*, October 2017. arXiv:1601.02800.

[23] U. Liqat, K. Georgiou, S. Kerrison, P. Lopez-Garcia, M. V. Hermenegildo, J. P. Gallagher, and K. Eder. Inferring Parametric Energy Consumption Functions at Different Software Levels: ISA vs. LLVM IR. In M. Van Eekelen and U. Dal Lago, editors, *Foundational and Practical Aspects of Resource Analysis: 4th International Workshop, FOPARA 2015, London, UK, April 11, 2015. Revised Selected Papers*, volume 9964 of *Lecture Notes in Computer Science*, pages 81–100. Springer, 2016.

[24] V. Perez-Carrasco, M. Klemen, P. Lopez-Garcia, J.F. Morales, and M. V. Hermenegildo. Cost Analysis of Smart Contracts via Parametric Resource Analysis. In David Pichardie and Mihaela Sighireanu, editors, *Proceedings of the 27th Static Analysis Symposium (SAS 2020)*, volume 12389 of *LNCS*, pages 7–31. Springer, November 2020.

[25] Stefanos Kaxiras and Margaret Martonosi. *Computer Architecture Techniques for Power-Efficiency.* Springer International Publishing, 2008.

[26] Venkat Satagopan, Bonita Bhaskaran, Anshul Singh, and Scott C. Smith. Automated energy calculation and estimation for delay-insensitive digital circuits. *Microelectronics Journal*, 38(10):1095–1107, 2007.

[27] Peter Marwedel et al. Energy aware c compiler workflow. `https://ls12-www.cs.tu-dortmund.de/daes/en/research/energy-aware-c-compiler/encc-workflow.html`.

[28] Umer Liqat. *A multi-language and multi-platform framework for resource consumption analysis and its application to energy-efficient software development.* PhD thesis, 2018.

[29] Kerstin Eder, John P Gallagher, Pedro López-García, Henk Muller, Zorana Banković, Kyriakos Georgiou, Rémy Haemmerlé, Manuel V Hermenegildo, Bishoksan Kafle, Steve Kerrison, et al. Entra: Whole-systems energy transparency. *Microprocessors and Microsystems*, 47:278–286, 2016.

[30] James Pallister, Steve Kerrison, Jeremy Morse, and Kerstin Eder. Data dependent energy modeling for worst case energy consumption analysis. In *Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems*, pages 51–59, 2017.

[31] Christian Ferdinand, Florian Martin, Reinhard Wilhelm, and Martin Alt. Cache behavior prediction by abstract interpretation. *Science of Computer Programming*, 35(2):163–189, 1999.

[32] Reinhold Heckmann and Christian Ferdinand. ait: Worst-case execution time prediction by static program analysis. *International Federation for Information Processing Digital Library; Building the Information Society;*, 156, 01 2004.

[33] Stephen Wolfram. *The mathematica book*, volume 1. Wolfram Research, Inc., 2003.

[34] Roberto Bagnara, Andrea Pescetti, Alessandro Zaccagnini, and Enea Zaffanella. Purrs: Towards computer algebra support for fully automatic worst-case complexity analysis. *arXiv preprint cs/0512056*, 2005.

[35] Marko Petkovsek, Herbert S Wilf, and Doron Zeilberger. *$A = B$.* A K Peters, Natick, MA, January 1996.

[36] Maximiliano Klemen. *A General Framework for Static Resource Analysis and Profiling of (Parallel) Programs and an Application to Runtime Checking.* PhD thesis, 2020.

[37] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Automatic inference of upper bounds for recurrence relations in cost analysis. In *Static Analysis*, pages 221–237. Springer Berlin Heidelberg.

[38] Antonio Flores. *Cost Analysis of Programs Based on the Refinement of Cost Relations.* PhD thesis, TU Darmstadt, 2017.

[39] Jérôme Feret. The Arithmetic-Geometric Progression Abstract Domain. In Radhia Cousot, editor, *the 6th International Conference on Verification, Model Checking and Abstract Interpretation - VMCAI 2005*, volume 3385 of *Verification, Model Checking, and Abstract Interpretation*, pages 42–58, Paris, France, January 2005. Springer.

[40] Daniel Le Métayer. Ace: An automatic complexity evaluator. 10(2):248–266, apr 1988.

[41] Jan Hoffmann. *Types with Potential: Polynomial Resource Bounds via Automatic Amortized Analysis.* PhD thesis, Ludwig-Maximilians-Universität München, 2011.

[42] Álvaro Portillo, Kevin Hammond, Hans-Wolfgang Loidl, and Pedro Vasconcelos. Cost analysis using automatic size and time inference. pages 232–248, 09 2002.

[43] Ministère de l'Économie et des Finances. Réduire la consommation énergétique du numérique, 2019.

[44] The Shift Project. Déployer la sobriété numérique, plan de transformation de l'Économie française, 2020.

[45] France Stratégie. Maîtriser la consommation du numérique : le progrès technologique n'y suffira pas, 2020.

[46] Alliance Green IT, France Datacenter, and Gimélec. Livre blanc, les indicateurs de performance énergétique et environnementale des data centers, 2017.

[47] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Barroso. Power provisioning for a warehouse-sized computer. *ACM Sigarch Computer Architecture News*, 35:13–23, 06 2007.

[48] Kerstin Eder. Whole systems energy transparency: More power to software developers, 2021. Supergen Energy Networks Hub, `https://www.youtube.com/watch?v=Rm9JjqCCdNg`.

[49] Dally Bill. Gpu computing to exascale and beyond, 2010. NVIDIA, `https://www.nvidia.com/content/PDF/sc_2010/theater/Dally_SC10.pdf`.

[50] Kyriakos Georgiou, Steve Kerrison, and Kerstin Eder. On the value and limits of multi-level energy consumption static analysis for deeply embedded single and multi-threaded programs. *CoRR*, abs/1510.07095, 2015.

[51] Jorge Navas, Edison Mera, Pedro López-García, and Manuel V. Hermenegildo. User-definable resource bounds analysis for logic programs. In *Logic Programming*, pages 348–363. Springer Berlin Heidelberg.

[52] Roberto Giacobazzi, Saumya K Debray, and Giorgio Levi. Generalized semantics and abstract interpretation for constraint logic programs. *The Journal of Logic Programming*, 25(3):191–247, 1995.

[53] Florence Benoy and Andy King. Inferring argument size relationships with CLPR. In John Gallagher, editor, *Logic Program Synthesis and Transformation*, pages 204–223, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[54] Chuangyin Dang, Qi Qi, and Yinyu Ye. Computations and complexities of tarski's fixed points and supermodular games. *ArXiv*, abs/2005.09836, 2020.

[55] John Fearnley, Dömötör Pálvölgyi, and Rahul Savani. A faster algorithm for finding tarski fixed points. *ACM Trans. Algorithms*, mar 2022.

[56] Garrett Birkhoff. *Lattice Theory.* American Mathematical Society, dec 1940.

[57] Richard Dedekind. *Über Zerlegung von Zahlen durch ihre grössten gemeinsamen Theiler.* Vieweg, 1897.

[58] Bronisław Knaster. Un théorème sur les fonctions d'ensembles. *Ann. Soc. Polon. Math.*, 6:133 – 134, 1928.

[59] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285 – 309, 1955.

[60] P. Cousot and R. Cousot. Constructive versions of Tarski's fixed point theorems. *Pacific Journal of Mathematics*, 81(1):43–57, 1979.

[61] Mirna Džamonja, Sylvain Schmitz, and Philippe Schnoebelen. On ordinal invariants in well quasi orders and finite antichain orders. In *Trends in Logic*, pages 29–54. Springer International Publishing, 2020.

B Order-theoretical background

In this annex, we recall some background in order theory and abstract interpretation, useful to understand the ideas presented in Section 4.

We first recall facts from *lattice theory*[8]. This subfield of order theory is remarkably useful in program semantics and in the computation of (approximate) program properties, especially through the use of fixpoints. It has been largely studied by Birkhoff in the middle of the 20th century [56], and inspired by earlier work by Dedekind, such as [57]. To stay close to intuitions useful in this report, we directly define a strong object of this theory, the *complete lattice.*

Definition B.1. A *complete lattice* $(L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ is a partial order $(L, \sqsubseteq)$ which has[9]

1. A greatest lower bound $\sqcup S \in L$, for every subset $S \subset L$, called the *join* of S,
2. A least upper bound $\sqcap S \in L$, for every subset $S \subset L$, called the *meet* of S,
3. A least element $\bot$, called *bottom*,
4. A greatest element $\top$, called *top.*

Recall that, in a partial order $(L, \leq)$, three situations may occur given $x, y \in L$. Either $x \leq y$, or $y \leq x$, or x and y and incomparable. We call *chain* a subset of L containing elements which are all comparable to each other, and *antichain* a subset of L containing no distinct comparable elements.

Example B.2. The canonical example of a complete lattice is given by $(\mathcal{P}(S), \subseteq, \cup, \cap, \varnothing, S)$, the *powerset lattice* of a set S.

In the context of computer science, program semantics and verification, the order relation $\sqsubseteq$ may be used to order logical formulas, behaviours of programs, information, etc.

Moreover, as noted by domain theorists, this structure can be used to give meaning to programs which contains recursion (or loops), through the use of (least) fixed points.

Let us introduce some notations and useful theorems.

Definition B.3. A function $f : (L, \leq) \to (L', \sqsubseteq)$ between partial orders is said to be *monotone* whenever $\forall x, y \in L,\ x \leq y \implies f(x) \sqsubseteq f(y)$.

Definition B.4. An element $x \in X$ is said to be a fixpoint of $f : X \to X$ whenever $f(x) = x$.

Theorem B.5 (Knaster-Tarski [58, 59]). *Given a complete lattice L and a monotone function $f : L \to L$, the set of fixpoints of f is itself a complete lattice. In particular, f admits unique least and greatest fixed points, respectively written* $\operatorname{lfp} f$ *and* $\operatorname{gfp} f$.

Cousot and Cousot have given, as a generalisation of Kleene's ideas, a constructive version of this beautiful fixed point theorem [60].

Theorem B.6. *Given a complete lattice L and a monotone function $f : L \to L$,* $\operatorname{lfp} f$ *and* $\operatorname{gfp} f$ *can be computed by transfinite iteration*[10]*. Writing* **Ord** *the class of ordinals, we have*

$$\operatorname{lfp} f = \bigsqcup_{\alpha \in \mathbf{Ord}} f^{\alpha}(\bot) \quad \textit{and} \quad \operatorname{gfp} f = \bigsqcap_{\alpha \in \mathbf{Ord}} f^{\alpha}(\top).$$

Moreover, the notions of pre- and post-fixpoints will be of interest for us.

Definition B.7. Given a monotone function between complete lattices $f : L \to L$, $x \in L$ is said to be a prefixpoint whenever $f(x) \sqsubseteq x$, and a postfixpoint whenever $x \sqsubseteq f(x)$.

Proposition B.8. *If x is a prefixpoint,* $\operatorname{lfp}(f) \sqsubseteq x$. *If x is a postfixpoint,* $x \sqsubseteq \operatorname{gfp}(f)$.

[8] *Lattices in order theory* – called *treillis/retículos* in French/Spanish – are not to be confused with the *lattices in geometrical group theory*, more well-known in cryptography, and called *réseaux/redes* in French/Spanish.

[9] Note that only the first condition is necessary, and all others can be deduced from it.

[10] For limit ordinals λ, $f^{\lambda}(x)$ is defined as $\sqcup_{\alpha<\lambda} f^{\alpha}(x)$ in the case of the increasing sequence and as $\sqcap_{\alpha<\lambda} f^{\alpha}(x)$ in the case of the decreasing sequence. Otherwise, $f^0(x) := x$ and $f^{\alpha+1}(x) := f(f^{\alpha}(x))$.

To complete this annex, we go back to applications to program analysis through abstract interpretation.

In denotational semantics, the meaning of a program may be defined as $\operatorname{lfp} F$, where $F : D \to D$ is a operator corresponding to the program, and D is a complete lattice – called the *concrete domain* – related the execution environment.

The issue is that *the exact semantic* $\operatorname{lfp} F$ *is often non-computable.* This can be viewed as a consequence of the fact that the ordinal length of the chain $\bot \sqsubseteq F(\bot) \sqsubseteq F^2(\bot) \sqsubseteq ... \sqsubseteq \operatorname{lfp} F$ may be transfinite, because the *height*[11] of D may be transfinite.

To tackle this issue, abstract interpretation proposes to compute a *sound approximation* of the semantic. To do this, we replace the overly complicated lattice D by a simpler lattice $D^\sharp$, called the *abstract domain.* In this context, we construct a *sound abstraction* $F^\sharp : D^\sharp \to D^\sharp$ of F, and compute $\operatorname{lfp} F^\sharp$, which will be a *sound approximation* of $\operatorname{lfp} F$ thanks to a *fixpoint transfer theorem.*

More precisely, we ask for D and $D^\sharp$ to be related by a *Galois connection*, i.e. an adjunction of thin categories.

Definition B.9. A Galois connection between partial orders $(D, \sqsubseteq)$ and $(D^\sharp, \sqsubseteq^\sharp)$ is given by monotone functions $\alpha : D \to D^\sharp$ and $\gamma : D^\sharp \to D$, respectively called *abstraction* and *concretisation* operators, such that $\forall c \in D$, $\forall a \in D^\sharp$, $\alpha(c) \sqsubseteq^\sharp a \iff c \sqsubseteq \gamma(a)$.

In this case, we write

$$(D, \sqsubseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (D^\sharp, \sqsubseteq^\sharp).$$

Proposition B.10. *Equivalently,* $(D, \sqsubseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (D^\sharp, \sqsubseteq^\sharp)$ *is Galois connection whenever α is monotone, γ is monotone, $\gamma \circ \alpha$ is increasing and $\alpha \circ \gamma$ is decreasing.*

Definition B.11. Given a Galois connection, a *sound abstraction* of $c \in D$ is an $a \in D^\sharp$ such that $c \sqsubseteq \gamma(a)$, and a *sound abstraction* of $f : D \to D$ is an $f^\sharp : D^\sharp \to D^\sharp$ such that $\forall a \in D^\sharp$, $(f \circ \gamma)(a) \sqsubseteq (\gamma \circ f^\sharp)(a)$, or equivalently $(\alpha \circ f \circ \gamma)(a) \sqsubseteq^\sharp f^\sharp(a)$.

$$\begin{tikzcd} D \arrow[d, "f"'] & D^\sharp \arrow[l, "\gamma"'] \arrow[d, "f^\sharp"] \\ D \arrow[r, "\alpha"] \arrow[ur, phantom, "\sqsubseteq^\sharp"] & D^\sharp \end{tikzcd}$$

We also have notions of *best abstractions*, which are generally not computable.

Proposition B.12 (Composability)**.** *Given a Galois connection, if $f^\sharp$ and $g^\sharp$ are sound abstractions of f and g, then $f^\sharp \circ g^\sharp$ is a sound abstraction of $f \circ g$.*

This allows us to state the following fixpoint approximation theorem.

Theorem B.13. *Given a Galois connection, a monotone $F : D \to D$ and a sound abstraction $F^\sharp : D^\sharp \to D^\sharp$,* $\operatorname{lfp} F^\sharp$ *is a sound abstraction of* $\operatorname{lfp} F$*, i.e.*

$$\operatorname{lfp} F \sqsubseteq \gamma(\operatorname{lfp} F^\sharp).$$

More generally, any postfixpoint of $F^\sharp$ is a sound abstraction of $\operatorname{lfp} F$*.*

Thanks to all of this, when the height of $D^\sharp$ is finite, $\operatorname{lfp} F^\sharp$ is computable and provides a sound abstraction of the concrete semantic $\operatorname{lfp} F$.

Remark B.14 (Widenings)**.** Note that, in most applications, $\operatorname{lfp} F^\sharp$ may still be non-computable, because even if the abstract domain $D^\sharp$ is much simpler than the concrete domain D, the height of $D^\sharp$ may still be infinite or prohibitively high. To deal with this, a notion of *widening* is often used. Due to size limitations, we will not give a precise detail on widenings, but the idea is that it accelerates the convergence of sequences while adding overapproximations.

The following intuition may be given: a widening step may be viewed as a giant leap which overapproximates multiple small steps in the iteration chain. Since going through a limit ordinal may be viewed as computing an induction, a widening step may also be viewed as the overapproximation of the result of an induction.

[11]Defined as the supremum of the ordinal length of the well-ordered chains of D, cf. [61] for more background.

C Syntax and semantics of a sequence operators language

Remember that, for us, recurrence equations correspond exactly to operators on the domain of sequences.

In this annex, we describe the syntax and semantics of a small language – which we will call `Eqs` – describing operators on sequence. Since `Eqs`' syntax is finitely described, to perform abstract interpretation, it is sufficient to compute a sound abstraction of the semantic of each construction in the language.

Syntax. For the moment, `Eqs` allows basic arithmetic operations, initialisation at 0, and composition – used for nested equations. Moreover, it introduces shifting and scaling operations, which will be used to deal with recursive calls.

```
⟨full_equation⟩ ::= <initialisation> <expr>

⟨initialisation⟩ ::= Set_0 <nat>

⟨expr⟩ ::= ( <expr'> )

⟨expr'⟩ ::=
  | Shift <int> <expr> | Push <nat> <expr> | Pop <expr>
  | Mult_in <nat> <expr> | Div_in <nat> <expr>
  | + <expr> <expr> | × <expr> <expr>
  | ∘ <expr> <expr> | − <expr> <expr>
  | Cst <nat> | 'n'
  | 'f'

⟨nat⟩ ::= n ∈ ℕ    <int> ::= n ∈ ℤ
```

Figure 8: Syntax of `Eqs`.

For example, let u, v be constants in $\mathbb{N}$. Consider the following simple recurrence equation.

$$\begin{cases} f(0) & = v \\ f(n) & = f(n-1) + u \text{ when } n \neq 0, \end{cases}$$

In our language, it can be represented both by the expression $\text{Set}_0\ v\ (\text{Shift } 1\ (+\ (\text{Cst } u)\ (\texttt{f})))$, and the expression $\text{Push } v\ (+\ (\text{Cst } u)\ (\texttt{f}))$.

The operators "+" and "Cst u" are used to add u everywhere in the sequence. The operator "Shift 1" is used to deal with the recursive call $f(n-1)$. The operator "$\text{Set}_0\ v$" is used to enforce $f(0) = v$.

When we compute the semantics of the second expression, we get $[\![\texttt{f}]\!](f) = f$, $[\![\text{Cst } u]\!](f) = (n \mapsto u)$, $[\![+\ (\text{Cst } u)\ (\texttt{f})]\!](f) = (n \mapsto f(n) + u)$, hence

$$[\![\text{Push } v\ (+\ (\text{Cst } u)\ (\texttt{f}))]\!](f) = \left(n \mapsto \begin{cases} v & \text{if } n = 0 \\ f(n-1) + u & \text{if } n \neq 0, \end{cases}\right).$$

Semantics. The precise definition of $[\![\cdot]\!] : \texttt{Eqs} \to (D \to D)$ is given by structural induction.

$$[\![\text{Cst } c]\!](f) = (n \mapsto c)$$
$$[\![\texttt{n}]\!](f) = (n \mapsto n)$$
$$[\![\texttt{f}]\!](f) = f$$

$$[\![+\ e_1\ e_2]\!](f) = (n \mapsto [\![e_1]\!](f)(n) + [\![e_2]\!](f)(n))$$
$$[\![\times\ e_1\ e_2]\!](f) = (n \mapsto [\![e_1]\!](f)(n) \times [\![e_2]\!](f)(n))$$
$$[\![-\ e_1\ e_2]\!](f) = (n \mapsto [\![e_1]\!](f)(n) - [\![e_2]\!](f)(n))$$
$$[\![\circ\ e_1\ e_2]\!](f) = \Big(n \mapsto [\![e_1]\!](f)([\![e_2]\!](f)(n))\Big)$$

$$[\![\text{Push } c]\!](f) = \left(n \mapsto \begin{cases} c & \text{if } n = 0 \\ f(n-1) & \text{otherwise} \end{cases}\right)$$
$$[\![\text{Pop}]\!](f) = (n \mapsto f(n+1))$$
$$[\![\text{Shift } \delta]\!](f) = \begin{cases} [\![\text{Push } 0]\!]^{\delta}(f) & \text{for } \delta \geq 0 \\ [\![\text{Pop}]\!]^{|\delta|}(f) & \text{for } \delta \leq 0 \end{cases}$$
$$[\![\text{Set}_0\ c]\!](f) = [\![\text{Push } c]\!] \circ [\![\text{Pop}]\!](f)$$

$$[\![\text{Mult}_{in}\ c]\!](f) = (n \mapsto f(c \times n))$$
$$[\![\text{Div}_{in}\ c]\!](f) = \Big(n \mapsto f\Big(\Big\lfloor \frac{n}{c} \Big\rfloor\Big)\Big)$$

D Domains of affine sequences and initial exactness

D.1 Affine sequences

We consider the domain A of affine sequences, whose elements concretise as $f : n \mapsto an + b$ with $a, b \in \mathbb{N}_\infty$.

$$A \triangleq \mathbb{N}_\infty \times \mathbb{N} + \{\top_A\} \qquad \begin{aligned} \phi : A &\to D \\ (a,b) &\mapsto (n \mapsto an+b) \\ \top_A &\mapsto (n \mapsto \infty) \end{aligned} \qquad f^\sharp \sqsubseteq_A g^\sharp \overset{\Delta}{\iff} \phi(f^\sharp) \leq_D \phi(g^\sharp).$$

This gives us

$$(a_1, b_1) \sqsubseteq_A (a_2, b_2) \iff a_1 \leq_{\mathbb{N}_\infty} a_2 \wedge b_1 \leq_{\mathbb{N}} b_2,$$
$$\bigsqcup_i \{(a_i, b_i)\} = \big(\max_i a_i,\, \max_i b_i\big), \quad \text{and} \quad \bigsqcap_i \{(a_i, b_i)\} = \big(\min_i a_i,\, \min_i b_i\big)$$

A is a complete lattice, of height $\omega \cdot 2 + 1$, which enjoys the DCC. We can thus use the notion of extremal bounds like defined above.

Transfer functions We define $[\![\cdot]\!]^\sharp : \texttt{Eqs} \to (A \to \mathcal{P}(A))$ by structural induction.

This can be extended to $\texttt{Eqs} \to (\mathcal{P}(A) \to \mathcal{P}(A))$ by setting $[\![\mathrm{C}]\!]^\sharp(U) \triangleq \bigcup\{[\![\mathrm{C}]\!]^\sharp(f^\sharp) \mid f^\sharp \in U\}$.

$$\begin{aligned} [\![\mathrm{Cst}\ c]\!]^\sharp(f^\sharp) &= \{(0, c)\} \\ [\![\texttt{n}]\!]^\sharp(f^\sharp) &= \{(1, 0)\} \\ [\![\texttt{f}]\!]^\sharp(f^\sharp) &= \{f^\sharp\} \end{aligned}$$

To ease composition later on, for arithmetic operations, we first introduce operators $[\![\Diamond]\!]^\sharp : A \times A \to \mathcal{P}(A)$ for $\Diamond \in \{+, \times, \circ, -\}$, and then use this to define $[\![\Diamond\ e_1\ e_2]\!]^\sharp$.

$$\begin{aligned} [\![\Diamond]\!]^\sharp(\top_A, g^\sharp) = [\![\Diamond]\!]^\sharp(f^\sharp, \top_A) &= \{\top_A\} && \text{for } \Diamond \in \{+, \times, \circ, -\} \\ [\![+]\!]^\sharp((a_1, b_1), (a_2, b_2)) &= \big\{(a_1 + a_2, b_1 + b_2)\big\} \\ [\![\times]\!]^\sharp((a_1, b_1), (a_2, b_2)) &= \big\{(a_1 a_2 \infty + a_1 b_2 + a_2 b_1, b_1 b_2)\big\} && \text{where we set } 0 \times \infty = \infty \\ [\![\circ]\!]^\sharp((a_1, b_1), (a_2, b_2)) &= \big\{(a_1 a_2, a_1 b_2 + b_1)\big\} \\ [\![-]\!]^\sharp((a_1, b_1), (a_2, b_2)) &= \big\{(a_1 - a_2, b_1 - b_2)\big\} \end{aligned}$$

Notice that $[\![-]\!]^\sharp$ is not a sound abstraction of $[\![-]\!]$ in general, and has to be dealt with in a specific way.

$$[\![\Diamond\ e_1\ e_2]\!]^\sharp(f^\sharp) = \bigcup_{\substack{g_1^\sharp \in [\![e_1]\!]^\sharp(f^\sharp) \\ g_2^\sharp \in [\![e_2]\!]^\sharp(f^\sharp)}} [\![\Diamond]\!]^\sharp(g_1^\sharp, g_2^\sharp), \qquad \text{for } \Diamond \in \{+, \times, \circ\}$$

$$[\![-\ e_1\ e_2]\!]^\sharp(f^\sharp) = \begin{cases} \bigcup_{\substack{g_1^\sharp \in [\![e_1]\!]^\sharp(f^\sharp) \\ g_2^\sharp \in [\![e_2]\!]^\sharp(f^\sharp)}} [\![-]\!]^\sharp(g_1^\sharp, g_2^\sharp), & \begin{array}{l}\text{if we know that the approximation of } e_2 \\ \text{is exact and a singleton,} \\ \text{e.g. if } e_2 = \mathrm{Cst}\ c \text{ or } e_2 = \texttt{n}\end{array} \\ [\![e_1]\!]^\sharp(f^\sharp) & \text{otherwise (or use lower bounds)} \end{cases}$$

You may have noticed that, for the moment, we only create one bound for each individual operation applied in A. This is not the case anymore in operations derived from Push, where a single bound may give rise to two bounds.

$$[\![\mathrm{Push}\ c]\!]^\sharp(f^\sharp) = \begin{cases} \{(\infty, c)\} & \text{if } f^\sharp = \top_A \\ \{(a, c)\} & \text{if } f^\sharp = (a, b) \text{ and } b \leq a + c \\ \{(a, b - a), (b - c, c)\} & \text{if } f^\sharp = (a, b) \text{ and } b > a + c \end{cases}$$

$$[\![\mathrm{Pop}]\!]^\sharp((a, b)) = \begin{cases} \{\top_A\} & \text{if } f^\sharp = \top_A \\ \{(a, b + a)\} & \text{if } f^\sharp = (a, b) \end{cases}$$

$$\llbracket \text{Shift } \delta \rrbracket^\sharp(f^\sharp) = \begin{cases} (\llbracket \text{Push } 0 \rrbracket^\sharp)^\delta(f^\sharp) & \text{if } \delta \geq 0 \\ \{\top_A\} & \text{if } \delta \leq 0 \text{ and } f^\sharp = \top_A \\ \{(a, b + |\delta| a)\} & \text{if } \delta \leq 0 \text{ and } f^\sharp = (a,b) \end{cases}$$

$$\begin{aligned} \llbracket \text{Set}_0\ c \rrbracket^\sharp(f^\sharp) &= \llbracket \text{Push } c \rrbracket^\sharp \circ \llbracket \text{Pop} \rrbracket^\sharp(f^\sharp) \\ &= \begin{cases} \{(\infty, c)\} & \text{if } f^\sharp = \top_A \\ \{(a, c)\} & \text{if } f^\sharp = (a,b) \text{ and } b \leq c \\ \{(a,b), ((a+b)-c, c)\} & \text{if } f^\sharp = (a,b) \text{ and } b > c \end{cases} \end{aligned}$$

$$\begin{aligned} \llbracket \text{Mult}_{in}\ c \rrbracket^\sharp((a,b)) &= \{(ca, b)\} \\ \llbracket \text{Div}_{in}\ c \rrbracket^\sharp((a,b)) &= \left\{ \left(\left\lceil \frac{a}{c} \right\rceil, b \right) \right\} \end{aligned}$$

In addition, we add the following special case (more precise than doing the composition, because we only have access to integer slopes in the representation).

$$\llbracket \text{Set}_0\ c\ (\text{Div}_{in}\ d\ \cdot) \rrbracket^\sharp((a,b)) = \begin{cases} \left\{ \left(\left\lceil \frac{a}{d} \right\rceil, b \right), (b-c, c) \right\} & \begin{array}{l} \text{if } d \geq 2,\ a \leq (b-c)d \\ \text{and } c \leq b \leq c + (b-c)d - a \end{array} \\ \llbracket \text{Set}_0\ c \rrbracket^\sharp \circ \llbracket \text{Div}_{in}\ d \rrbracket^\sharp((a,b)) & \text{otherwise} \end{cases}$$

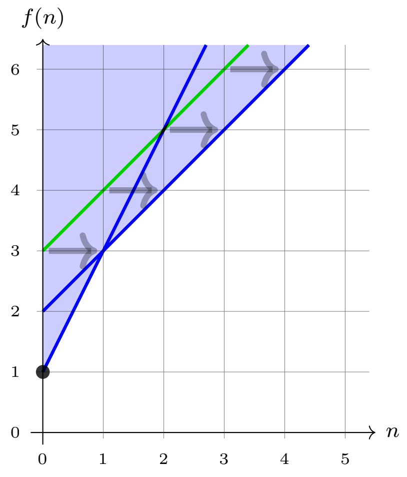

Figure 9: $1n+3$ (green), $\llbracket \text{Push } 1 \rrbracket(1n+3)$ (frontier), and $\llbracket \text{Push } 1 \rrbracket^\sharp(1n+3)$ (blue).
When pushing 1 to the beginning of the $1n+3$, the best affine representation admits two extremal bounds.

D.2 Initial exactness

Sometimes, the solution to a recurrence equation may display very chaotic behaviour for the first few values, which are edge cases, and then adopt a more regular behaviour that we can capture with abstract sequences.

In order to accommodate this, we introduce abstract domains with *initial exactness*, where the first k values – with k fixed – are recorded exactly, and the rest of the sequence is represented by an abstract sequence.

Suppose given a domain A of sequences with concretisation $\phi_A : A \to D$. For $k \in \mathbb{N}^*$, we introduce the domain A_k of *A-abstract sequences with k-initial exactness.* We set

$$A_k \triangleq \mathbb{N}_\infty^k \times A,$$
$$(\overrightarrow{u}, f^\sharp) \sqsubseteq_{A_k} (\overrightarrow{v}, g^\sharp) \overset{\Delta}{\iff} (\forall 0 \leq i < k,\ u_i \leq_{\mathbb{N}_\infty} v_i) \ \wedge\ f_\sharp \sqsubseteq_A g_\sharp,$$

$$\begin{aligned} \phi_{A_k} &: A_k \to D \\ (\overrightarrow{u}, f^\sharp) &\mapsto \left(n \mapsto \begin{cases} u_n & \text{if } n < k, \\ \phi_A(f^\sharp)(n-k) & \text{otherwise.} \end{cases} \right) \end{aligned}$$

The joins and meets can be computed pointwise. A_k is a complete lattice of infinite height, and enjoys the DCC whenever A does. As can be seen from the formula of the height of a cardinal product recalled in [61], the height of A_k is $\sup_{\alpha < h(A)} \omega \cdot k + \alpha + 2$.

Transfer functions All pointwise operations are easy to extend. Similarly, Set_0 becomes trivial. The shifts are not too complicated, but scalings require a bit more work.

The most difficult case is composition, which requires to compute some global inequalities, and where we have some design space.

We define $[\![\cdot]\!]^\sharp_{A_k} : \texttt{Eqs} \to (A_k \to \mathcal{P}(A_k))$ by structural induction, making use of $[\![\cdot]\!]^\sharp_A$.

$$\begin{aligned}
[\![\mathrm{Cst}\ c]\!]^\sharp_{A_k}((\overrightarrow{u}, f^\sharp)) &= \left\{((c, \cdots, c), g^\sharp) \,\middle|\, g^\sharp \in [\![\mathrm{Cst}\ c]\!]^\sharp_A(f^\sharp)\right\} \\
[\![\texttt{n}]\!]^\sharp_{A_k}((\overrightarrow{u}, f^\sharp)) &= \left\{((0, \cdots, k-1), g^\sharp) \,\middle|\, g^\sharp \in [\![\mathrm{Shift}\ (-k)]\!]^\sharp_A \circ [\![\texttt{n}]\!]^\sharp_A(f^\sharp)\right\} \\
[\![\texttt{f}]\!]^\sharp_{A_k}((\overrightarrow{u}, f^\sharp)) &= \{(\overrightarrow{u}, f^\sharp)\} \\
[\![\Diamond]\!]^\sharp_{A_k}((\overrightarrow{u}, g_1^\sharp)),\ (\overrightarrow{v}, g_2^\sharp)) &= \left\{\left(i \mapsto u_i \Diamond v_i,\ [\![\Diamond]\!]^\sharp_A(g_1^\sharp, g_2^\sharp)\right)\right\} \qquad \text{for } \Diamond \in \{+, \times, -\}
\end{aligned}$$

$$[\![\Diamond\ e_1\ e_2]\!]^\sharp_{A_k}(f^\sharp) = \bigcup_{\substack{g_1^\sharp \in [\![e_1]\!]^\sharp_{A_k}(f^\sharp) \\ g_2^\sharp \in [\![e_2]\!]^\sharp_{A_k}(f^\sharp)}} [\![\Diamond]\!]^\sharp_{A_k}(g_1^\sharp, g_2^\sharp), \qquad \text{for } \Diamond \in \{+, \times, \circ\}$$

$$[\![-\ e_1\ e_2]\!]^\sharp_{A_k}(f^\sharp) = \begin{cases} \bigcup_{\substack{g_1^\sharp \in [\![e_1]\!]^\sharp_{A_k}(f^\sharp) \\ g_2^\sharp \in [\![e_2]\!]^\sharp_{A_k}(f^\sharp)}} [\![-]\!]^\sharp_{A_k}(g_1^\sharp, g_2^\sharp), & \begin{array}{l}\text{if we know that the approximation of } e_2 \\ \text{is exact and a singleton,} \\ \text{e.g. if } e_2 = \mathrm{Cst}\ c \text{ or } e_2 = \texttt{n}\end{array} \\ [\![e_1]\!]^\sharp_{A_k}(f^\sharp) & \text{otherwise (or use lower bounds)} \end{cases}$$

Composition is harder, and will be described at the end of this annex.

Now, notice that Set_0 becomes trivial, and that we can reuse previous transfer functions for shifts.

$$\begin{aligned}
[\![\mathrm{Set}_0\ c]\!]^\sharp_{A_k}((\overrightarrow{u}, f^\sharp)) &= \left\{\left((c, u_1, ..., u_{k-1}), f^\sharp\right)\right\} \\
[\![\mathrm{Push}\ c]\!]^\sharp_{A_k}((\overrightarrow{u}, f^\sharp)) &= \left\{\left((c, u_0, ..., u_{k-2}),\ g^\sharp\right) \,\middle|\, g^\sharp \in [\![\mathrm{Push}\ u_{k-1}]\!]^\sharp_A(f^\sharp)\right\} \\
[\![\mathrm{Pop}]\!]^\sharp_{A_k}((\overrightarrow{u}, f^\sharp)) &= \left\{\left((u_1, ..., u_{k-1}, \phi_{A_k}(f^\sharp)(0)),\ g^\sharp\right) \,\middle|\, g^\sharp \in [\![\mathrm{Pop}]\!]^\sharp_A(f^\sharp)\right\} \\
[\![\mathrm{Shift}\ \delta]\!]^\sharp_{A_k}((\overrightarrow{u}, f^\sharp)) &= \begin{cases} ([\![\mathrm{Push}\ 0]\!]^\sharp_{A_k})^\delta((\overrightarrow{u}, f^\sharp)) & \text{if } \delta \geq 0 \\ ([\![\mathrm{Pop}]\!]^\sharp_{A_k})^{|\delta|}((\overrightarrow{u}, f^\sharp)) & \text{if } \delta \leq 0 \end{cases}
\end{aligned}$$

For scalings, we can lookup exact value in the initial part using concretisation, and use combinations of scalings and push/pop to update the abstract part.

$$\begin{aligned}
[\![\mathrm{Mult}_{in}\ c]\!]^\sharp_{A_k}((\overrightarrow{u}, f^\sharp)) &= \left\{\left(i \mapsto \phi_{A_k}((\overrightarrow{u}, f^\sharp))(ci), g^\sharp\right) \,\middle|\, g^\sharp \in [\![\mathrm{Mult}_{in}\ c]\!]^\sharp_A \circ [\![\mathrm{Shift}\ (1-c)k]\!]^\sharp_A(f^\sharp)\right\} \\
[\![\mathrm{Div}_{in}\ c]\!]^\sharp_{A_k}((\overrightarrow{u}, f^\sharp)) &= \left\{\left(i \mapsto \phi_{A_k}\left((\overrightarrow{u}, f^\sharp)\right)\left(\left\lfloor \frac{i}{c} \right\rfloor\right), g^\sharp\right) \,\middle|\, g^\sharp \in [\![\mathrm{Div}_{in}\ c]\!]^\sharp_A(U)\right\} \\
&\quad \text{where } U \triangleq [\![\mathrm{Push}\ u_{\lfloor \frac{k-1}{c} \rfloor + 1}]\!]^\sharp_A \circ \cdots \circ [\![\mathrm{Push}\ u_{k-2}]\!]^\sharp_A \circ [\![\mathrm{Push}\ u_{k-1}]\!]^\sharp_A(f^\sharp)
\end{aligned}$$

Composition To compute composition, we can still use lookups for the initial part. However, the abstract part is bit more complex. Indeed, suppose that we want to find a $h^\sharp$ in order to define

$$[\![\circ]\!]^\sharp_{A_k}((\overrightarrow{u}, g_1^\sharp)),\ (\overrightarrow{v}, g_2^\sharp)) = \left\{\left(i \mapsto \phi_{A_k}(\overrightarrow{u}, g_1^\sharp)(v_i),\ h^\sharp\right)\right\}.$$

Then, $\phi_A(h^\sharp)(n)$ may either have to represent a composition of $\overrightarrow{u}$ and $g_2^\sharp$, or of $g_1^\sharp$ and $g_2^\sharp$, depending of the behaviour of $\phi_A(g_2^\sharp)$.

There are actually three cases.

- It can be shown that $\forall n,\ \phi_A(g_2^\sharp)(n) \geq k$.

  To show this, it is sufficient to check that $\phi_A(g_2^\sharp)(0) \geq k$.

  In this case, we can set
$$h^\sharp = [\![\circ]\!]^\sharp_A(g_1^\sharp,\ [\![-\ \texttt{f}\ (\mathrm{Cst}\ k)]\!]^\sharp_A(g_2^\sharp)).$$

Note that we chose minus because we have in mind the example of affine sequences where this is more precise and efficient than a positive shift, but we might imagine domains in which it is better to do a positive shit on $g_1^\sharp$.

- At the other extreme, it can be shown that $\forall n,\ \phi_A(g_2^\sharp)(n) \leq k-1$.

  To show this, it is sufficient to check that $\phi_A(g_2^\sharp)(\infty) \leq k-1$.

  In this can, we need to compute a sound abstraction of $n \mapsto u_{\phi_A(g_2^\sharp)}$.

  One easy choice, which is for the moment precise enough for us, is to compute $\max \phi_A(g_2^\sharp)$ as $\phi_A(g_2^\sharp)(\infty)$, and set

$$\begin{aligned} h^\sharp &= \left[\!\!\left[\text{Cst } \left(\max_{0 \leq i \leq \phi_A(g_2^\sharp)(\infty)} u_i\right)\right]\!\!\right]_A^\sharp (g_2^\sharp) \\ &= [\![\text{Cst } (u_{\phi_A(g_2^\sharp)(\infty)})]\!]_A^\sharp (g_2^\sharp) \qquad \text{when } u \text{ is forced to be increasing} \end{aligned}$$

- The harder case is when $\phi_A(g_2^\sharp)$ is first smaller than k, then bigger than k.

  ○ Let $\overline{n_{crit}}$ be a finite *overapproximation* of the smallest n_{crit} such that

$$\phi_A(g_2^\sharp)(n_{crit}) \geq k, \ \text{ i.e.; } \forall n \geq \overline{n_{crit}},\ \phi_A(g_2^\sharp)(n) \geq k.$$

  We can choose to set

$$\begin{aligned} h^\sharp = &[\![\text{Push } (\phi_{A_k}((\overrightarrow{u}, g_1^\sharp))(\phi_A(g_2^\sharp)(0)))]\!]_A^\sharp \circ ... \circ [\![\text{Push } (\phi_{A_k}((\overrightarrow{u}, g_1^\sharp))(\phi_A(g_2^\sharp)(\overline{n_{crit}}-1)))]\!]_A^\sharp \\ &\circ [\![\circ]\!]_A^\sharp \Big(g_1^\sharp, [\![- \texttt{f} \text{ (Cst } k)]\!]_A^\sharp \circ [\![\text{Shift } (-\overline{n_{crit}})]\!]_A^\sharp (g_2^\sharp)\Big). \end{aligned}$$

  Note that to do this, we need to be able to compute such a n_0. In the case of affine sequences, this can be computed efficiently (and exactly).

  Moreover, for large n_0, this operation may become very costly.

  ○ If this is not acceptable, another possibility is to look for an *underapproximation* $\underline{n_{crit}}$ of n_{crit}, i.e. such that

$$\forall n < n_0,\ \phi_A(g_2^\sharp)(n) \leq k-1.$$

  For this, a trivial underapproximation can always be computed.

  With this, we can define

$$\widetilde{g_1}^\sharp = [\![\text{Push } u_0]\!]_A \circ ... \circ [\![\text{Push } u_{k-1}]\!]_A (g_1^\sharp),$$

  and choose

$$\begin{aligned} h^\sharp = [\![\text{Push } (u_{\phi_A(g_2^\sharp)(0)})]\!]_A^\sharp \circ ... \circ [\![\text{Push } (u_{\phi_A(g_2^\sharp)(\underline{n_{crit}}-1)})]\!]_A^\sharp \\ \circ [\![\text{Shift } (-\underline{n_{crit}})]\!]_A^\sharp \circ [\![\circ]\!]_A^\sharp \Big(\widetilde{g_1}^\sharp, g_2^\sharp\Big). \end{aligned}$$

  Note that less precise $\widetilde{g_1}^\sharp$ may be computed if efficiency is key.

  If we choose the trivial underapproximation $\underline{n_{crit}} = 0$, this simply gives

$$h^\sharp = [\![\circ]\!]_A^\sharp \Big(\widetilde{g_1}^\sharp, g_2^\sharp\Big).$$