Formal Verification of Resource Usage

Ana Carolina Ferreira da Silva
Mestrado em Ciência de Computadores
Departamento de Ciência de Computadores
2022

Orientador
Mário Florido, Professor Associado, Faculdade de Ciências da Universidade do Porto

Coorientador
Manuel Barbosa, Professor Associado, Faculdade de Ciências da Universidade do Porto

U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, ______/______/_________

Declaração de Honra

Eu, Ana Carolina Ferreira da Silva, inscrito(a) no Mestrado em Ciência de Computadores da Faculdade de Ciências da Universidade do Porto declaro, nos termos do disposto na alínea a) do artigo 14.º do Código Ético de Conduta Académica da U.Porto, que o conteúdo da presente dissertação reflete as perspetivas, o trabalho de investigação e as minhas interpretações no momento da sua entrega.

Ao entregar esta dissertação, declaro, ainda, que a mesma é resultado do meu próprio trabalho de investigação e contém contributos que não foram utilizados previamente noutros trabalhos apresentados a esta ou outra instituição.

Mais declaro que todas as referências a outros autores respeitam escrupulosamente as regras da atribuição, encontrando-se devidamente citadas no corpo do texto e identificadas na secção de referências bibliográficas. Não são divulgados na presente dissertação quaisquer conteúdos cuja reprodução esteja vedada por direitos de autor.

Tenho consciência de que a prática de plágio e auto-plágio constitui um ilícito académico.

Ana Carolina Silva

25 Janeiro de 2023

Acknowledgments

I'm incredibly grateful to my supervisors, Mário Florido and Manuel Barbosa, for their continuous support during the development of this dissertation. Their encouragement made the work that much more enjoyable and helped me keep going, even in moments of self-doubt. I always left our weekly meetings feeling happier and more relaxed than when they started.

I would also like to thank my family and friends for always believing in me and for the support they've shown during the past years.

Abstract

This thesis presents a proof system for reasoning about execution time bounds for a core imperative programming language. We define an axiomatic semantics with assertions on properties related to execution time. Proof systems are defined for three different scenarios: approximations of the worst-case execution time, exact time reasoning, and less pessimistic execution time estimation using amortized analysis. We define a Hoare logic for the three cases and prove its soundness with respect to an annotated cost-aware operational semantics. Finally, we define a verification conditions (VC) generator that generates the goals needed to prove program correctness, cost, and termination. Those goals are then sent to the Easycrypt toolset for validation. The practicality of the proof system is demonstrated with an implementation in OCaml of the different modules needed to apply it to example programs. Our case studies are motivated by real-time and cryptographic software.

Resumo

Esta tese apresenta um sistema de prova para raciocinar sobre limites de tempo de execução para uma linguagem de programação imperativa. Definimos uma semântica axiomática com asserções sobre propriedades relacionadas com o tempo de execução. Os sistemas de prova são definidos para três cenários diferentes: aproximações do tempo de execução para o pior caso, cálculo de tempo exato e estimativa de tempo de execução menos pessimista usando análise amortizada. Definimos uma lógica de Hoare para os três casos e provamos sua correção em relação a uma semântica operacional com capacidade para cálculo de custos de execução. Por fim, definimos um gerador de condições de verificação (VC) que gera as condições necessárias para provar a correção, custo e terminação do programa. Estas condições são enviadas para uma ferramente de provas, como o EasyCrypt, para validação. A praticalidade do sistema de prova é demonstrada com uma implementação em OCaml dos diferentes módulos necessários para aplicá-lo a programas exemplo. Os nossos casos de estudo são motivados por software criptográfico e de tempo real.

Contents

List of Tables

List of Figures

List of Code

Acronyms

AST	Abstract Syntax Tree
wp	Weakest Precondition
wpc	Weakest Precondition-Cost
VC	Verification Condition
VCG	Verification Condition Generator

Chapter 1

Introduction

Semantics-based approaches to program verification usually belong to two different broad classes: 1) partial correctness assertions expressing relations between the initial and final state of program variables in the form of pre and postconditions, assuming that the program terminates; and 2) total correctness properties, which besides those assertions which specify claims about program behavior, also express program termination.

However, another class of properties is fast growing in relevance as a target for program verification: resource consumption when executing a program. The term *resource* is used broadly: resources can be time used to execute the program on a particular architecture, memory used (stack or heap) during program execution, or even energy consumption. Resource consumption has a significant impact in different specific areas, such as real-time systems, critical systems relying on limited power sources, and the analysis of timing side-channels in cryptographic software.

A proof system for total correctness can be used to prove that a program execution terminates, but it does not give any information about the resources it needs to terminate. In this dissertation, we want to study extended proof systems for proving assertions about program behavior that may refer to the required resources and, in particular, to the execution time.

Proof systems to prove bounds on the execution time of program execution were defined before in [5, 22]. Inspired by the work presented in [5] our goal is to study inference systems that allow proving assertions of the form $\{\varphi\}C\{\psi|t\}$, meaning that if the execution of the statement C is started in a state that validates the precondition φ then it terminates in a state that validates postcondition ψ and the required execution time is at most of magnitude t.

1.1 Goals and Contributions

Our main goal is to define an axiomatic semantics-based proof system for reasoning about execution time bounds for a core imperative programming language. Such a system would be useful not only to understand the resource necessities of a program but also it could be applied to cryptographic implementations to prove the independence of resource usage from certain program variables. This high-level goal translates into the following concrete objectives:

1. Study axiomatic systems. This will further our knowledge in the field and allow us to understand how to define our logic for resource analysis.
2. Study amortized analysis so we can understand how to apply amortization to a proof system to refine cost-bound estimation of while loops.
3. Analyze the state-of-the-art to understand what has already been developed to analyze resource consumption and the main limitations found.
4. Develop a sound logic capable of verifying correction, terminations, and bounds on resource consumption using a simple imperative programming language.
5. Create a tool based on our logic, capable of verifying time bounds, correction, and terminations, for example-problems.
6. Apply this logic to analyze the time complexity of classic algorithms.

The main contribution of this paper is a proof system that can verify resource assumptions in three different scenarios:

1. Upper bounds on the required execution time. This is mostly an adaptation of previous work in [5].
2. Amortized costs denoting less pessimistic bounds on the execution time.
3. Exact costs for a fragment of the initial language with bounded recursion and a constrained form of conditional statements.

The two last scenarios are a novel contribution of our system, and we treat them in a unified way to enable their integrated use.

Assertions on program behavior that establish upper bounds on execution time may be useful for general programming, where one wants to prove safety conditions concerning the worse case program complexity. As in prior approaches, the tightness of the bound is not captured by the logic, and there is often a trade-off between the tightness of the proved bound and the required proof effort.

Proofs that leverage amortized costs may be used when trivially composing worst-case run-time bounds results in overly pessimistic analyses. This is particularly useful for algorithms where some components imply a significant cost in resources, whereas other components are not as costly. With amortized costs, we may prove assertions about the aggregate use of costly and less costly operations over the whole algorithm execution.

Finally, the third class of assertions denoting exact costs are useful in scenarios where the approximation of execution time is not enough to guarantee safety, as it happens for critical systems and real-time programming. Moreover, proving that the exact execution time of a program is an expression that does not depend on confidential data provides a direct way to prove the absence of timing leakage, which is relevant in cryptographic implementations. We must restrict the programming language to guarantee the ability to prove exact costs. Thus, in this third scenario, programs have bound recursion, and conditional statement branches have to have the same cost.

Before defining our proof system, we defined an operational semantics capable of computing the execution time for expressions and statements during program execution. This cost-aware operational semantics is another contribution of our work, and it is used to prove the soundness of our inference system.

A third contribution of this work, which shows the practicality of our proof system, is an implementation in OCaml of the different modules needed to apply it to example programs. We then present several application examples motivated by real-time and cryptographic software.

1.2 Thesis Structure

This thesis is organized as follows:

- The first chapter - **Introduction** - gives a context of our work in the field, the motivation for this project, our main goals, our contributions, and how the document is organized.
- The second chapter - **Background** - elaborates on the theoretical results used in the basis of our work and needed to understand our definitions and results.
- The third chapter - **Related Work** - presents an analysis of the literature on static resource analysis, from type-based systems to axiomatic semantics systems.
- The fourth chapter - **Cost Aware Program Logic** - presents our language definition, our original logic for upper bound estimation, the respective Verification Condition Generator (VCG), and some illustrative examples.
- The fifth chapter - **Amortized Costs** - briefly introduces the field of amortized analysis and presents an extension to the logic and VCG from chapter 4, with the use of amortized analysis to improve the upper-bound estimation.
- The sixth chapter - **Exact Logic** - presents a variation to our language and an extension to our logic that allows for the derivation of the exact cost of a program.
- The seventh chapter - **Implementation and Experimental Results** - shows the architecture of the tool developed, as well as some implementation details and practical results.

- The eighth chapter - **Conclusion and Future Work** - reflects on the main conclusions from our research and developed work and presents some goals to further extend and improve our project.

Chapter 2

Background

In this chapter, we provide an overview of the field of formal verification and some of the most relevant theoretical results that are the basis of our work. We start by giving historical background on the field of formal verification. Here we will present results, such as the ones achieved by Floyd and Hoare, used as the base of our definitions. We will also define concepts fundamental to understanding the work presented in this dissertation.

2.1 Historical Background

As computers became more powerful, programs also became longer and more complex. When programs were still relatively small, flowcharts or extensive testing was enough to prove a program's functionality. But programs quickly started being so complex that these methods became unreliable and more prone to error.

At the beginning of the second half of the 20th-century, experts started to find vulnerabilities in public distributed software. Since then, the use of computational systems has grown exponentially, and so did the number of vulnerabilities and their impact. A simple error might have drastic consequences, such as a leak of confidential information, the crash of critical systems, and direct loss of assets.

This problem proved to be enough reason to start thinking about a more reliable way to guarantee the properties of a program and develop tools that help verify these properties.

Formal Verification refers to using mathematical principles to prove the correction of a given specification of a program. It is hard to pinpoint where it all started, but the works of Robert Floyd [11] and Tony Hoare [14] were undoubtedly pioneers in the field, and their definitions are still the base of verification tools used today.

2.2 Semantics

When defining programming languages, we want a way to be capable of reasoning about what programs are doing. The syntax describes the grammatical rules we must follow to write a program in a language. The syntax allows us to distinguish between languages and identify a program's language. But if we want to understand what that program is doing, we need to look at its semantics. Semantics is a way to make sense of the meaning of a program and understand what it is trying to accomplish.

There are multiple strategies to analyze the meaning of a program. The most popular ones are operational, denotational, and axiomatic semantics. Operational semantics focus on what steps we take during the program's execution. In denotational semantics, we do not care about the "how" but only about "what" the program is doing. In axiomatic semantics, we are concerned about evaluating the satisfability of assertions on the program and its variables. We will go more in-depth on how operational and axiomatic semantics work.

2.2.1 Operational Semantics

Operational semantics describes the meaning of a program by specifying the transitions between states of an abstract state machine. As we mentioned, unlike with denotational semantics, here we are concerned about *how* the machine changes states with the execution of a statement.

There are two styles of operational semantics

- Small-step or Structural Operational Semantics
- Big-step or Natural Semantics

In Structural Operational Semantics or Small-step Semantics, we are concerned about every individual transition we take throughout the program's execution. In Natural Semantics or Big-step semantics, we want to understand how we transition from the initial to the final state. We are concerned about a high-level analysis of how the machine state changes and not about each individual step.

2.2.2 Axiomatic Semantics (Hoare Logic)

In 1967 Floyd specified a method that would allow proving properties on programs, such as correctness, equivalence, and termination [11]. They achieved this by representing programs as flowcharts and associating propositions to each connection on the flowchart. The proof is done by induction on the number of steps. If an instruction is reached by a connection whose proposition is true, then we must leave it with a true condition as well. In 1969 Hoare wrote a paper where they extended Floyd's logic to prove properties on a simple imperative program [14]. In this paper, they defined what we now call Hoare (or Floyd-Hoare) triples.

Definition 1 (Hoare Triples)**.** *A Hoare triple is represented as*

$$\{P\}Q\{R\}$$

and can be interpreted as "if the assertion P is true before we run program Q, then assertion R will be true when the program ends".

Notice this definition does not offer guarantees over termination. Executing program Q from a state validating P, does not have to halt. As long as whenever it does the final state validates R. We call assertions in the form $\{P\}Q\{R\}$ *partial correctness assertions.*

Using this definition Hoare specifies a proof system with a set of axioms and inference rules, which allow to prove assertions on any program written in this language. A derivation on this proof system is called a *theorem* and is written as $\vdash \{P\}Q\{R\}$.

As an example, let us look at the assignment axiom. Consider an assignment to variable x of an expression a in the form

$$x := a$$

If an assertion P is true after executing the assignment (when variable x takes the value of expression a) then it has to be true before the assignment if we replace any mentions of x in P by a. This is usually represented as $P[a/x]$. Therefore the assignment axiom is written as

$$\{P[a/x]\}\ x := a\ \{P\}$$

If, in addition to proving a program specification is correct, we also want to prove the program always halts, we are looking for Total Correctness.

Definition 2 (Total Correctness)**.** *A total correctness assertion is represented as*

$$[P]Q[R]$$

where P and R are assertions and Q is a program. If we execute Q from a state that satisfies P program Q will terminate and the final state will satisfy R.

$$\text{partial correctness} + \text{termination} = \text{total correctness}$$

Total correctness is harder to prove than partial correctness and not always possible. But it also gives a stronger guarantee about a programs behavior.

We consider two properties on this proof system, soundness an completeness. Soundness ensures our proof systems generates valid partial correctness assertions. Completeness ensures that our system is capable of deriving every valid assertion.

Definition 3 (Validity). *We say an assertion {P}Q{R} is valid if it is true for all possible states.*

$$\forall \sigma \in \Sigma_{\perp}.\ \sigma \models \{P\}Q\{R\}$$

Or we can simply represent it as

$$\models \{P\}Q\{R\}$$

Definition 4 (Soundness). *Our proof system is sound if every rule preserves validity. In other words, for any partial correctness assertion* $\{P\}Q\{R\}$

$$\text{if } \vdash \{P\}Q\{R\} \text{ then } \models \{P\}Q\{R\}$$

Proof. Soundness is proved by structural induction on the statement Q. □

Definition 5 (Completeness). *A proof system is complete if every true assertion* $\{P\}Q\{R\}$ *can be proved by our system.*

$$\text{if } \models \{P\}Q\{R\} \text{ then } \vdash \{P\}Q\{R\}$$

Proving completeness is not as trivial and most of the times not possible. The completeness of the proof system presented by Hoare was established by Cook in 1978 [9]. In this paper he presents a proof of *relative completeness*, that is, assuming our assertion language is complete then the logic presented by Hoare is also complete.

2.3 Verification Conditions Generator

We now have the necessary notation to specify the behavior of a program. Hoare's set of axioms and rules allows proving that program's said behavior. Manually proving these properties is not only long and tedious but also prone to error. We are missing a mechanized solution we could apply to any program to guarantee its validity.

Dijkstra defined the weakest precondition algorithm in is 1976 book "A Discipline of Programming" [10].

Definition 6 (Weakest Precondition). *The weakest precondition is the simplest condition, necessary and sufficient to guarantee the post-condition is true when the program terminates. We use the notation* $wp(Q, R)$ *where* Q *is a statement and* R *is a post-condition.*

$$\models \{P\}Q\{R\} \textit{ iff } P \rightarrow wp(Q, R)$$

Let us consider as an example the statement $Q \equiv x := y + 2$ and we want to prove $R \equiv x \geq 0$ is true after executing Q. The weakest precondition $wp(Q, R)$ would say that as long as $y \geq -2$ is true before executing statement Q, then R will be true after execution.

In 1979, JC King presents the first mechanized algorithm to automatically verify the correctness of a program [19]. His work was based on the definitions provided by Floyd [11] and

Manna [21]. In more recent implementations this algorithm is usually based on Hoare's definition of correctness and it is called a Verification Condition Generator (VCG). This algorithm makes use of the weakest precondition function. Given a Hoare triple $\{P\}Q\{R\}$, for the program Q to be correct we must guarantee that $P \rightarrow wp(Q, R)$ is true. This is the condition that needs to be proved in order to guarantee correctness. If our language contains loops we need to satisfy extra conditions, including the preservation of the loop invariant. The *loop invariant* is an assertion that is satisfied before and after every execution of the loop's body.

Definition 7 (Verification Condition Generator). *A VCG is an algorithm that when applied to a Hoare triple returns a set of Verification Condition (VC). The Hoare triple is derivable in our proof system, if and only if all the generated conditions are valid.*

$$\models VCG(\{P\}Q\{R\}) \text{ iff } \vdash \{P\}Q\{R\}$$

Theorem 1 (Soundness of VCG). $\models VCG(\{P\}Q\{R\}) \implies \vdash \{P\}Q\{R\}$.

Proof. By induction on the structure of Q. □

Theorem 2 (Completness of VCG). $\vdash \{P\}Q\{R\} \implies \models VCG(\{P\}Q\{R\})$.

Proof. By induction in the derivation of $\vdash \{P\}Q\{R\}$. □

Returning to our last example where $wp(x := y + 2, x \geq 0) = y \geq -2$ let us consider the partial correctness assertion $\{y = 0\}x := y + 2\{x \geq 0\}$. Since in this case the program does not contain any loops there is only one verification condition that needs to be satisfied in order to prove correctness: $(y = 0) \rightarrow (y \geq -2)$. Since this VC is valid, we know our program is correct.

Chapter 3

Related Work

There has been increasing interest in the field of static resource analysis. Knowing bounds on resources or rough estimates of resource consumption can help us optimize embedded software and real-time systems.

This chapter briefly describes some of the most relevant work on resource estimation. We divide the chapter into sections according to the methodology used to prove or infer resource bounds. We also compare the literature with the work presented in this dissertation and show the relevance of the work we developed.

3.1 Axiomatic Semantics Systems

One way of proving bounds on a program 's resources is by using axiomatic semantics. Other works have already implemented systems that use axiomatic semantics for resource analysis, which differ from our work in multiple ways, from the paradigm of language used to the precision of the derived bounds. This section explains some of the most relevant work in the literature that inspired our definitions. We further subdivided this literature into two categories: classical cost analysis and amortized cost analysis.

3.1.1 Classical Cost Analysis

There is already some work using derivations on the logic presented by Hoare [14] for resource analysis. Some estimate orders of magnitude, while others use a more detailed annotation to automate the process but lack ways of optimizing the bounds.

One of the first and still one of the most relevant works on this topic is the one presented by Nielsen [22, 23]. The author defines an axiomatic semantics for a simple imperative programming language in this work and extends Hoare's logic so that the proof system would be capable of proving the magnitude of worst-case running time and termination. This system is also proven sound. Even though this work operates on a similar imperative language to the one we defined, it lacks the precision our logic provides since it only allows proving order-of-magnitude.

In 2014, Carbonneaux *et al.* [7] presented a system that verifies stack-space bounds of compiled machine code (x86 assembly) at the C level. That is, it derives bounds during compilation from C to assembly. They developed a quantitative Hoare logic capable of reasoning about resource consumption. This work is an extension of the CompCert C Compiler [20]. Coq was used to implement and verify the compiler. The work by Carbonneaux focuses on the compilation from C to assembly using quantitative logic, which does not serve the same purpose we are trying to achieve. With our work, we can prove tight bounds on imperative programs using an assisted proof system, where the user can help make the bounds as precise as possible. Also, how default constructors' costs are defined makes our work easy to adjust for a system with different resource usage.

In 2018, Kaminski *et al.* defined a conservative extension of Nielsen's logic for deterministic programs [22, 23] by developing a Weakest Precondition calculus for expected runtime on probabilistic programs [18]. Again this work is largely automated, which differs from our user-assisted approach. Since it reasons about probabilistic programs, it faces other challenges than the ones we are interested in this work.

In 2021 a paper was released extending the logic of EasyCrypt [5]. EasyCrypt is an interactive tool created to prove the security of cryptographic implementations. One of its core concepts is a set of Hoare Logics, which allow proofs on relational procedures and probabilistic implementations. In this paper, the authors propose an extension to the EasyCrypt tool, allowing to prove properties on the cost of a program. To achieve this, they extended the existing logic to include cost rules. They also implemented a way to define the cost of custom operators. Our work operates on a subset of EasyCrypt's language, but we extended the logic to use the potential method of amortized analysis, increasing the accuracy of the generated bounds.

3.1.2 Amortized Cost Analysis

We will now present some of the literature that, in addition to using axiomatic semantics for static cost analysis, also uses amortized analysis to increase the accuracy of the bounds.

Carbonneaux *et al.* [8] continued their previous work [7] on deriving worst-case resource bounds for C programs, but they now implemented a system that uses amortized analysis and abstract interpretations.

In [13], Haslbeck and Nipkow analyze the works of Nielson[23], Carbonneaux *et al.* [7, 8] and Atkey [3] and prove the soundness of their systems. In this paper, they implement Verification Condition Generators based on Nielsen's logic and Carbonneaux's Logic, proving it sound and complete. They compare all three methodologies and explain some of the limitations of these systems.

3.2 Type-Based Systems

While our system uses a Hoare logic to prove upper bounds on program cost, many existing systems are type-based. Usually, these systems use type inference and type size/cost annotation in order to be able to analyze resource usage statically. Even though these works highly differ from ours, we will briefly mention some of the most relevant works in the field.

In [24], the authors present a proof system for cost analysis on functional programs using a fine-grained program logic for verifying relational and unary costs of higher-order programs. The paper [4] presents a fully automated methodology for complexity analysis of higher-order functional programs based on a type system for size analysis with a sound type inference procedure. Hoffman and Jost, [17] defined a type system capable of analyzing heap space requirements during compilation time based on amortized analysis. This work was limited to linear bounds in the size of the input, so they later extended it to polynomial resource bounds [15].

In 2017, Hoffman *et al.* [16] developed a resource analysis system capable of proving worst-case resource bounds. This resource is a user-defined input and can be anything from time or memory to energy usage. This work is an extension of their previous work in Automatic Amortized Resource Analysis (AARA), where they used amortized analysis to derive polynomial bounds for the first time. Their proof system is a type system with inductive type, refined from OCaml's type system.

Atkey [3] presents a type-based amortized resource analysis system adapted from Hofmann's work to imperative pointer-manipulating languages. They achieve this by implementing a separation logic extension to reason about resource analysis.

Serrano *et al.* [25] introduced a general resource analysis for logic programs based on sized types, i.e., types that contain structural information and lower and upper bounds on the size of the terms. They achieved this by using an abstract Interpretation Framework.

In [26], the authors develop a type-based proof system capable of automatically and statically analyzing heap allocations. This work is an extension of Hoffman's work for a lazy setting. Vasconcelos *et al.* [28] defined a type system capable of predicting upper bounds on the cost of memory allocation for co-recursive definitions in a lazy functional language.

3.3 Other Proof Systems

Some other works on static estimation of resource bounds use other methodologies other than axiomatic semantics or type theory. In a 2009 paper, the tool COSTA is presented [1]. COSTA is a static analyzer for Java bytecode. It infers cost and termination information. It takes a cost model for a user-defined resource as input and obtains an upper bound on the execution time of this resource.

In [12] they compute symbolic bounds on the number of statements a procedure executes in terms of its input. They use the notion of counter variables and an invariant generation tool to compute linear bounds on these counters. These bounds are then composed to generate a total non-linear bound.

Brockschmidt *et al.* [6] uses Polynomial Rank Functions (PRF) to compute time bounds. Then these bounds are used to infer size bounds on program variables. They consider small parts of the program in each step and incrementally improve the bound approximation.

Chapter 4

Cost Aware Program Logic

In this chapter, we will focus on our first goal of formally verifying the worst-case execution time of imperative programs. To achieve this, we specify a simple while language with annotations. We first define an operational semantics capable of computing the execution time. Having this, we define a cost logic, which we use to prove correctness, termination, and worst-case execution time. In the last section of this chapter, we present a Verification Condition Generator (VCG) algorithm for our logic. Practical results using this algorithm definition are shown in chapter 7.

4.1 Annotated While Language

We start by defining a core imperative language (**IMP**) with the following syntactic structures: numbers, booleans, identifiers, arithmetic expressions, boolean expressions, statements, and assertions. To simplify the presentation and explanation of these structures we will use meta-variables to refer to elements in these sets: n for numbers, x for identifiers, a for arithmetic expressions, b for boolean expressions, S for statements, and P, Q for assertions.

Numbers and Booleans We consider numbers, to be the usual set of signed decimal numerals for positive and negative integer numbers. Our boolean set is defined as $\{true, false\}$.

Arithmetic Expressions Arithmetic expressions are defined by the following rules

$$a ::= n \mid x \mid x[a] \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1/a_2 \mid {a_1}^{a_2} \mid \sum_{x=n}^{a_1} a_2$$

Boolean Expressions Similarly to arithmetic expressions, booleans expressions are defined as

$$b ::= true \mid false \mid b_1 = b_2 \mid b_1 \neq b_2 \mid b_1 < b_2 \mid b_1 > b_2 \mid b_1 \leq b_2 \mid b_1 \geq b_2 \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2$$

Statements Finally, we define the following statement rules:

$$S ::= \textbf{skip} \mid x = a \mid x[a_1] = a_2 \mid \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ done} \mid \textbf{while } b \textbf{ do } S \textbf{ done} \mid S_1; S_2$$

An example of a simple program in our language is shown in figure 4.1, where we note the time annotation on the right-hand side of the post-condition. For now, let us ignore the annotations inside {}, they will be explained later in section 4.3.

$\{r = x \ \wedge \ q = 0 \ \wedge \ y > 0 \ \wedge \ x \geq 0\}$

```
while y <= r do
   r = r - y;
   q = q + 1
end
```

$\{x = r + y \times q \ \wedge \ r < y \mid 20x + 5\}$

Figure 4.1: Division algorithm implemented in IMP.

4.2 Operational Semantics

Now that we have defined the syntax of our language we need to set semantic rules that will give the meaning of a program. In order to achieve this, and since our language has variable declarations, we first need to define the notion of *state*. A state can be defined as a function that given a variable will return its value.

$$State : var \rightarrow int$$

A variable is defined as either an identifier or a position in an array.

$$var ::= x \mid x[n]$$

Thus, writing $\sigma\ x$ will specify the value of variable x in state σ.

4.2.1 Semantic of Expressions

In order to evaluate arithmetic expressions, we define a semantic function $\mathcal{A}$ which will receive two arguments, an arithmetic expression and a state.

$$\mathcal{A} : aexp \rightarrow state \rightarrow int$$

Writing $\mathcal{A}[\![a]\!]\sigma$ will return the value of evaluating expression a in state σ. The function is defined in figure 4.2.

Similarly, we define a semantic function $\mathcal{B}$ that, given a state, will convert a boolean expression to truth values.

$$\mathcal{B} : bexp \rightarrow state \rightarrow bool$$

In figure 4.3 we define $\mathcal{B}$ using the previous definition of $\mathcal{A}$.

$$
\begin{aligned}
\mathcal{A}\llbracket n \rrbracket\sigma &= n\\
\mathcal{A}\llbracket x \rrbracket\sigma &= \sigma\ x\\
\mathcal{A}\llbracket x[a] \rrbracket\sigma &= \sigma\ x[\mathcal{A}\llbracket a \rrbracket\sigma]\\
\mathcal{A}\llbracket a_1 + a_2 \rrbracket\sigma &= \mathcal{A}\llbracket a_1 \rrbracket + \mathcal{A}\llbracket a_2 \rrbracket\\
\mathcal{A}\llbracket a_1 * a_2 \rrbracket\sigma &= \mathcal{A}\llbracket a_1 \rrbracket * \mathcal{A}\llbracket a_2 \rrbracket\\
\mathcal{A}\llbracket a_1 - a_2 \rrbracket\sigma &= \mathcal{A}\llbracket a_1 \rrbracket - \mathcal{A}\llbracket a_2 \rrbracket\\
\mathcal{A}\llbracket a_1 / a_2 \rrbracket\sigma &= \mathcal{A}\llbracket a_1 \rrbracket / \mathcal{A}\llbracket a_2 \rrbracket\\
\mathcal{A}\llbracket {a_1}^{a_2} \rrbracket\sigma &= \mathcal{A}\llbracket a_1 \rrbracket^{\mathcal{A}\llbracket a_2 \rrbracket}\\
\mathcal{A}\llbracket \sum_{x=n}^{a_1} a_2 \rrbracket\sigma &= \sum_{x=n}^{\mathcal{A}\llbracket a_1 \rrbracket} \mathcal{A}\llbracket a_2 \rrbracket
\end{aligned}
$$

Figure 4.2: Semantics of arithmetic expressions.

$$
\begin{aligned}
\mathcal{B}\llbracket \mathrm{true} \rrbracket\sigma &\quad=\quad \mathbf{true}\\
\mathcal{B}\llbracket \mathrm{false} \rrbracket\sigma &\quad=\quad \mathbf{false}\\
\mathcal{B}\llbracket a_1 = a_2 \rrbracket\sigma &\quad=\quad \mathcal{A}\llbracket a_1 \rrbracket\sigma = \mathcal{A}\llbracket a_2 \rrbracket\sigma\\
\mathcal{B}\llbracket a_1 \neq a_2 \rrbracket\sigma &\quad=\quad \mathcal{A}\llbracket a_1 \rrbracket\sigma \neq \mathcal{A}\llbracket a_2 \rrbracket\sigma\\
\mathcal{B}\llbracket a_1 < a_2 \rrbracket\sigma &\quad=\quad \mathcal{A}\llbracket a_1 \rrbracket\sigma < \mathcal{A}\llbracket a_2 \rrbracket\sigma\\
\mathcal{B}\llbracket a_1 > a_2 \rrbracket\sigma &\quad=\quad \mathcal{A}\llbracket a_1 \rrbracket\sigma > \mathcal{A}\llbracket a_2 \rrbracket\sigma\\
\mathcal{B}\llbracket a_1 \leq a_2 \rrbracket\sigma &\quad=\quad \mathcal{A}\llbracket a_1 \rrbracket\sigma \leq \mathcal{A}\llbracket a_2 \rrbracket\sigma\\
\mathcal{B}\llbracket a_1 \geq a_2 \rrbracket\sigma &\quad=\quad \mathcal{A}\llbracket a_1 \rrbracket\sigma \geq \mathcal{A}\llbracket a_2 \rrbracket\sigma\\
\mathcal{B}\llbracket \neg a \rrbracket\sigma &\quad=\quad \neg\mathcal{B}\llbracket a \rrbracket\sigma\\
\mathcal{B}\llbracket a_1 \wedge a_2 \rrbracket\sigma &\quad=\quad \mathcal{B}\llbracket a_1 \rrbracket\sigma \wedge \mathcal{B}\llbracket a_2 \rrbracket\sigma\\
\mathcal{B}\llbracket a_1 \vee a_2 \rrbracket\sigma &\quad=\quad \mathcal{B}\llbracket a_1 \rrbracket\sigma \vee \mathcal{B}\llbracket a_2 \rrbracket\sigma
\end{aligned}
$$

Figure 4.3: Semantics of boolean expressions.

4.2.2 Cost of expressions

Our semantics will not only evaluate the meaning of a program but also compute the exact cost of executing it. To achieve this, we start by defining semantics for the cost of evaluating arithmetic and boolean expressions, as shown in Figure 4.4. To evaluate the cost of an expression, we must establish the cost of atomic operations in our language, such as reading from memory or performing basic arithmetic (addition, multiplication, etc) and logic operations (disjunction, negation, etc). For example, C_{CST} corresponds to the cost of evaluating a constant, and C_{VAR} is the cost of evaluating a variable. The cost of evaluating a multiplication $\mathcal{TA}\llbracket a_1 * a_2 \rrbracket$ is defined as a sum of the cost of evaluating each of the arithmetic expressions, a_1 and a_2, plus the cost of the multiplication operation C_*. The cost of evaluating a sum $\sum_{i=b}^{e} a$ is the cost of evaluating a multiplied by the number of times we evaluate it $e - b$. These rules are simultaneously used by our operational semantics when executing our program, and by our axiomatic semantics, when

$$
\begin{aligned}
&\mathcal{TA}[\![\ n\ ]\!] && = C_{CST}\\
&\mathcal{TA}[\![\ x\ ]\!] && = C_{VAR}\\
&\mathcal{TA}[\![\ x[a]\ ]\!] && = \mathcal{TA}[\![\ a\ ]\!] + C_{ARRAY}\\
&\mathcal{TA}[\![\ a_1 + a_2\ ]\!] && = \mathcal{TA}[\![\ a_1\ ]\!] + \mathcal{TA}[\![\ a_2\ ]\!] + C_{+}\\
&\mathcal{TA}[\![a_1 * a_2]\!] && = \mathcal{TA}[\![\ a_1\ ]\!] + \mathcal{TA}[\![\ a_2\ ]\!] + C_{*}\\
&\mathcal{TA}[\![\ a_1 - a_2\ ]\!] && = \mathcal{TA}[\![\ a_1\ ]\!] + \mathcal{TA}[\![\ a_2\ ]\!] + C_{-}\\
&\mathcal{TA}[\![\ \sum_{i=b}^{e} a]\!] && = (e - b) \times \mathcal{T}[\![\ a\ ]\!]
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{TB}[\![\ \text{true}\ ]\!] && = C_{BOOL}\\
&\mathcal{TB}[\![\ \text{false}\ ]\!] && = C_{BOOL}\\
&\mathcal{TB}[\![\ \mathrm{a}_1 = \mathrm{a}_2\ ]\!] && = \mathcal{TA}[\![\ a_1\ ]\!] + \mathcal{TA}[\![\ a_2\ ]\!] + C_{=}\\
&\mathcal{TB}[\![\ \mathrm{a}_1 \leq \mathrm{a}_2\ ]\!] && = \mathcal{TA}[\![\ a_1\ ]\!] + \mathcal{TA}[\![\ a_2\ ]\!] + C_{\leq}\\
&\mathcal{TB}[\![\ \neg \mathrm{b}\ ]\!] && = \mathcal{TB}[\![\ b\ ]\!] + C_{\neg}\\
&\mathcal{TB}[\![\ \mathrm{b}_1 \wedge \mathrm{b}_2\ ]\!] && = \mathcal{TB}[\![\ b_1\ ]\!] + \mathcal{TB}[\![\ b_2\ ]\!] + C_{\wedge}\\
&\mathcal{TB}[\![\ \mathrm{b}_1 \vee \mathrm{b}_2\ ]\!] && = \mathcal{TB}[\![\ b_1\ ]\!] + \mathcal{TB}[\![\ b_2\ ]\!] + C_{\vee}
\end{aligned}
$$

Figure 4.4: Cost of Arithmetic and Boolean Expressions.

proving time restrictions statically using our VCG. For simplicity in the rest of the document, we will consider all the atomic costs as 1, except in logic definitions and soundness proofs.

4.2.3 Free Variables and Substitution

Before we can define the semantics of a statement we need to first look at two important definitions: Free Variables, and Substitution.

Definition 8 (Free Variables)**.** *The Free Variables of an arithmetic expression can be defined as the set of variables occurring in an expression that are not bounded by any variable binding operator, such as $\sum$.*
If we define this as a function $FV : a \to \{x\}$ we get the definition in figure 4.5.

For example, the free variables of $\sum_{x=0}^{10} x + 2^y + z$ are $\{y, z\}$.

Definition 9 (Substitution)**.** *A substitution consists of replacing every occurrence of a variable (x_1) in an arithmetic expression (a) with another arithmetic expression (a_0). This is written as $a[a_0/x_1]$ and the substitutions rules are as described in figure 4.6.*

As an example, let us look at the following substitution

$$(x + 4y + 3)[z + 4/y] = x + 4(z + 4) + 3$$

$$
\begin{aligned}
FV(n) &= \emptyset \\
FV(x) &= \{x\} \\
FV(x[a]) &= \{x[a]\} \cup FV(a) \\
FV(a_1 + a_2) &= FV(a_1) \cup FV(a_2) \\
FV(a_1 - a_2) &= FV(a_1) \cup FV(a_2) \\
FV(a_1 * a_2) &= FV(a_1) \cup FV(a_2) \\
FV({a_1}^{a_2}) &= FV(a_1) \cup FV(a_2) \\
FV(\sum_{x=n}^{a_1} a_2) &= FV(a_1) \cup FV(a_2) - \{x\}
\end{aligned}
$$

Figure 4.5: Free Variables of Arithmetic Expressions.

$$
\begin{aligned}
n[a_0/x_1] &= n \\
x[a_0/x_1] &= \begin{cases} a_0, & \text{if } x = x_1 \\ x, & \text{if } x \neq x_1 \end{cases} \\
x[a][a_0/x_1] &= \begin{cases} a_0, & \text{if } x_1 = x[a] \\ x[(a[a_0/x_1])], & \text{if} x_1 \neq x[a] \end{cases} \\
(a_1 + a_2)[a_0/x_1] &= a_1[a_0/x_1] + a_2[a_0/x_2] \\
(a_1 - a_2)[a_0/x_1] &= a_1[a_0/x_1] - a_2[a_0/x_2] \\
(a_1 * a_2)[a_0/x_1] &= a_1[a_0/x_1] * a_2[a_0/x_2] \\
(a_1/a_2)[a_0/x_1] &= a_1[a_0/x_1]/a_2[a_0/x_2] \\
({a_1}^{a_2}[a_0/x_1] &= a_1[a_0/x_1]^{a_2[a_0/x_2]} \\
(\sum_{x=n}^{a_1} a_2)[a_0/x_1] &= \begin{cases} \sum_{x=n}^{a_1[a_0/x_1]} a_2, & \text{if } x = x_1 \\ \sum_{x=n}^{a_1[a_0/x_1]} a_2[a_0/x_1], & \text{if } x \neq x_1 \end{cases}
\end{aligned}
$$

Figure 4.6: Substitution algorithm for arithmetic expressions.

Substitutions might also be applied to states. For example, $\sigma[n/x]$ represents a state that is identical to σ, with the exception that x takes the value of n. Note that, since a state is a mapping from variable to an integer value, n has to always be an integer and never an expression.

4.2.4 Evaluating statements

In order to prove assertions on the execution time of a program, we need to define a cost-aware semantics. We define a natural operational semantics, where transitions are of the form

$$\langle S, \sigma \rangle \rightarrow^t \sigma'$$

meaning that after executing statement S from state σ the final state is σ' and the execution time was t.

The cost-instrumented operational semantics is defined in Figure 4.7.

$$[\mathit{skip}] \qquad \langle \text{skip}, \sigma\rangle \rightarrow^{C_{SKIP}} \sigma$$

$$[\mathit{assign}] \qquad \langle x = a, \sigma\rangle \rightarrow^{\mathcal{TA}\llbracket a\rrbracket + C_{ASSIGN\_V}} \sigma[\ \mathcal{A}\llbracket a\rrbracket\sigma\ /\ x\ ]$$

$$[\mathit{array}] \qquad \langle x[a_1] = a_2, \sigma\rangle \rightarrow^{\mathcal{TA}\llbracket a_1\rrbracket + \mathcal{TA}\llbracket a_2\rrbracket + C_{ASSIGN\_A}} \sigma[\ \mathcal{A}\llbracket a_2\rrbracket\sigma\ /\ x[\mathcal{A}\llbracket a_1\rrbracket\sigma]\ ]$$

$$[\mathit{seq}] \qquad \frac{\langle S_1, \sigma\rangle \rightarrow^{t_1} \sigma' \qquad \langle S_2, \sigma'\rangle \rightarrow^{t_2} \sigma''}{\langle S_1; S_2, \sigma\rangle \rightarrow^{t_1+t_2} \sigma''}$$

$$[\mathit{if}^{\text{true}}] \qquad \frac{\langle S_1, \sigma\rangle \rightarrow^{t} \sigma'}{\langle \text{if } b \text{ then } S1 \text{ else } S2, \sigma\rangle \rightarrow^{\mathcal{TA}\llbracket b\rrbracket + t} \sigma'} \qquad \text{if } \mathcal{B}\llbracket b\rrbracket\sigma = \text{true}$$

$$[\mathit{if}^{\text{false}}] \qquad \frac{\langle S_2, \sigma\rangle \rightarrow^{t} \sigma'}{\langle \text{if } b \text{ then } S1 \text{ else } S2, \sigma\rangle \rightarrow^{\mathcal{TA}\llbracket b\rrbracket + t} \sigma'} \qquad \text{if } \mathcal{B}\llbracket b\rrbracket\sigma = \text{false}$$

$$[\mathit{while}^{\text{true}}] \qquad \frac{\langle S, \sigma\rangle \rightarrow^{t} \sigma'' \qquad \langle \text{while } b \text{ do } S, \sigma''\rangle \rightarrow^{t'} \sigma'}{\langle \text{while } b \text{ do } S, \sigma\rangle \rightarrow^{\mathcal{TB}\llbracket b\rrbracket + t + t'} \sigma'} \qquad \text{if } \mathcal{B}\llbracket b\rrbracket\sigma = \text{true}$$

$$[\mathit{while}^{\text{false}}] \qquad \langle \text{while } b \text{ do } S, \sigma\rangle \rightarrow^{\mathcal{TB}\llbracket b\rrbracket} \sigma \qquad \text{if } \mathcal{B}\llbracket b\rrbracket\sigma = \text{false}$$

Figure 4.7: Operational Semantics.

The skip axiom [*skip*] says that *skip* does not change the state of the program, and we associate a constant cost for its execution of C_{SKIP}.

The assignment axiom [*assign*] says that executing the assignment $x = a$ in state σ will lead to a state $\sigma[\mathcal{A}\llbracket a\rrbracket\sigma/x]$, which means state σ where x takes the value of $\mathcal{A}\llbracket a\rrbracket\sigma$. The cost of this expression is defined as the cost of evaluating a, $\mathcal{TA}\llbracket a\rrbracket$, plus the constant cost of an assignment, C_{ASSIGN_V}.

Similarly, the array assignment axiom [*array*] says that executing $x[a_1] = a_2$ from state σ will lead to state $\sigma[\mathcal{A}\llbracket a_2\rrbracket\sigma/x[\mathcal{A}\llbracket a_2\rrbracket s\sigma]$, which is a similar state to σ, except $x[\mathcal{A}\llbracket a_1\rrbracket\sigma]$ takes the value of $\mathcal{A}\llbracket a_2\rrbracket\sigma$. This execution cost will be the cost of evaluating a_1, $\mathcal{TA}\llbracket a_1\rrbracket$, plus the cost of evaluating a_2, $\mathcal{TA}\llbracket a_2\rrbracket$, plus the constant cost of assigning a value to a position in an array, C_{ASSIGN_A}.

The sequence rule [*seq*] says that if we want to execute a sequence $S_1; S_2$ from state σ we will first execute statement S_1 from state σ, this execution will lead to a certain state σ' in t_1 time. If we execute S_2 from this state σ' we will reach a final state σ'' in t_2 time. Therefore executing $S_1; S_2$ from state σ will lead to state σ'' in $t_1 + t_2$ time.

We have two conditional rules, $[if^{true}]$ and $[if^{false}]$. In order to decide which of the rules to apply, we must first evaluate $\mathcal{B}\llbracket b\rrbracket\sigma$. If this evaluates to true we apply rule $[if^{true}]$, which means we simply execute statement S_1, otherwise we apple rule $[if^{false}]$, which means we execute statement S_2. For both rules, the cost of executing the if statement is the cost of executing S_1 when true or S_2 when false, plus the cost of evaluating b, $\mathcal{TB}\llbracket b\rrbracket$.

We have one rule and one axiom for while, $[while^{true}]$ and $[while^{false}]$.

If $\mathcal{B}\llbracket b\rrbracket\sigma$ is false, we apply the axiom $[while^{false}]$, that says we will remain in the same state σ and the cost is simply the cost of the evaluation of b, $\mathcal{TB}\llbracket b\rrbracket$.

If $\mathcal{B}\llbracket b\rrbracket\sigma$ is true, we apply rule $[while^{true}]$, which means we will execute the loop body, S, once from state σ and this will lead to a state σ''. Finally, we execute the while loop again, but this time from state σ''. The cost of the while loop, in this case, is the cost of evaluating b, plus the cost of executing the body, plus the cost of executing the while loop from state σ''.

Example

Let us consider the following program that swaps the values of x and y:

```
z = x; x = y; y = z
```

Let the initial state σ_0 be a state such that $\sigma_0\ x = 3$, $\sigma_0\ y = 10$ and $\sigma_0\ z = 0$.

Then the derivation tree of this program will look like

$$\dfrac{\dfrac{\overline{\langle z = x, \sigma_0\rangle \rightarrow^{t_1} \sigma_1} \qquad \overline{\langle x = y, \sigma_1\rangle \rightarrow^{t_2} \sigma_2}}{\langle z = x; x = y, \sigma\rangle \rightarrow^{t_1+t_2} \sigma_2} \qquad \overline{\langle y = z, \sigma_2\rangle \rightarrow^{t_3} \sigma_3}}{\langle z = x; x = y; y = z, \sigma_0\rangle \rightarrow^{t_1+t_2+t_3} \sigma_3}$$

Where $\sigma_1 = \sigma_0[3/z]$, $\sigma_2 = \sigma_1[10/x]$ and $\sigma_3 = \sigma_2[3/y]$.

From the assign rule we get that $t_1 = \mathcal{TA}\llbracket x\rrbracket + C_{ASSIGN\_V} = C_{VAR} + C_{ASSIGN\_V}$, t_2 and t_3 will be the same. Therefore $t_1 + t_2 + t_3 = 3 \times (C_{VAR} + C_{ASSIGN\_V})$.

4.3 Axiomatic Semantics

We will now define a logic with triples in the form $\{P\}S\{Q|t\}$. This triple can be read as executing S from a state σ that validates the precondition P leads to a state that validates postcondition Q, and this execution costs at most t to complete. We will call these triples *total correctness assertions.*

Before defining our assertion language, we must distinguish between program variables and logic variables. Let us imagine the following triple $\{x = n\}y = x + 1\{y > n\}$. In this case, x and y are program variables since they are both present in the statement inside the triple. In our pre and postconditions, we reference variable n, which does not appear on the program. This is what we call a *logic variable.*

Assertions Our language supports annotations with preconditions and postconditions in order to prove correctness, termination, and time restrictions on our program. Our assertion language will be an extension of the boolean expressions extended with quantifiers over integer variables and implications.

$$\begin{aligned}&P ::= true \mid false\\ &\mid a_1 = a_2 \mid a_1 \neq a_2 \mid a_1 < a_2 \mid a_1 > a_2 \mid a_2 \leq a_2 \mid a_1 \geq a_2\\ &\mid \neg P \mid P \wedge Q \mid P \vee Q \mid P \implies Q\\ &\mid \exists x.\ P \mid \forall x.\ P\end{aligned}$$

Note that a_1 and a_2 are arithmetic expressions extended with logic variables. As assertions may include logic variables, the semantics require an interpretation function to provide the value of a variable. Given an interpretation, it is convenient to define the states which satisfy an assertion. We will use the notation $\sigma \models^I P$ to denote that state σ satisfies P in interpretation I, or equivalently that assertion P is true at state σ, in interpretation I. The definition of $\sigma \models^I P$ is the usual for a first-order language.

Now, note that we are not interested in the particular values associated with variables in an interpretation I. We are interested in whether or not an assertion is true at all states for all interpretations. This motivates the following definition.

Definition 10 (Validity). $\models \{P\}S\{Q|t\}$ *if and only if, for every state* σ *and interpretation* I *such that* $\sigma \models^I P$ *and* $\langle S, \sigma\rangle \to^{t'} \sigma'$ *we have that* $\sigma' \models^I Q$ *and* $\mathcal{A}[\![t]\!]\sigma \geq t'$.

If $\models \{P\}S\{Q|t\}$ we say that the total correctness assertion $\{P\}S\{Q|t\}$ is *valid.*

We now define a set of proof rules that generate valid total correctness assertions. The rules of our logic are given in figure 4.8.

The *skip axiom* says that if P is true before executing *skip*, then it is also true after its execution. The upper bound for this execution is the constant C_{SKIP}.

The *assign axiom* says that P will be true after executing $x = a$ if $P[\mathcal{A}[\![a]\!]/x]$ is true before its execution. The upper bound of this execution is defined as the sum of evaluating a and assigning a simple variable $\mathcal{TA}[\![a]\!] + C_{ASSIGN\_V}$.

The *array axiom* is very similar to the assignment axiom, except in the upper bound we also need to consider the time to evaluate a_1.

The *seq rule* says if P is true before executing $S_1; S_2$ then R will be true after the execution, as long as we can prove that

- If P is true before executing S_1 then Q is true after and t_1 is an upper bound on this execution
- If Q is true before executing S_2 then R is true after and t_2 is an upper bound on this execution

$$[Skip] \qquad \{\ P\ \} skip \{\ P \mid C_{SKIP}\ \}$$

$$[Assign] \qquad \{\ P[\ \mathcal{A}\llbracket a \rrbracket / x\ ]\ \}\ x = a\ \{\ P \mid \mathcal{TA}\llbracket a \rrbracket + C_{ASSIGN\_V}\ \}$$

$$[Array] \qquad \{\ P[\ \mathcal{A}\llbracket a_2 \rrbracket / x[\mathcal{A}\llbracket a_1 \rrbracket]\ ]\ \}\ x[a_1] = a_2\ \{\ P \mid \mathcal{TA}\llbracket a_1 \rrbracket + \mathcal{TA}\llbracket a_2 \rrbracket + C_{ASSIGN\_A}\ \}$$

$$[Seq] \qquad \frac{\{\ P\ \}\ S_1\ \{\ Q \mid t_1\ \} \qquad \{\ Q\ \}\ S_2\ \{\ R \mid t_2\ \}}{\{\ P\ \}\ S_1;\ S_2\ \{\ R \mid t_1 + t_2\ \}}$$

$$[If] \qquad \frac{\{\ P \wedge \mathcal{B}\llbracket b \rrbracket\ \}\ S_1\ \{\ Q \mid t_1 \} \qquad \{\ P \wedge \neg\mathcal{B}\llbracket b \rrbracket\ \}\ S_2\ \{\ Q \mid t_2\ \}}{\{\ P\ \}\ \text{if } b \text{ then } S_1 \text{ else } S_2\ \{\ Q \mid max(t_1, t_2) + \mathcal{TB}\llbracket b \rrbracket\ \}}$$

$$[While] \qquad \frac{\underline{I} \wedge \mathcal{B}\llbracket b \rrbracket \Rightarrow \underline{f} \leq \underline{N} \qquad \forall k. \{\ \underline{I} \wedge \mathcal{B}\llbracket b \rrbracket\ \wedge f = k \}\ S\ \{\ \underline{I}\ \wedge \underline{f} > k \mid t(k) \}}{\{\ \underline{I}\ \wedge \underline{f} \geq 0\ \}\ \text{while } b \text{ do } S\ \{\ I \wedge \neg\mathcal{B}\llbracket b \rrbracket \mid \sum_{i=0}^{N-1} t(i) + (N+1) \times \mathcal{TB}\llbracket b \rrbracket \}}$$

$$[Weak] \qquad \frac{\{P'\} s \{Q' | t'\} \qquad P \Rightarrow P' \qquad Q' \Rightarrow Q \qquad t' \leq t}{\{P\} S \{Q | t\}}$$

Figure 4.8: Proof Rules for the IMP Language.

The upper bound for the sequence is the sum of both upper bounds, $t_1 + t_2$.

The *if rule* says if P is true before executing if b then S_1 else S_2, then Q will be true after the execution, as long as we can prove that

- If P and $\mathcal{B}\llbracket b \rrbracket$ are both true before executing S_1, then Q will be true after executing it. This execution is upper bounded by t_1
- If P and $\neg\mathcal{B}\llbracket b \rrbracket$ are both true before executing S_2, then Q will be true after executing it. This execution is upper bounded by t_2

In the *while rule*, f, I, N and $t(k)$ are values provided by an oracle: in an interactive proof system, these can be provided by the user, and in a non-interactive setting, they can be annotated into the program. I is the loop invariant, which must remain true before and after every iteration of the loop. f is a termination function that must start as a positive value, increase with every iteration, but remain smaller than N. N is therefore the maximum number of iterations. $t(k)$ is a function describing the cost of the body of the while at iteration k. If the *while* loop runs at most N times and, for each k iteration, the cost of the loop body is given by $t(k)$ we have the following upper bound for the while statement: the sum of the cost of all the iterations $\sum_{i=0}^{N-1} t(i)$; plus the sum of evaluating the loop condition, b, each time we enter the while body (at most N), plus one evaluation of b when the condition fails, and the loop terminates. This leads to the term $\sum_{i=0}^{N-1} t(i) + (N+1) \times \mathcal{TB}\llbracket b \rrbracket$ used in the rule for the *while* loop.

Besides each rule and axiom for every statement in our language, we need an extra rule that we call *weak rule.* This rule says

- If a weaker precondition is sufficient, then so is a stronger one. If we know $\models \{P\}S\{Q|t\}$ and $P' \to P$, then $\models \{P'\}S\{Q|t\}$,
- If a stronger postcondition is provable, then so is a weaker one. If we know $\models \{P\}S\{Q|t\}$ and $Q \to Q'$, then $\models \{P\}S\{Q'|t\}$
- An upper bound can always be replaced with a bigger one. If we know $\models \{P\}S\{Q|t\}$ and $t' > t$, then $\models \{P\}S\{Q|t'\}$

Proof rules should preserve validity in the sense that if the premise is valid, then so is the conclusion. When this holds for every rule, we say that the proof system is *sound.* For the proposed Hoare logic, it is generally easy to show by induction that every assertion $\{P\}S\{Q|t\}$ which is the conclusion of a derivation in the proof system, is a valid assertion.

The proof of soundness depends on an important property of substitution.

Lemma 1. *Let P be an assertion, x a variable, and a an arithmetic expression. Then for every state σ*

$$\sigma \models P[a/x] \quad \textit{iff} \quad \sigma[\mathcal{A}[\![a]\!]\sigma/x] \models P$$

Proof. The proof follows by structural induction on P. □

Theorem 3 (Soundness)**.** *Let $\{P\}S\{Q|t\}$ be a total correctness assertion. Then*

$$\vdash \{P\}S\{Q|t\} \quad \textit{implies} \quad \models \{P\}S\{Q|t\}$$

Proof. The proof follows by induction on the length of the derivation of $\{P\}S\{Q|t\}$.

Case Skip: Consider a state σ, where $\sigma \models P$. According to our semantic, after executing *skip* we are still in state σ and the statement *skip* takes C_{SKIP} to execute, $\langle \sigma, \text{skip} \rangle \to^{C_{SKIP}} \sigma$. Therefore, the rule for *skip* is sound.

$$\models \{P\}skip\{P|C_{SKIP}\}$$

Case Assign: Consider a state σ where $\sigma \models P[\mathcal{A}[\![a]\!]/x]$. We have $\langle \sigma, x = a \rangle \to^{\mathcal{TA}[\![a]\!]+C_{ASSIGN\_V}}$ $\sigma[\mathcal{A}[\![a]\!]\sigma \ / \ x]$, that is, after the statement is executed we are in state $\sigma[\mathcal{A}[\![a]\!]\sigma \ / \ x]$ and it costs $\mathcal{TA}[\![a]\!] + C_{ASSIGN\_V}$. By the substitution lemma 1 we have that $\sigma[\mathcal{A}[\![a]\!]\sigma \ / \ x] \models P$. Therefore, the rule for assignment is sound.

$$\models \{ \ P[\ \mathcal{A}[\![a]\!]/x \ ] \ \} \ x = a \ \{ \ P \mid \mathcal{TA}[\![a]\!] + C_{ASSIGN\_V} \ \}$$

Case Array: Consider a state σ where $\sigma \models P[\mathcal{A}[\![a_2]\!]/x[\mathcal{A}[\![a_1]\!]]]$. We have

$$\langle \sigma, x[a_1] = a_2 \rangle \to^{\mathcal{TA}[\![a]\!]+C_{ASSIGN\_A}} \sigma[\mathcal{A}[\![a_2]\!]\sigma/x[\mathcal{A}[\![a_1]\!]\sigma]]$$

That is, after the statement is executed we are in state

$$\sigma[\mathcal{A}[\![a_2]\!]\sigma/x[\mathcal{A}[\![a_1]\!]\sigma]]$$

and the cost is $\mathcal{TA}[\![a]\!] + C_{ASSIGN\_A}$. By lemma 1 we have that $\sigma[\mathcal{A}[\![a_2]\!]\sigma/x[\mathcal{A}[\![a_1]\!]\sigma]] \models P$. Therefore, the rule for assignment to an array is sound.

$$\models \{\ P[\ \mathcal{A}[\![a_2]\!]/x[\mathcal{A}[\![a_1]\!]]\ ]\ \}\ x[a_1] = a_2\ \{\ P \mid \mathcal{TA}[\![a_1]\!] + \mathcal{TA}[\![a_2]\!] + C_{ASSIGN\_A}\ \}$$

Case Sequence: Assume $\models \{P\}S_1\{Q|t_1\}$ and $\models \{Q\}S_2\{R|t_2\}$. Lets consider a state σ that validates P, $\sigma \models P$. From our semantics we have that, there is a state σ_1 such that $\langle S_1, \sigma\rangle \to^t \sigma_1$. Since we have that $\models \{P\}S_1\{Q|t_1\}$, then $\sigma_1 \models Q$, and $\mathcal{A}[\![t_1]\!]\sigma \geq t$. We also have that $\langle S_2, \sigma_1\rangle \to^{t'} \sigma_2$, and since we assume $\models \{Q\}S_2\{R|t_2\}$, then $\sigma_2 \models R$, and $\mathcal{A}[\![t_2]\!]\sigma_1 \geq t'$. Since t_1 and t_2 are both constant values, $\mathcal{A}[\![t_2]\!]\sigma_1 = \mathcal{A}[\![t_2]\!]\sigma$, which means $\mathcal{A}[\![t_1 + t_2]\!]\sigma \geq t + t'$. Therefore the proof for *sequence* is sound.

$$\models \{P\}S_1; S_2\{Q|t_1 + t_2\}$$

Case If: Assume $\models \{P \wedge b\}S_1\{Q|t_1\}$ and $\models \{P \wedge \neg b\}S_2\{Q|t_2\}$. Suppose $\sigma \models P$.

If $\sigma \models b$, then $\sigma \models P \wedge b$ so, assuming $\langle S_1, \sigma\rangle \to^t \sigma_1$, we have that $\sigma_1 \models Q$, and $\mathcal{A}[\![t_1]\!]\sigma \geq t$.

If $\sigma \models \neg b$, then $\sigma \models P \wedge \neg b$ so, assuming $\langle S_2, \sigma\rangle \to^{t'} \sigma_2$, we have that $\sigma_2 \models Q$, and $\mathcal{A}[\![t_2]\!]\sigma \geq t'$.

Since $\mathcal{A}[\![\max(t_1, t_2) + \mathcal{TB}[\![b]\!]]\!]\sigma \geq \mathcal{TB}[\![b]\!] + t$ and $\mathcal{A}\ [\![\max(t_1, t_2) + \mathcal{TB}[\![b]\!]]\!]\sigma \geq \mathcal{TB}[\![b]\!] + t'$. The rule for *if* is sound.

$$\models \{\ P\ \}\ \text{if } b \text{ then } S_1 \text{ else } S_2\ \{\ Q \mid max(t_1, t_2) + \mathcal{TB}[\![b]\!]\ \}$$

Case While: Assume $I \wedge \mathcal{B}[\![b]\!] \Rightarrow f \leq N$ and $\models \{\ I \wedge \mathcal{B}[\![b]\!]\ \wedge f = k\}\ S\ \{\ I\ \wedge f > k \mid t(k)\}$. Considering a state σ such that $\sigma \models I \wedge f \geq 0$ and $\langle \text{while } b \text{ do } S, \sigma\rangle \to^t \sigma_1$.

If $\sigma \models \neg\mathcal{B}[\![b]\!]$ then $\sigma_1 = \sigma$, therefore $\sigma_1 \models \neg\mathcal{B}[\![b]\!] \wedge I$ and the cost is $t = \mathcal{TB}[\![b]\!]$.

If $\sigma \models \mathcal{B}[\![b]\!]$, we have that $\sigma \models \mathcal{B}[\![b]\!] \wedge I$. Considering a state σ_2 such that $\langle S, \sigma\rangle \to^{t'} \sigma_2$ and $\langle \text{while } e \text{ do } S, \sigma_2\rangle \to^{t''} \sigma_1$. Applying our initial assumption we get $\sigma_2 \models I \wedge f > k$, and $\mathcal{A}[\![t(k)]\!]\sigma \geq t'$. Finally by applying the induction hypothesis we have that $\sigma_1 \models \neg\mathcal{B}[\![b]\!] \wedge I$.

Given the function f provided by the user and taking into account our assumptions, we know that the program will eventually stop and at most, it will iterate $N + 1$ times. For each iteration the cost is $\mathcal{TB}[\![b]\!] + t'$ and $\mathcal{TB}[\![b]\!]$ for the last time, when b is false and the while terminates. Therefore the cost of this program is always smaller or equal to

$$\mathcal{TB}[\![b]\!] + \sum_{i=0}^{N}(\mathcal{TB}[\![b]\!] + t')$$

Since $t(k) \geq t'$, our upper bound is

$$\sum_{i=0}^{N+1} \mathcal{TB}[\![b]\!] + \sum_{i=0}^{N} t(k)$$

Therefore the rule for *while* is sound.

$$\models \{\ I\ \ \wedge f \geq 0\ \}\ \text{while}\ b\ \text{do}\ S\ \{\ I \wedge \neg\mathcal{B}[\![b]\!]\ |\ \sum_{i=0}^{N} t(i) + \sum_{i=0}^{N+1} \mathcal{TB}[\![b]\!]\}$$

□

Example: Division Algorithm

To illustrate our logic, let us apply our rules to the division algorithm, as presented in figure 4.1. For simplicity we assume unitary values for all "atomic" operations (C_+, C_{SKIP}, etc).

Since our example contains a while loop we will have to define, the invariant I as $x = q \times y + r \wedge y \geq 0 \wedge r \geq 0$, the variant f as $x - r$, the maximum number of iterations N as x, and the function of cost $t(k)$ as $fun\ k \rightarrow 10$.

Let us consider $S_w \equiv r = r - y; q = q + 1$.

By the *assign rule* we know

$$\vdash \{(I \wedge f > k)[q + 1/q]\}q = q + 1\{I \wedge f > k|\mathcal{TA}[\![q + 1]\!] + 1\} \tag{4.1}$$

Where

$$\mathcal{TA}[\![q + 1]\!] = \mathcal{TA}[\![q]\!] + \mathcal{TA}[\![1]\!] + 1 = 3$$

$$(I \wedge f > k)[q + 1/q] \equiv x = (q + 1) \times y + r \wedge y \geq 0 \wedge r \geq 0 \wedge x - r > k$$

Again by the *assign rule* we know

$$\vdash \{(I \wedge f > k)[q + 1/q][r - y/r]\}r = r - y\{(I \wedge f > k)[q + 1/q]|\mathcal{TA}[\![r - y]\!] + 1\} \tag{4.2}$$

Where

$$\mathcal{TA}[\![r - y]\!] = \mathcal{TA}[\![r]\!] + \mathcal{TA}[\![y]\!] + 1 = 3$$

$$(I \wedge f > k)[q + 1/q][r - y/r] \equiv x = (q + 1) \times y + r - y \wedge y \geq 0 \wedge r - y \geq 0 \wedge x - r + y > k$$

Since 4.1, and 4.2, by the *seq rule*, we have

$$\vdash \{(I \wedge f > k)[q + 1/q][r - y/r]\}S_w\{I \wedge f > k|8\}$$

$$\begin{aligned} I \land y \leq r \land f = k \quad &\equiv \quad x = q \times y + r \land y \geq 0 \land r \geq 0 \land y \geq r \land x - r = k \\ &\rightarrow \quad x = (q+1) \times y + r - y \land y \geq 0 \land r \geq y \land x - r + y \geq k \\ &\equiv \quad (I \land f > k)[q+1/q][r-y/r] \end{aligned}$$

Given this result, and since $10 \geq 8$, by the *weak rule* we get

$$\vdash \{I \land y \leq r \land f = k\} r = r - y; q = q + 1 \{I \land f > k | 10\}$$

It also apparent that

$$I \land y \leq r \rightarrow r \geq 0 \rightarrow x - r \leq x$$

We are now ready to apply the *while rule*, and we get

$$\vdash \{I \land f \geq 0\} S \{I \land \neg(y \leq r) | \sum_{i=0}^{x-1} 10 + (x+1) \times \mathcal{TB}[\![y \leq r]\!]\}$$

Where

$$\sum_{i=0}^{x-1} 10 + \mathcal{TB}[\![y \leq r]\!] = 10 \times x + (x+1) \times 3 = 13x + 3$$

Since $P \rightarrow I \land f \geq 0$, $I \land \neg(y \leq r)$, and $20x + 5 \geq 13x + 3$, by the *weak rule* we get

$$\vdash \{P\} S \{Q | T\}$$

4.4 Verification Conditions Generation

To implement a verification system based on our logic, we define a Verification Condition Generator (VCG) based on the weakest-precondition algorithm.

Given a Hoare triple $\{P\}S\{Q|t\}$, we start by identifying the weakest precondition of S given Q as the postcondition. In figure 4.9 we define the Weakest Precondition-Cost (wpc) function which receives a statement and a postcondition and returns a tuple (wp, t_S), where wp is the weakest precondition of the program and t_S is an upper bound on the program's cost. The weakest precondition is calculated by a standard algorithm such as the one presented in [2]. Let us focus on the second value of our tuple, which estimates an upper bound for the program. This upper bound will be equivalent to the ones presented in figure 4.2. The upper-bound for *skip*, *assignment*, and *array* are very straightforward since they are the same as the exact cost calculated by our operational semantics in section 4.2. To calculate the upper bound of a sequence $S_1; S_2$ we need to both calculate the upper bound of $S1$ given by t_1 and the upper bound of S_2, given by t_2. The upper bound for the sequence is then defined as the sum of both upper bounds $t_1 + t_2$. In the case of *if*, we calculate the upper bound of each conditional statement $S1$ and S_2. The upper bound for if is defined as the max between both upper bounds $max(t_1, t_2)$ plus the cost of evaluating b, $\mathcal{TB}[\![b]\!]$. Finally, looking at the while, the upper bound is defined as the sum

$$
\begin{aligned}
\mathbf{wpc}(\text{skip}, Q) &= (Q\ ,\ C_{SKIP}) \\[1em]
\mathbf{wpc}(x := a, Q) &= (Q[a/x]\ ,\ C_{ASSIGN\_V} + \mathcal{TA}\llbracket a \rrbracket)) \\[1em]
\mathbf{wpc}(x[a_1] := a_2, Q) &= (Q[a_2/x[a_1]]\ ,\ C_{ASSIGN\_A} + \mathcal{TA}\llbracket a_1 \rrbracket + \mathcal{TA}\llbracket a_2 \rrbracket) \\[1em]
\mathbf{wpc}(S_1; S_2, Q) &= (wp_1\ ,\ t_1 + t_2) \\
&\ \textit{where } (wp_2\ ,\ t_2) = \mathbf{wpc}(S_2, Q) \\
&\ (wp_1\ ,\ t_1) = \mathbf{wp1}(S_1, wp_2) \\[1em]
\mathbf{wpc}(\text{if } b \text{ then } S_1 \text{ else } S_2, Q) &= ((b \to wp_1) \wedge (\neg b \to wp_2)\ ,\ max(t_1, t_2) + \mathcal{TB}\llbracket b \rrbracket) \\
&\ \textit{where } (wp_1\ ,\ t_1) = \mathbf{wpc}(S_1, Q), \\
&\ (wp_2\ ,\ t_2) = \mathbf{wpc}(S_2, Q) \\[1em]
\mathbf{wpc}(\text{while } b \text{ do } S, Q) &= (\underline{I} \wedge \underline{f} \geq 0\ ,\ \sum_{i=0}^{\underline{N}+1} \times \mathcal{TB}\llbracket b \rrbracket + \sum_{i=0}^{\underline{N}} \underline{t(i)})
\end{aligned}
$$

Figure 4.9: Weakest Precondition Algorithm.

of evaluating the while condition N+1 times and the sum of the cost of each execution of the *while* body, given by t(k).

The program is correct if P implies the weakest precondition. If $t \geq t_S$, we also guarantee that t is, in fact, an upper bound on the execution of S. Even though these conditions are enough to prove both correction and cost upper-bound of our program, we still need extra Verification Conditions to handle while loops. The VC function in figure 4.10 receives a program and a postcondition. It returns a set of purely mathematical statements (the verification conditions) needed to handle loops and guarantee termination. Let us take a more in-depth look to the *while* case, $VC(\text{while } b \text{ do } S, Q)$. To prove the while statement, we need the oracle to provide some extra information. All the values provided need to be demonstrated. Firstly, as seen previously, the weakest precondition of a while is the invariant being true and the variant being positive. The loop invariant is assured by the following VC

$$\forall k. I \to wp_S$$

We also need to guarantee the termination of the loop. For this we prove that the variant (f) always increases $\forall k, I \wedge b \wedge f = k \to wp_S$ and that the variant is always limited by N, $I \wedge \mathcal{B}\llbracket b \rrbracket \to f \leq N$. Lastly, we prove the postcondition $I \wedge \neg b \to Q$, and call VC recursively for the body. The VC function applied to *sequence* and *if*, is just a recursive call of the function to their sub-statements. Finally for *skip*, *assign*, and *array* there are no extra Verification Condition (VC). The function VCG is the glue that puts all these VCs together. The first condition $P \to wp$ implies the correctness of our program. Secondly, we call $VC(S, Q)$ to potentially deal with loop invariants, termination, and cost. The last condition states that $t \geq t_S$ proves if the user-provided upper bound is indeed valid. These verification conditions are then passed to an interactive prover, such as EasyCrypt [5], which attempts to prove them automatically. If it fails, some advice is needed from the user.

$$\begin{aligned}
\mathbf{VC}(\text{skip}, Q) &= \emptyset \\[1em]
\mathbf{VC}(x := a, Q) &= \emptyset \\[1em]
\mathbf{VC}(x[a_1] := a_2, Q) &= \emptyset \\[1em]
\mathbf{VC}(S_1; S_2, Q) &= \mathbf{VC}(S_1, wp_2) \ \cup \ \mathbf{VC}(S_2, Q) \\
\textit{where} \quad & \quad (\text{wp}_2, t_2) = \mathbf{wpc}(S_2, Q) \\[1em]
\mathbf{VC}(\text{if } B \text{ then } S_1 \text{ else } S_2, Q) &= \mathbf{VC}(S_1, Q) \ \cup \ \mathbf{VC}(S_2, Q) \\[1em]
\mathbf{VC}(\text{while } b \text{ do } S, Q) &= \{\forall k. (\underline{I} \wedge b \wedge \underline{f} = \underline{k}) \rightarrow wp_S \wedge \underline{t(k)} \geq t_S\} \cup \\
& \quad \ \{(\underline{I} \wedge \neg b) \rightarrow Q\} \ \cup \ \mathbf{VC}(S, \underline{I}) \cup \\
& \quad \ \{\underline{I} \wedge \mathcal{B}[\![b]\!] \Rightarrow \underline{f} \leq \underline{N}\} \\
\textit{where} \quad & \quad (wp_S, t_S) = \mathbf{wpc}(S, \underline{I} \wedge \underline{f} > \underline{k})
\end{aligned}$$

$$\begin{aligned}
&\mathbf{VCG}(\ \{P\}\ S\ \{Q \mid t\}\ ) = \{P \rightarrow wp\} \ \cup \ \mathbf{VC}(S, Q) \cup \ \{P \rightarrow t \geq t_S\} \\
&\textit{where} \\
&\qquad (\text{wp}, t_S) = \mathbf{wpc}(S, Q)
\end{aligned}$$

Figure 4.10: VC and VCG functions.

We need to ensure that this algorithm is actually sound with our Hoare Logic. For this, we need to prove theorem 1 that states that the VCG algorithm is sound if the Verification Condition generated implies of the Hoare triple we wish to prove, $\models VCG(\{P\}Q\{R\}) \implies \vdash \{P\}Q\{R\}$. This assures that by proving our VCs, we are actually proving our triple. If we also want to show that if a triple is valid, then we can also validate every VC generated by the VCG algorithm, then we are looking at the completeness theorem 2, $\vdash \{P\}Q\{R\} \implies \models VCG(\{P\}Q\{R\})$. To prove both soundness and completeness, we then need to prove

$$\models VCG(\{P\}Q\{R\}) \textit{ iff } \vdash \{P\}Q\{R\}$$

Proof. We prove $\implies$ by induction on the structure of Q.

Case Skip:

$$VCG(\{P\}skip\{Q|T\}) = \{P \rightarrow Q\} \cup \{T \geq C_{SKIP}\}$$

By the *skip axio*m, we know $\vdash \{Q\}skip\{Q|C_{SKIP}\}$.

By the weak rule, knowing $\vdash \{Q\}skip\{Q|C_{SKIP}\}$, $P \rightarrow Q$, and $T \geq C_{SKIP}$, we have

$$\vdash \{P\}skip\{Q|T\}$$

Case Assign:

$$VCG(\{P\}x = a\{Q|T\}) = \{P \rightarrow Q[\mathcal{A}[\![a]\!]/x]\} \cup \{T \geq C_{ASSIGN\_V}\}$$

By the *assign axiom* $\vdash \{Q[\mathcal{A}[\![a]\!]/x]\}x = a\{Q|C_{ASSIGN\_V}\}$. By the weak rule, knowing $\vdash \{Q[\mathcal{A}[\![a]\!]/x]\}x = a\{Q|C_{ASSIGN\_V}\}$, $P \rightarrow Q[\mathcal{A}[\![a]\!]/x]$, and $T \geq C_{ASSIGN\_V}$, we have

$$\vdash \{P\}x = a\{Q|T\}$$

Case Array:

$$VCG(\{P\}x[a_1] = a_2\{Q|T\}) = \{P \rightarrow Q[\mathcal{A}[\![a_2]\!]/x[\mathcal{A}[\![a_1]\!]]]\} \cup \{T \geq C_{ASSIGN\_A}\}$$

By the *array axiom* $\vdash \{Q[\mathcal{A}[\![a_2]\!]/x[\mathcal{A}[\![a_1]\!]]]\}x[a_1] = a_2\{Q|C_{ASSIGN\_A}\}$. By the weak rule, knowing $\{Q[\mathcal{A}[\![a_2]\!]/x[\mathcal{A}[\![a_1]\!]]]\}x[a_1] = a_2\{Q|C_{ASSIGN\_A}\}$, $P \rightarrow Q[\mathcal{A}[\![a_2]\!]/x[\mathcal{A}[\![a_1]\!]]]$, and $T \geq C_{ASSIGN\_A}$, we have

$$\vdash \{P\}x[a_1] = a_2\{Q|T\}$$

Case seq:

Induction Hypothesis:

$$\models VCG(\{P\}S_1\{R|T_1\} \rightarrow \vdash \{P\}S_1\{R|T_1\})$$

$$\models VCG(\{P\}S_2\{R|T_2\} \rightarrow \vdash \{P\}S_2\{R|T_2\})$$

Let us consider:

- $wp_2, t_2 = wpc(S_2, Q)$
- $wp_1, t_1 = wpc(S_1, wp_2)$
- $wpc(S1; S2, Q) = (wp_1, t_1 + t_2)$

$$\models VCG(\{P\}S_1; S_2\{R|T\} = \{P \rightarrow wp_1\} \cup \{T \geq t_1 + t_2\} \cup VC(S1; S2, R)$$

Where $VC(S_1; S_2, R) = VC(S_1, wp_2) \cup VC(S_2, R)$

Assuming $\models VCG(\{P\}S_1; S_2\{R|T\}$

- since we know $P \rightarrow wp_1$, $t_1 \geq t_1$, and $\models VC(S_1, wp_2)$, then

$$\models VCG(\{P\}S_1\{wp_2|t_1\}$$

- since we know $wp_2 \rightarrow wp_2$, $t_2 \geq t_2$, and $\models VC(S_2, R)$, then

$$\models VCG(\{wp_2\}S_2\{R|t_2\}$$

By our Induction Hypothesis, we have $\vdash \{P\}S_1\{wp_2|t_1\})$, and $\vdash \{wp_2\}S_2\{R|t_2\})$. By the seq rule

$$\vdash \{P\}S_1; S_2 \ \{R|t_1 + t_2\}$$

Since $T \geq t_1 + t_2$, by the weak rule

$$\vdash \{P\}S_1; S_2\{R|T\}$$

Case if:

Induction Hypothesis:

$$\models VCG(\{P\}S_1\{Q|t_1\} \rightarrow \vdash \{P\}S_1\{Q|t_1\})$$

$$\models VCG(\{P\}S_2\{Q|t_2\} \rightarrow \vdash \{P\}S_2\{Q|t_2\})$$

Let us consider

- $wp_1, t_1 = wpc(S_1, Q)$
- $wp_2, t_2 = wpc(S_2, Q)$
- $wpc(\text{if } b \text{ then } S_1 \text{ else } S_2, Q) = (\mathcal{B}[\![b]\!] \rightarrow wp_1 \wedge \neg\mathcal{B}[\![b]\!] \rightarrow wp_2, max(t_1, t_2) + \mathcal{TB}[\![b]\!])$

$$\begin{aligned} VCG(\{P\}\text{if } b \text{ then } S_1 \text{ else} S_2\{Q|T\}) = &\{P \rightarrow (b \rightarrow wp_1 \wedge \neg b \rightarrow wp_2)\} \cup \\ &\{T \geq max(t_1, t_2) + \mathcal{TB}[\![b]\!]\} \cup \\ &VC(\text{if } b \text{ then } S_1 \text{ else } S_2, Q) \end{aligned}$$

Where $VC(\text{if } b \text{ then } S_1 \text{ else} S_2, Q) = VC(S_1, Q) \cup VC(S_2, Q)$

Assuming $\models VCG(\{P\}\text{if } b \text{ then } S_1 \text{ else } S_2\{Q|T\})$

- Since $P \wedge \mathcal{B}[\![b]\!] \rightarrow wp_1$, $t_1 \geq t_1$, and $VC(S_1, Q)$,

$$\models VCG(\{P \wedge \mathcal{B}[\![b]\!]\}S_1\{Q|t_1\})$$

- Since $P \wedge \neg\mathcal{B}[\![b]\!] \rightarrow wp_2$, $t_2 \geq t_2$, and $VC(S_2, Q)$,

$$\models VCG(\{P \wedge \neg\mathcal{B}[\![b]\!]\}S_2\{Q|t_2\})$$

From our Induction Hypothesis, we have $\vdash \{P \wedge \mathcal{B}[\![b]\!]\}S_1\{Q|t_1\}$, and $\vdash \{P \wedge \neg\mathcal{B}[\![b]\!]\}S_2\{Q|t_2\}$

By the if the rule, we get

$$\{P\}\text{if } b \text{ then } S_1 \text{ else } S_2\{Q|max(t_1, t_2) + \mathcal{TB}[\![b]\!]\}$$

Since $T \geq max(t_1, t_2) + \mathcal{TB}[\![b]\!]$, by the weak rule

$$\vdash \{P\}\text{if } b \text{ then } S_1 \text{ else } S_2\{Q|T\}$$

Case while:

Induction Hypothesis:

$$\models VCG(\{P\}S\{Q|T\}) \rightarrow \vdash \{P\}S\{Q|T\}$$

Let us consider:

- $wpc(\text{while } b \text{ do } S, Q) = (I \wedge f \geq 0, \sum_{i=0}^{N-1} t(i) + (N+1) \times \mathcal{TB}[\![b]\!])$
- $wp_S, t_S = wpc(S, I \wedge f > k)$

$$\begin{aligned} VC(\text{while } b \text{ do } S, Q) = &\{(I \wedge \mathcal{B}[\![b]\!] \wedge f = k) \rightarrow wp_S \wedge t(k) \geq t_S\} \cup \\ &\{(I \wedge \neg b) \rightarrow Q\} \cup \\ &\{(I \wedge B[\![b]\!]) \rightarrow f \leq N\} \cup \\ &VC(S, I \wedge f > k) \end{aligned}$$

$$\begin{aligned} VCG(\{P\}\text{while } b \text{ do } S\{Q|T\}) = &P \rightarrow (I \wedge f \geq 0) \cup \\ &\{P \rightarrow T \geq \sum_{i=0}^{N-1} t(i) + (N+1) \times \mathcal{TB}[\![b]\!]\} \cup \\ &VC(\text{while } b \text{ do } S, Q) \end{aligned}$$

Assuming $\models VCG(\{P\}\text{while } b \text{ do } S\{Q|T\})$

Since $(I \wedge \mathcal{B}[\![b]\!] \wedge f = k) \rightarrow wp_S \wedge t(k) \geq t_S$, and $VC(S, I \wedge f > k)$ we have

$$\models VCG(\{I \wedge \mathcal{B}[\![b]\!] \wedge f = k\}S\{I \wedge f > k|t(k)\})$$

By our Induction Hypothesis

$$\vdash \{I \wedge \mathcal{B}[\![b]\!] \wedge f = k\}S\{I \wedge f > k|t(k)\}$$

Since $I \wedge \mathcal{B}[\![b]\!] \to f \leq N$, and $\vdash \{I \wedge \mathcal{B}[\![b]\!] \wedge f = k\}S\{I \wedge f > k|t(k)\}$, by the *while rule*

$$\vdash \{I \wedge f \geq 0\}\text{while } b \text{ do } S\{I \wedge \mathcal{B}[\![b]\!]| \sum_{i=0}^{N-1} t(i) + (N+1) \times \mathcal{T}\mathcal{B}[\![b]\!]\}$$

Since $P \to (I \wedge f \geq 0)$, $(I \wedge \neg\mathcal{B}[\![b]\!]) \to Q$, and $P \to T \geq \sum_{i=0}^{N-1} t(i) + (N+1) \times \mathcal{T}\mathcal{B}[\![b]\!]$, by the *weak rule*

$$\vdash \{P\}\text{while } b \text{ do } S\{Q|T\}$$

□

Example

Let us apply the VCG algorithm to the division example 4.1. We will use the same notation as in section 4.3 and refer to our precondition as P, our program as S, our postcondition as Q and our cost upperbound as T. The algorithm starts with a call to the VCG function.

$$\begin{aligned} VCG(\{P\}S\{Q|T\}) =& \{P \to wp) \cup && (4.3)\\ & \{P \to T \geq t\} \cup && (4.4)\\ & VC(S, Q) \end{aligned}$$

Where

$$wp, t = wpc(S, Q) = (I \wedge x - r \geq 0, \sum_{i=0}^{x} 10 + (x+1) \times 3)$$

Then the VCG function calls the VC function for S. Since only *while* loops generate extra VCs, we will omit other calls to the VC function for simplicity.

$$\begin{aligned} VC(S, Q) =& \{(I \wedge y \leq r \wedge x - r = k) \to wp_S\} \cup && (4.5)\\ & \{(I \wedge \neg(y \leq r)) \to Q\} \cup && (4.6)\\ & \{(I \wedge y \leq r) \to x - r \leq x\} \cup && (4.7)\\ & \{(I \wedge y \leq r \wedge x - r = k) \to t(k) \geq t_S\} \cup && (4.8)\\ & VC(r = r - y; q = q + 1, I \wedge x - r > k) \end{aligned}$$

Where

$$\begin{aligned}
wp_S, t_S &= wpc(r = r - y; q = q + 1, I \wedge x - r > k) \\
&= (wp_1, t_1 + t_2)
\end{aligned}$$

$$\begin{aligned}
wp_1, t_1 &= wpc(r = r - y, wp_2) \\
&= (wp_2[r - y/r], \mathcal{TA}[\![r - y]\!] + 1)
\end{aligned}$$

$$\begin{aligned}
wp_2, t_2 &= wpc(q = q + 1, I \wedge x - r > k) \\
&= ((I \wedge x - r > k)[q + 1/q], \mathcal{TA}[\![q + 1]\!] + 1)
\end{aligned}$$

To prove our triple, we now simply need to prove all of the VCs generated by the algorithm (4.3 to 4.8), this can easily be done for all the conditions manually, or with the assistance of a theorem prover.

As would be expected, proving a Hoare triple by applying the Verification Condition Generator algorithm is simpler and more mechanic than proving it directly by applying our rules and deriving the inference tree.

Chapter 5

Amortized Costs

In chapter 4 we introduced our language, a cost-aware operational semantics, an axiomatic semantics to verify cost upper-bounds, and a Verification Condition Generator algorithm to apply our logic. In this chapter, we present a variation to our logic by considering amortized analysis to refine the estimation of costs for the *while* loop. We start by giving some background on amortization before presenting our updated definitions and some examples.

5.1 Background

Amortized analysis is a method defined by Tarjan for analyzing the complexity of a sequence of operations [27]. Instead of reasoning about the worst-case cost of individual operations, amortized analysis concerns the worst-case cost of a sequence of operations. The advantage of this method is that some operations can be more expensive than others. Thus, distributing the cost of expensive operations over the cheaper ones can produce better bounds than analyzing the worst case for individual operations.

Let a_i and t_i be, respectively, the amortized cost and the actual cost of each i operation. In order to obtain an amortized analysis, it is necessary to define an *amortized cost* such that

$$\sum_{i=1}^{n} a_i \geq \sum_{i=1}^{n} t_i$$

i.e., for a sequence of n operations, the total amortized cost is an upper bound on the total actual cost.

Thus for each intermediate step, the accumulated amortized cost is an upper bound on the accumulated actual cost. This allows the use of operations with an actual cost that exceeds their amortized cost. Conversely, other operations have a lower actual cost than their amortized cost. Expensive operations can only occur when the difference between the accumulated amortized cost and the accumulated actual cost is enough to cover the extra cost.

There are three methods for amortized analysis: the *aggregate method*, the *accounting method*, and the *potential method.* Let us analyze each of the methods with the same example.

Dynamic Array Let us consider a simple dynamic array algorithm. We will perform n insertions on the array. Every time the array is full, we create a new array with double the size, and all the elements must be copied to the new array. Consider a worst-case analysis of this algorithm for n insertions. When inserting element i, we might need to resize, so one insertion might lead to a resize, which would copy $i-1$ elements to the new array, plus the cost of inserting i, which means the worst-case scenario cost of one insertion is i. The cost of inserting n elements would be $\sum_{i=1}^{n} i$, which is $\mathcal{O}(n^2)$.

Aggregate Method The aggregate method considers the worst-case execution time $T(n)$ to run a sequence of n operations. The amortized cost for each operation is $T(n)/n$. Applying the aggregate method to our dynamic array example gives us an amortized cost of $\frac{\mathcal{O}(n^2)}{n} = \mathcal{O}(n)$ per insertion.

Accounting Method The Accounting Method, sometimes also called the taxation method, is a method where we tax cheaper operations, so we always have enough saved up to cover more expensive operations without ever going out of credit. To apply the accounting method to the dynamic array, we need to define what are the cheap operations we need to tax. Let us consider we have already inserted m elements. Inserting an element in an array is $T(1)$ if the array is not full. If the array is full, we create a new array with size $2m$ and copy all m elements to this new array. In this case, the cost of inserting an element is $T(m)$ to copy all the elements, plus $T(1)$ to insert the new element. If we consider the cost of inserting as $T(3)$, 1 being the actual insertion cost we will never run out of credit. For every array state, all elements after position $m/2$, have never been copied, so they still have 2 extra credits. If we need to resize again, 1 of these credits will be used to copy the element, the remainder $m/2$ credits will be used to pay for copying elements before position $m/2$ that might have run out of credit. Since we always double the array size, this will always be enough. Therefore the amortized cost of insertion is 3, i.e. $\mathcal{O}(1)$.

Potential method The Potential Method considers a function Φ, which maps a data structure's state d_i to a real number. While with the accounting method, we would tax operations, with the potential method, credit is associated with the state of the data structure. Let d_i represent the state of the data structure after i operations, and Φ_i represent its potential. A valid potential function guarantees two properties: the initial potential is 0, $\Phi_0 = 0$; the potential always remains positive, $\forall i. \Phi_i \geq 0$. The amortized cost of an operation a_i is defined as its actual cost t_i, plus the change in potential between d_{i-1} and d_i:

$$a_i = t_i + \Phi_i - \Phi_{i-1}$$

From this, we get for j operations:

$$\sum_{i=1}^{j} t_i = \sum_{i=1}^{j}(a_i + \Phi(d_{i-1}) - \Phi(d_i))$$
$$\sum_{i=1}^{j} t_i = \sum_{i=1}^{j} a_i + \sum_{i=1}^{j}(\Phi(d_{i-1}) - \Phi(d_i))$$

The sequence of potential function values forms a telescoping series; thus, all terms except the initial and final values cancel.

$$\sum_{i=1}^{j} t_i = \sum_{i=1}^{j} a_i + \Phi(d_0) - \Phi(d_j)$$

Since $\Phi_0 = 0$ and $\Phi_j \geq 0$ then

$$\sum_{i=1}^{j} a_i \geq \sum_{i=1}^{j} t_i$$

Let us apply the potential method to the dynamic array problem. We start by defining our potential function ϕ as 2 times the number of elements after position $m/2$ for an array of size m, $\Phi_n = 2 \times (n - m/2) = 2n - m$. If the array is not full, the real cost (t_i) of inserting i in the array is 1, and the change in potential $\phi_i - \phi_{i-1}$ is 2, which means the amortized cost of this insertion is 3. If the array is full the real cost (t_i) is the cost of copying m elements plus the cost of one insertion, $t_i = m + 1$. The change in potential is $2 - m$, then the amortized cost a_i will be 3, i.e. $\mathcal{O}(1)$. This method, like the accounting method, gives us a better amortized cost than the aggregated method.

The choice of method depends on how convenient each method is to the situation. The proof rules we will show in the next section use the potential method. As we have seen in chapter 3, amortized analysis based on the potential method was already used in previous work, which derives upper bounds on the use of computational resources [8, 17, 26, 28]. Here we use it to prove tighter bounds when the composition of worst-case execution times is overly pessimistic.

5.2 Proof Rules with Amortized Costs

We now present our modified logic for amortized analysis. To this end, we modify the *while* rule (Figure 5.1) to allow deriving more precise bounds. Similarly to the *while rule* presented in figure 4.8, we still get the variant f, the invariant I, and the maximum number of iteration N from the oracle. But now, this new rule requires additional information from the oracle: an amortized cost for each iteration a and a potential function ϕ. Regarding these new values, we add the following restrictions:

- The potential function ϕ must be zero before we start the *while*, $\phi = 0$
- The potential function ϕ must remain positive after every iteration of the *while* $I \rightarrow \phi \geq 0$

$$\frac{\underline{I} \wedge \mathcal{B}\llbracket b\rrbracket \Rightarrow \underline{f} \leq \underline{N} \qquad \underline{I} \rightarrow \phi \geq 0 \\ \{\ \underline{I} \wedge \mathcal{B}\llbracket b\rrbracket\ \wedge f = k\}\ S\ \{\ \underline{I}\ \wedge \underline{f} > k \wedge \phi = P_k \mid \underline{a} + \underline{\phi} - P_k\ \}}{\{\ \underline{I}\ \wedge \underline{f} \geq 0 \wedge \phi = 0\ \}\ \text{while } b \text{ do } S\ \{\ I \wedge \neg\mathcal{B}\llbracket b\rrbracket \mid \sum_{i=0}^{N} a + \sum_{i=0}^{N+1} \mathcal{T}\mathcal{B}\llbracket b\rrbracket\}}$$

Figure 5.1: Hoare rule for *while* statement with amortized costs.

- Knowing that $\langle S, \sigma\rangle \rightarrow^t \sigma_1$, we must have that $\forall k.\mathcal{A}\llbracket a + \phi - P_k\rrbracket\sigma \geq t$

Note that we have added a logic variable P_k. This variable is used in the time assertion $a+\phi-P_k$, and allows us to refer to the value of variable ϕ in two different states, before and after executing statement S.

Soundness

Assume $I \wedge \mathcal{B}\llbracket b\rrbracket \Rightarrow f \leq N$, $I \Rightarrow \phi \geq 0$, and

$$\models \{\ I \wedge \mathcal{B}\llbracket b\rrbracket\ \wedge f = k\}\ S\ \{\ I\ \wedge \phi = P_k \wedge f > k \mid a + \phi - P_k\ \}$$

Considering a state σ such that $\sigma \models I\ \wedge\ f \geq 0\ \wedge\ \phi = 0$ and $\langle \text{while } b \text{ do } S, \sigma\rangle \rightarrow^{t_w} \sigma_1$.

Given our assumption, we know that $\sigma \models f \geq 0$ and that every time we enter the *while* body, f increases. We also know that as long as $I \wedge B[b]$ are true, $f \leq N$. Therefore we know that the program will eventually stop and at most iterate $N + 1$ times.

If $\sigma \models \neg\mathcal{B}\llbracket b\rrbracket$ then $\sigma_1 = \sigma$, therefore $\sigma_1 \models \neg\mathcal{B}\llbracket b\rrbracket \wedge I$. In this case $t_w = \mathcal{T}\mathcal{B}\llbracket b\rrbracket$. Since $N \geq 0$, then $\mathcal{A}\llbracket\sum_{i=0}^{N} a + \sum_{i=0}^{N+1} \mathcal{T}\mathcal{B}\llbracket b\rrbracket\rrbracket\sigma \geq t_w$.

If $\sigma \models \mathcal{B}\llbracket b\rrbracket$, we have that $\sigma \models \mathcal{B}\llbracket b\rrbracket \wedge I$. Considering a state σ_2 such that $\langle S, \sigma\rangle \rightarrow^{t_1} \sigma_2$ and $\langle \text{while } b \text{ do } S, \sigma_2\rangle \rightarrow^{t_2} \sigma_1$. Applying our initial assumption we get $\sigma_2 \models I \wedge \phi = P_k \wedge f > k$. Finally by applying the induction hypothesis we have that $\sigma_1 \models \neg\mathcal{B}\llbracket b\rrbracket \wedge I$.

By our assumption we also know that $\forall k.\mathcal{A}\llbracket(a + \phi - P_k)\rrbracket\sigma \geq t_1$.

By induction, we know that $\langle \text{while b do S}, \sigma_2\rangle \rightarrow^{t_2} \sigma_1$ where

$$\mathcal{A}\llbracket(N - 1) \times a + N \times \mathcal{T}\mathcal{B}\llbracket b\rrbracket\rrbracket\sigma_2 \geq t_2$$

The real cost of the *while* is given by $t_1 + t_2 + \mathcal{T}\mathcal{B}\llbracket b\rrbracket$.

$$\mathcal{A}\llbracket(a + \phi - P_k)\rrbracket\sigma \geq t_1$$

Since $\sigma \models \phi = 0 \rightarrow \mathcal{A}\llbracket\phi\rrbracket\sigma = 0$. And since $\sigma_2 \models I$ and $I \rightarrow \phi \geq 0$, then $\sigma \models \phi \geq 0$. $P_k = \mathcal{A}\llbracket\phi\rrbracket\sigma_2$, then $P_k \geq 0$. Then we have

$$\mathcal{A}[\![a]\!]\sigma \geq t_1$$

Since a, N, and $\mathcal{TB}[\![b]\!]$ are all constant, we get

$$\mathcal{A}[\![N \times a + N \times \mathcal{TB}[\![b]\!]]\!]\sigma \geq t_1 + t_2$$
$$\mathcal{A}[\![N \times a + (N+1) \times \mathcal{TB}[\![b]\!]]\!]\sigma \geq t_1 + t_2 + \mathcal{TB}[\![b]\!]$$

Therefore the rule for *while* is sound.

$$\models \{\ I\ \ \wedge f \geq 0\ \}\ \text{while}\ b\ \text{do}\ S\ \{\ I \wedge \neg\mathcal{B}[\![e]\!]\ |\ N \times a + (N+1) \times \mathcal{TB}[\![b]\!]\}$$

Example: Binary Counter

We now illustrate this new version of the Hoare Logic on a typical application of amortized analysis: a binary counter. In the binary counter algorithm, we represent a binary number as an array of zeros and ones. We start with an array with every value at zero, and with each iteration, we increase the number by one until we reach the desired value. Our implementation can be seen in figure 5.2.

$\{n \geq 0 \wedge size = log(n) \wedge \forall i.0 \leq i \wedge i \leq size \Rightarrow B[i] = 0\}$

```
i = 0;
while i < n do
   j = 0;
   while B[j] == 1 do
      B[j] = 0;
      j = j + 1
   end;
   B[j] = 1;
   i = i + 1
end
```

$\{n = \sum_{i=0}^{log(n)-1} B[i] \times 2^i | 30n + 30\}$

Figure 5.2: Binary Counter Implementation with Annotation.

To start our proof we will define the oracle information for each of the two *while* loops. For simplicity, we refer to the external *while* loop as *while* 1, and the internal as *while* 2.

The invariant of *while* 1, $I_1 \equiv i = \sum_{k=0}^{size-1} B[k] \times 2^k \wedge 0 \leq i \leq n$, the variant $f_1 \equiv i$, the maximum number of iterations $N_1 \equiv n$, the amortized cos $a_1 \equiv 20$, and the potential function $\phi_1 \equiv fun\ k \rightarrow \sum_{i=0}^{size} B[i]$. Here log(n) is the base two logarithm of n.

For the *while* 2 we have $I_2 \equiv \forall k.0 \leq k < j \to B[k] = 0 \wedge 0 \leq j \leq size$, $f_2 \equiv j$, $N_2 \equiv size$, $a_2 \equiv 10$, and $\phi_2 \equiv fun\ k \to 1$.

By the *assign rule*, we have

$$\vdash \{(I_1 \wedge f_1 \geq k \wedge \phi_1 = P_k)[i+1/i]\}i = i+1\{I_1 \wedge f_1 \geq k \wedge \phi_1 = P_k|4\}$$

By the *assign rule*, we have

$$\vdash \{(I_1 \wedge f_1 \geq k \wedge \phi_1 = P_k)[i+1/i][1/B[j]]\}B[j] = 1\{(I_1 \wedge f_1 \geq k \wedge \phi_1 = P_k)[i+1/i]|3\}$$

Applying the *seq rule*, we get

$$\vdash \{(I_1 \wedge f_1 \geq k \wedge \phi_1 = P_k)[i+1/i][1/B[j]]\}B[j] = 1; i = i+1\{I_1 \wedge f_1 \geq k \wedge \phi_1 = P_k|7\}$$

By the *assign rule*, we get

$$\vdash \{(I_2 \wedge f_2 \geq k \wedge \phi_2 = P_k)[j+1/j]\}j = j+1\{I_2 \wedge f_2 \geq k \wedge \phi_2 = P_k|4\}$$

By the *assign rule*, we get

$$\vdash \{(I_2 \wedge f_2 \geq k \wedge \phi_2 = P_k)[j+1/j][0/B[j]]\}B[j] = 0\{(I_2 \wedge f_2 \geq k \wedge \phi_2 = P_k)[j+1/j]|3\}$$

By the *seq rule*

$$\vdash \{(I_2 \wedge f_2 \geq k \wedge \phi_2 = P_k)[j+1/j][0/B[j]]\}B[j] = 0; j = j+1\{I_2 \wedge f_2 \geq k \wedge \phi_2 = P_k|7\}$$

Since we know $I_2 \wedge B[j] = 1 \wedge f_2 = k \to (I_2 \wedge f_2 \geq k \wedge \phi_2 = P_k)[j+1/j][0/B[j]] \wedge a_2 + \phi_2 - Pk \geq 7$, then by the *weak rule*

$$\vdash \{I_2 \wedge B[j] = 1 \wedge f_2 = k\}B[j] = 0; j = j+1\{I_2 \wedge f_2 > k \wedge \phi_2 = P_k|a_2 + \phi_2 - P_k\}$$

Then by the *while rule*, we get

$$\vdash \{I_2 \wedge f_2 \geq 0\}while_2\{I_2 \wedge \neg(B[j] = 1)|N_2 \times a_2 + (N_2+1) \times \mathcal{TB}\llbracket B[j] = 1\rrbracket\}$$

Since $I_2 \wedge \neg(B[j] = 1) \to (I_1 \wedge f_1 \geq k \wedge \phi_1 = P_k)[i+1/i][1/B[j]]$, then by the *weak rule*

$$\vdash \{I_2 \wedge f_2 \geq 0\}while_2\{(I_1 \wedge f_1 \geq k \wedge \phi_1 = P_k)[i+1/i][1/B[j]]|N_2 \times a_2 + (N_2+1) \times \mathcal{TB}\llbracket B[j] = 1\rrbracket\}$$

By the *seq rule*

$$\vdash \{I_2 \wedge f_2 \geq 0\}while_2; B[j] = 1; i = i+1\{I_1 \wedge f_1 \geq k \wedge \phi_1 = P_k|N_2 \times a_2 + (N_2+1) \times \mathcal{TB}\llbracket B[j] = 1\rrbracket + 7\}$$

By the *assign rule*, we have

$$\{(I_2 \wedge f_2 \geq 0)[0/j]\}j = 0\{I_2 \wedge f_2 \geq 0|2\}$$

By the *seq rule*

$$\vdash \{(I_2 \wedge f_2 \geq 0)[0/j]\}j = 0; while_2; B[j] = 1; i = i+1\{I_1 \wedge f_1 \geq k \wedge \phi_1 = P_k|N_2 \times a_2 + (N_2+1) \times \mathcal{TB}\llbracket B[j] = 1\rrbracket + 9\}$$

Since $I_1 \wedge i < n \wedge f_1 = k \rightarrow (I_2 \wedge f_2 \geq 0)[0/j]$, and $a_1 + \phi_1 - P_k \geq N_2 \times a_2 + (N_2+1) \times \mathcal{TB}\llbracket B[j] = 1\rrbracket + 9$, then by the *weak rule*

$$\vdash \{I_1 \wedge i < n \wedge f_1 = k\}j = 0; while_2; B[j] = 1; i = i + 1\{I_1 \wedge f_1 \geq k \wedge \phi_1 = P_k|a_1 + \phi_1 - P_k\}$$

Since $I_1 \wedge i < n \rightarrow f_1 \leq N_1$, and $I_1 \rightarrow \phi_1 \geq 0$, then we can apply the *while rule* and we get

$$\vdash \{I_1 \wedge f_1 \geq 0 \wedge \phi_1 = 0\}while_1\{I_1 \wedge \neg(i < n)|N_1 \times a_1 + (N_1 + 1) \times 3\}$$

By the *assign rule*

$$\vdash \{(I_1 \wedge f_1 \geq 0 \wedge \phi_1 = 0)[0/i]\}i = 0\{I_1 \wedge f_1 \geq 0 \wedge \phi_1 = 0|2\}$$

Applying the *seq rule* gives us

$$\vdash \{(I_1 \wedge f_1 \geq 0 \wedge \phi_1 = 0)[0/i]\}i = 0; while_1\{I_1 \wedge \neg(i < n)|2 + N_1 \times a_1 + (N_1 + 1) \times 3\}$$

Since $n \geq 0 \wedge size = log(n) \rightarrow (I_1 \wedge f_1 \geq 0 \wedge \phi_1 = 0)[0/i]$, $I_1 \wedge \neg(i < n) \rightarrow n = \sum_{i=0}^{log(n)-1} B[i] \times 2^i$, and $40n + 10n + 30 \geq 2 + N_1 \times a_1 + (N_1 + 1) \times 3$, by the *weak rule*

$$\vdash \{n \geq 0 \wedge size = log(n)\}i = 0; while_1\{n = \sum_{i=0}^{log(n)-1} B[i] \times 2^i|40n + 10n + 30\}$$

5.3 Verification Conditions Generation with Amortized Costs

Considering the extensions to our logic, as presented and explained in the last section, we need to modify our VCG accordingly. The only changes we need to perform in wpc and VC are for the while cases. Both these changes are shown in Figure 5.3.

The new wpc rule for *while* returns a new term in the invariants conjunction that stipulates that $\phi = 0$, i.e., the potential function must be zero before the *while* begins. The upper bound is given by the sum of the amortized cost for every iteration $\sum_{i=0}^{N} a$, plus the sum of every $N + 1$ evaluation of b, $\sum_{i=0}^{N+1} \mathcal{TB}\llbracket b\rrbracket$.

The VC rule for *while* generates four VCs:

$$
\begin{aligned}
\mathbf{wpc}(\text{while } b \text{ do } S, Q) \;&=\; (\underline{I} \wedge \underline{f} \geq 0 \wedge \underline{\phi} = 0\ ,\ N \times a + (N+1) \times \mathcal{TB}\llbracket B \rrbracket \\[1em]
\mathbf{VC}(\text{while } b \text{ do } S, Q) \;&=\; \{(\underline{I} \wedge \mathcal{B}\llbracket b \rrbracket \wedge \underline{f} = \underline{k}) \rightarrow wp_S \wedge a + \Phi - P_k \geq t_S\} \cup \\
&\quad\; \{(\underline{I} \wedge \neg B) \rightarrow Q\} \cup \\
&\quad\; \{\underline{I} \wedge \mathcal{B}\llbracket b \rrbracket \Rightarrow \underline{f} \leq \underline{N}\} \cup \\
&\quad\; \{\underline{I} \rightarrow \phi \geq 0\} \\
&\quad\; \mathbf{VC}(S, \underline{I} \wedge \underline{f} > k \wedge \underline{\phi} = P_k) \cup \\[1em]
&\quad\; \text{where } wp_S, t_S = \mathbf{wpc}(S, \underline{I} \wedge \underline{f} > \underline{k} \wedge \phi = P_k)
\end{aligned}
$$

Figure 5.3: VCG rules for *while* statement with amortized costs.

- $\forall k.(I \wedge B \wedge f = k) \rightarrow wpc(S, I \wedge f > k \wedge \phi = P_k)$ guarantees both the invariant, the variant increase with every iteration, and defines a logic variable P_k which allows us to refer to the state of variable ϕ in the postcondition.
- $I \wedge \neg b \rightarrow Q$ which states that at the end of the *while* (when b is false), the invariant and $\neg b$ imply the postcondition Q we wish to prove.
- $I \wedge \mathcal{B}\llbracket b \rrbracket \rightarrow f \leq N$, which states that while the *while* is still running (when b is true), the variant is always less or equal to N.
- $I \rightarrow \phi \geq 0$, the potential function is always positive.

This rule also makes a recursive call to VC for the loop's body.

Soundness

Proof. We start by defining our Induction Hypothesis.

$$\models VCG(\{P\}S\{Q|T\}) \rightarrow \vdash \{P\}S\{Q|T\}$$

We also calculate the result of wpc, VC and VCG for *while.*

$$wpc(\text{while } b \ do \ S, Q) = (I \wedge f \geq 0 \wedge \Phi = 0, N \times a + (N+1) \times \mathcal{TB}\llbracket b \rrbracket)$$

$$wp_S, t_S = wpc(S, I \wedge f > k \wedge \phi = P_k)$$

$$\begin{aligned} VC(\text{while } b \textit{ do } S, Q) = &\{(I \land \mathcal{B}[\![b]\!] \land f = k) \to wp_S\} \cup & (5.1)\\ &\{I \land \mathcal{B}[\![b]\!] \land f = k \to a + \Phi - P_k \geq t_s\} \cup & (5.2)\\ &\{I \land \neg\mathcal{B}[\![b]\!] \to Q\} \cup & (5.3)\\ &\{I \land \mathcal{B}[\![b]\!] \to f \leq N\} \cup & (5.4)\\ &\{I \to \Phi \geq 0\} \cup & (5.5)\\ &VC(S, I \land f > k \land \Phi = P_k) & (5.6) \end{aligned}$$

$$\begin{aligned} VCG(\{P\}\text{while } b \textit{ do } S\{Q|T\}) = &\{P \to (I \land f \geq 0 \land \Phi = 0)\} \cup & (5.7)\\ &\{P \to T \geq N \times a + (N+1) \times \mathcal{TB}[\![b]\!]\} \cup & (5.8)\\ &VC(\text{while } b \textit{ do } S, Q) & (5.9) \end{aligned}$$

Assuming $\models VCG(\{P\}\text{while } b \textit{ do } S\{Q|T\})$.

Since we have 5.1, 5.2, and 5.6, then

$$\models VCG(\{I \land \mathcal{B}[\![b]\!] \land f = k\}S\{I \land f > k \land \Phi = P_k | a + \Phi - P_k\})$$

By the induction hypothesis

$$\vdash \{I \land \mathcal{B}[\![b]\!] \land f = k\}S\{I \land f > k \land \Phi = P_k | a + \Phi - P_k\} \tag{5.10}$$

Given 5.4, 5.5, and 5.10, by the *while rule*

$$\vdash \{I \land f \geq 0 \land \Phi = 0\}\text{while } b \textit{ do } S\{I \land \neg\mathcal{B}[\![b]\!] | N \times a + (N+1) \times \mathcal{TB}[\![b]\!]\}$$

Given 5.3, 5.7, and 5.8, by the *weak rule*

$$\vdash \{P\}\text{while } b \textit{ do } S\{Q|T\}$$

□

Example: Binary Counter

Let us apply the VCG algorithm to the binary counter example 5.2. We will use the same notation as in section 5.2 and refer to our precondition as P, our program as S, our postcondition as Q and our cost upperbound as T.

We start by calling the wpc function.

$$wpc(while_1, Q) = (I_1 \wedge I \geq 0 \wedge \phi_1 = 0, n \times a_1 + (n+1) \times 3)$$

$$wpc(i = 0, I_1 \wedge i \geq 0 \wedge \phi_1 = 0) = ((I_1 \wedge i \geq 0 \wedge \phi_1 = 0)[0/i], 2)$$

$$wp_S, t_S = wpc(S, Q) = ((I_1 \wedge i \geq 0 \wedge \phi_1 = 0)[0/i], 2 + n \times a_1 + (n+1) \times 3)$$

Then we call the VC function for our program. Since this function only generates extra VCs for *while* loops, we will omit other calls to the VC function for simplicity.

$$\begin{aligned}
&VC(S,Q) = VC(while_1, Q) = \\
&\quad \{(I_1 \wedge i < n \wedge i = k) \rightarrow wp_1\} \cup && (5.11)\\
&\quad \{(I_1 \wedge i < n \wedge i = k) \rightarrow 30 + \phi_1 - P_k \geq t_1\} \cup && (5.12)\\
&\quad \{(I_1 \wedge \neg i < n) \rightarrow Q\} \cup && (5.13)\\
&\quad \{I_1 \wedge i < n \Rightarrow i \leq n\} \cup && (5.14)\\
&\quad \{I_1 \rightarrow \phi_1 \geq 0\} \cup && (5.15)\\
&\quad \mathbf{VC}(S_w, I_1 \wedge i > k \wedge \phi_1 = P_k)
\end{aligned}$$

Where $S_w \equiv j = 0; while_2; B[j] = 1; i = i + 1$, and

$$\begin{aligned}
wp_1, t_1 &= \mathbf{wpc}(S_w, I_1 \wedge i > k \wedge \phi_1 = P_k)\\
&= (I_2 \wedge j \geq 0 \wedge \phi_2 = 0)[0/j], size \times a_2 + (size + 1) \times \mathcal{TB}[\![B[j] = 1]\!]
\end{aligned}$$

$$\begin{aligned}
&VC(S_w, I_1 \wedge i > k \wedge \phi_1 = P_k) = \\
&VC(while_2, (I_1 \wedge i > k \wedge \phi_1 = P_k)[i+1/i][1/B[j]]) = \\
&\quad \{(I_2 \wedge B[j] = 1 \wedge j = k) \rightarrow wp_2\} \cup && (5.16)\\
&\quad \{(I_2 \wedge B[j] = 1 \wedge j = k) \rightarrow a_2 + \phi_2 - P_K \geq t_2\} \cup && (5.17)\\
&\quad \{(I_2 \wedge \neg B[j] = 1) \rightarrow (I_1 \wedge i > k \wedge \phi_1 = P_k)[i+1/i][1/B[j]]\} \cup && (5.18)\\
&\quad \{I_2 \wedge B[j] = 1 \Rightarrow j \leq size\} \cup && (5.19)\\
&\quad \{I_2 \rightarrow \phi_2 \geq 0\} && (5.20)
\end{aligned}$$

Where

$$\begin{aligned}
wp_2, t_2 &= \mathbf{wpc}(B[j] = 0; j = j + 1, I_2 \wedge j > k \wedge \phi_2 = P_k)\\
&= ((I_2 \wedge j \geq 0 \wedge \phi_2 = 0)[j+1/j][0/B[j]], size \times a_2 + (size + 1) \times 3 + 9)
\end{aligned}$$

Finally we call the VCG function

$$
\begin{aligned}
VCG(\{P\}S\{Q|T\}) = & \{P \rightarrow wp_S\} \cup && (5.21)\\
& \{P \rightarrow T \geq t_S\} \cup && (5.22)\\
& VC(S, Q)
\end{aligned}
$$

To prove our triple, we now simply need to prove all of the VCs generated by the algorithm (5.11 to 5.21), this can easily be done for all the conditions manually, or with the assistance of a theorem prover.

As would be expected, proving a Hoare triple by applying the VCG algorithm is simpler and more mechanic than proving it directly by applying our rules and deriving the inference tree.

Chapter 6

Exact Costs

The final part of our work consists of extending our language and logic so that we can verify the exact cost of a program instead of an upper bound like in the previous chapters. This approach is advantageous in scenarios where the approximation of execution time is not enough to guarantee safety, as it happens for critical systems and real-time programming. Moreover, proving that the exact execution time of a program is an expression that does not depend on confidential data provides a direct way to prove the absence of timing leakage, which is relevant in cryptographic implementations. We must restrict the programming language to guarantee the ability to prove exact costs. Thus, in this third scenario, programs have bound recursion (*for* loops), and both branches of conditional statements must have identical costs.

6.1 Operational Semantics for Exact Costs

Since we need to be able to calculate exact costs, we need to make some extensions to our language so that the execution time is fully deterministic. The first one is a restriction to our conditional statement, if b then S_1 else S_2. It is still possible to have conditional statements in this scenario. However, we need to guarantee that both branches of the *if* will take the same time to run.

In our original language, we used *while* loops, but since we can not predict accurately the exact amount of times a *while* is going to run, we can not have this statement in this version. We will then replace our *while* loop with a *for* loop, which executes a deterministic number of times, solving our problem.

Let us now present our updated syntax rules for statements, which remain the same for every statement except the *for* loop.

$$S ::= skip \mid x = a \mid x[a_1] = a_2 \mid \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ done} \mid \textbf{for } i = a \textbf{ to } b \textbf{ do } S \textbf{ done} \mid S_1; S_2$$

The semantic rules for *for* loop are presented in figure 6.1. We have one rule and one axiom for *for* loop, $[for^{true}]$ and $[for^{false}]$.

If $\mathcal{B}\llbracket b\rrbracket\sigma$ is false, we apply the axiom $[for^{false}]$, that says we will remain in the same state σ and the cost is simply the cost of the evaluation of $a < b$, $\mathcal{TB}\llbracket a < b\rrbracket$.

If $\mathcal{B}\llbracket b\rrbracket\sigma$ is true, we apply rule $[for^{true}]$, which means we will, assign a to variable i, which will lead to a state σ''', we will then execute the loop body, S, once from state σ''' and this will lead to a state σ''. Finally, we execute the *for* loop again, but this time starting at $a + 1$ and from state σ''. The cost of the *for* loop, in this case, is the cost of evaluating $a < b$, plus the cost of executing the *assign*, plus the cost of executing the body, plus the cost of executing the *for* loop from state σ''.

$$[for^{\text{true}}] \qquad \frac{\langle i = a, \sigma\rangle \to^{t_1} \sigma''' \qquad \langle S, \sigma'''\rangle \to^{t_2} \sigma'' \qquad \langle \text{for } i = a + 1 \text{ to } b \text{ do } S, \sigma''\rangle \to^{t_3} \sigma'}{\langle \text{for } i = a \text{ to } b \text{ do } S, \sigma\rangle \to^{\mathcal{TB}\llbracket a<b\rrbracket + t_1 + t_2 + t_3} \sigma'} \qquad \text{if } \mathcal{B}\llbracket a < b\rrbracket\sigma = \text{true}$$

$$[for^{\text{false}}] \qquad \langle \text{for } i = a \text{ to } b \text{ do } S, \sigma\rangle \to^{\mathcal{TB}\llbracket a<b\rrbracket} \sigma \qquad \text{if } \mathcal{A}\llbracket \neg(a < b)\rrbracket\sigma = \text{false}$$

Figure 6.1: Operational semantic of *for* loop.

The semantic rules for *skip*, *assign*, *array*, *seq* and *if* remain the same as presented in section 4.2.

6.2 Proof Rules for Exact Costs

In this new logic we have that $\models \{P\}S\{Q|t\}$ if and only if, for all state σ such that $\sigma \models P$ and $\langle S, \sigma\rangle \to^{t} \sigma'$ we have that $\sigma' \models Q$ and $\mathcal{A}\llbracket t\rrbracket\sigma = t'$. Notice that this is fairly similar to what was presented in section 4.3 but now, for a Hoare triple to be valid, the value passed in the cost section needs to represent the exact cost of the program ($\mathcal{A}\llbracket t\rrbracket\sigma = t'$).

The new axiomatic rules for the *for* loop are defined in figure 6.2.

$$\frac{\{P \wedge a \leq i \wedge i < b\}\ S\ \{P[i+1/i] \mid t_S\}}{\{P[a/i] \wedge a < b\} \text{ for } i = a \text{ to } b \text{ do } S\ \{P[b/i] \mid (b-a) \times (\mathcal{TA}\llbracket a\rrbracket + C_{ASSIGN\_V} + t_S) + (b-a+1) \times (\mathcal{TB}\llbracket a < b\rrbracket)\}}$$

$$\{P \wedge b \leq a\} \text{ for } i = a \text{ to } b \text{ do } S\ \{P|\mathcal{TB}\llbracket a < b\rrbracket\}$$

Figure 6.2: Hoare rule for *for*-loop statement.

We have one rule and one axiom for for. The axiom says that if we are in a state that validates P and $b \leq a$, then after executing the *for*, we will be in a state that validates P, and this execution will have an exact cost of $\mathcal{TB}\llbracket a < b\rrbracket$. The for rule says that if we start in a state that validates P and $a < b$ then after executing the *for* loop we will be in a state that validates $P[b/i]$ and this will have a cost of $(b - a + 1) \times \mathcal{TB}\llbracket a < b\rrbracket + (b - a) \times (t_S + \mathcal{TA}\llbracket a\rrbracket + C_{ASSIGN\_V})$. To prove this, we must first prove that if we execute S from a state that validates P and $a \leq i < b$, then after executing S, we will be in a state that validates $P[b/i]$ and this will cost t_S to execute.

We also modify the rule for conditional statements by imposing that both branches must execute with the same exact cost. The rule for *if* is then redefined as shown in Figure 6.3. Note that balancing *if* branches with, e.g., dummy instructions is a common technique used in cryptography to eliminate execution time dependencies from branch conditions that may be related to secret data. The new rule is shown in Figure 6.3.

$$\frac{\{\ P \wedge b\ \}\ S_1\ \{\ Q \mid t\} \qquad \{\ P \wedge \neg b\ \}\ S_2\ \{\ Q \mid t\ \}}{\{\ P\ \}\ \text{if } b \text{ then } S_1 \text{ else } S_2\ \{\ Q \mid t + \mathcal{TB}[\![b]\!]\ \}}$$

Figure 6.3: Hoare rule for *if* statement where both branches take exactly the same time to execute.

The rule for if says that, if we start at a state validating P then after executing if we will arrive to a state that validates Q. This execution will take exactly $t + \mathcal{TB}[\![b]\!]$. To prove this, we need to guarantee

- Executing S_1 from a state validating $P \wedge b$ generates a state that validates Q and S_1 takes t to execute.
- Executing S_2 from a state validating $P \wedge \neg b$ generates a state that validates Q and S_2 takes t to execute.

Soundness

We need to ensure that our Hoare logic is sound with respect to our operational semantic. For this version, our soundness theorem will be slightly different than the one we have previously presented.

Theorem 4 (Soundness)**.** *We have that* $\models \{P\}S\{Q|t\}$ *if and only if, forall state* σ *such that* $\sigma \models P$ *and* $\langle S, \sigma\rangle \to^{t'} \sigma'$*, we have* $\sigma' \models Q$ *and* $\mathcal{A}[\![t]\!]\sigma = t'$*.*

Even though our theorem changed, the proof for *skip*, *assign*, *array* and *seq* will look exactly the same since the upper bound calculated by our previous logic was already identical to the real cost of execution. Therefore we will only show the proof for *if* and *for*.

Proof. Case if: Assume $\models \{P \wedge b\}S_1\{Q|t\}$ and $\models \{P \wedge \neg b\}S_2\{Q|t\}$. Suppose $\sigma \models P$.

If $\sigma \models b$, then $\sigma \models P \wedge b$ so, assuming $\langle S_1, \sigma\rangle \to^{t_1} \sigma_1$, we have that $\sigma_1 \models Q$

If $\sigma \models \neg b$, then $\sigma \models P \wedge \neg b$ e so, assuming $\langle S_2, \sigma\rangle \to^{t_2} \sigma_2$, we have that $\sigma_2 \models Q$.

Given our assumptions we know that $t_1 = t_2 = t$. We then have that the exact cost for the *if* statement is $t + \mathcal{TB}[\![b]\!]$. The rule for *if* is sound.

$$\models \{\ P\ \} \text{ if } e \text{ then } S_1 \text{ else } S_2\ \{\ t_1 + \mathcal{TB}[\![b]\!] \mid Q\ \}$$

Case for: Assume $\models \{\ P \wedge (a \leq i) \wedge i < b\ \}\ S\ \{\ P[i+1/i]\ |t_S\}$
Suppose $\sigma \models P[a/i]$ and $\langle \text{for } i = a \text{ to } b \text{ do } S, \sigma\rangle \rightarrow^t \sigma_1$.

If $\sigma \models \neg(a \leq b)$ then $\sigma = \sigma_1$. From our lemma 1 we get that $\sigma \models P$. In this case, the execution time is $\mathcal{TB}[\![a \leq b]\!]$. When $a > b$, $t = \mathcal{TB}[\![a < b]\!]$, then the axiom is sound.

If $\sigma \models a < b$ then $\sigma \models P[a/i] \wedge a < b$. Let us consider a state σ_2 such that $\langle i = a, \sigma\rangle \rightarrow^{t_1} \sigma_2$. By the *assign rule*, we know $\sigma_2 \models P$ and $t_1 = \mathcal{TA}[\![a]\!] + C_{ASSIGN\_V}$. We also have that $\sigma_2 \models a \leq i < b$, therefore if we execute S from state σ_2, we will get a state σ_3, $\langle S, \sigma_2\rangle \rightarrow^{t_2} \sigma_3$, such that $\sigma_3 \models P[i+1/i]$, and $t_S = t_2$. By our induction hypothesis: $\langle \text{for } i = a+1 \text{ to } b \text{ do } S, \sigma_3\rangle \rightarrow^{t_3} \sigma_1$, where $\sigma_1 \models P[b/i]$, and $t_3 = (b-a) \times \mathcal{TB}[\![a < b]\!] + (b-a-1) \times (t_S + \mathcal{TA}[\![a]\!] + C_{ASSIGN\_V)}$. The real cost of executing statement for from state σ is $t = t_1 + t_2 + t_3 + \mathcal{TB}[\![a < b]\!]$. Knowing $t_1 = \mathcal{TA}[\![a]\!] + C_{ASSIGN\_V}$, $t_2 = t_S$, and $t_3 = (b-a) \times \mathcal{TB}[\![a < b]\!] + (b-a-1) \times (t_S + \mathcal{TA}[\![a]\!] + C_{ASSIGN\_V)}$, gives us

$$t = \mathcal{TA}[\![a]\!] + C_{ASSIGN\_V} + t_S + (b-a) \times \mathcal{TB}[\![a < b]\!] + (b-a-1) \times (t_S + \mathcal{TA}[\![a]\!] + C_{ASSIGN\_V)} + \mathcal{TB}[\![a < b]\!]$$

$$t = (b-a+1) \times \mathcal{TB}[\![a < b]\!] + (b-a) \times (t_S + \mathcal{TA}[\![a]\!] + C_{ASSIGN\_V)} + \mathcal{TB}[\![a < b]\!]$$

Therefore the rule for *for* is sound. □

Example: Range Filter

To illustrate our logic, we will apply our rules to the range filter algorithm, as presented in figure 6.4. This algorithm consists of a simple filter where, given an array (a) and a range [l..u], we use an auxiliary array (b) to filter the elements in a that are within the range.

We provide the invariant $I \equiv \forall k.(0 \leq k \wedge k < i) \rightarrow (l \leq a[k] \wedge a[k] \leq u \rightarrow \exists k.b[j] = a[i]) \wedge (l > a[k] \wedge a[k] > u \rightarrow \neg(\exists j.b[k] = a[i]))$ in order to prove correctness.

Let us refer to the *if* statement as S_{if} and to the *for* statement as S_{for}.

By the *assign rule* we have

$$\{I[i+1/i][j+1/j]\}j = j+1\{I[i+1/i]|\mathcal{TA}[\![j+1]\!]+1\}$$

By the *assign rule* we also have

$$\{I[i+1/i][j+1/j][a[i]/b[j]]\}b[j] = a[i]\{I[i+1/i][j+1/j]|\mathcal{TA}[\![i]\!] + \mathcal{TA}[\![j]\!] + 1\}$$

$\{0 \leq l \land l < u \land n \geq 0\}$

```
j = 0;
for i=0 to n do
  if (l <= a[i] and a[i] <= u)
  then
    b[j] = a[i];
    j = j + 1
  else
    b[j] = b[j];
    j = j + 0
  end
end
```

$\{\forall i.(0 \leq i < n) \rightarrow (l \leq a[i] \leq u \rightarrow \exists j.B[j] = a[i])\land$
$(l > a[i] > u \rightarrow \neg(\exists j.b[j] = a[i]))|17 \;\times n + 22\ \}$

Figure 6.4: Array filtering algorithm with annotations for exact cost.

By the *seq rule* we get

$$\{I[i+1/i][j+1/j][a[i]/b[j]]\}b[j] = a[i]; j = j+1\{I[i+1/i]|4+3+1\}$$

By the *assign rule* we have

$$\{I[i+1/i][j+0/j]\}j = j+0\{I[i+1/i]|\mathcal{TA}\llbracket j+0\rrbracket + 1\}$$

By the *assign rule* we also have

$$\{I[i+1/i][j+0/j][b[j]/b[j]]\}b[j] = b[j]\{I[i+1/i][j+0/j]|\mathcal{TA}\llbracket j\rrbracket + \mathcal{TA}\llbracket j\rrbracket + 1\}$$

By the *seq rule* we get

$$\{I[i+1/i][j+0/j][b[j]/b[j]]\}b[j] = b[j]; j = j+0\{I[i+1/i]|4+3+1\}$$

Since $I[0/i] \land l \leq a[i] \leq u \rightarrow I[i+1/i][j+1/j][a[i]/b[j]]$, $I[0/i] \land \neg(l \leq a[i] \leq u) \rightarrow I[i+1/i][j+0/j][b[j]/b[j]]$, then by the *if rule*

$$\{I[0/i]\}S_{if}\{I[i+1/i]|8 + \mathcal{TB}\llbracket l \leq a[i] \land a[i] \leq u\rrbracket\}$$

By the *for rule* we get

$$\{I[0/i]\}S_{for}\{I[b/i]|n \times 17 + (n+1) \times \mathcal{TB}\llbracket i < n\rrbracket\}$$

By the *assign rule*

$$\{I[0/i][0/j]\}j = 0\{\ I[b/i]|2\}$$

By the *seq rule*

$$\{I[0/i][0/j]\}S\{I[i+1/i]|8+n\times 9+(n+1)\times \mathcal{TB}\llbracket i<n\rrbracket+2\}$$

Since $P\rightarrow I[0/i][0/j]$, $I[i+1/i]\rightarrow Q$, and $T\geq n\times 9+(n+1)\times \mathcal{TB}\llbracket i<n\rrbracket+2$, then by the *weak rule*

$$\vdash \{P\}S\{Q|T\}$$

6.3 Verification Conditions Generation for Exact Costs

Given the extensions to our logic, we must rewrite our VCG accordingly.

In figure 6.5 we present the new Weakest Precondition-Cost (wpc) algorithm for if and for. The rules for $skip$, $assign$, $array$, and seq remain the same since they already output exact costs.

$$\begin{aligned}
\mathbf{wpc}(\text{if } B \text{ then } S_1 \text{ else } S_2,\psi) &= ((B\rightarrow wp_1)\wedge(\neg B\rightarrow wp_2)\wedge t_1=t_2\ ,\ t_1+\mathcal{TB}\llbracket B\rrbracket)\\
&\quad \textit{where } (wp_1\ ,\ t_1)=\mathbf{wpc}(S_1,\psi),\\
&\quad (wp_2\ ,\ t_2)=\mathbf{wpc}(S_2,\psi),\\
\mathbf{wpc}(\text{for } i=a \text{ to } b \text{ do } S,\psi) &= (\underline{I}[a/i],\ (b-a)\times(\mathcal{TA}\llbracket a\rrbracket+C_{ASSIGN\_V}+t)+(b-a+1)\times\mathcal{TA}\llbracket a<b\rrbracket)\\
&\quad \textit{where } (wp,t)=\mathbf{wpc}(S,\underline{I}[i+1/i])
\end{aligned}$$

Figure 6.5: Weakest Precondition Algorithm for Exact Costs.

In the *wpc* result for if, we add a precondition restriction that says $t_1=t_2$. In the cost expression, instead of computing the max between t_1 and t_2, we can simply define the cost as $t_1+\mathcal{TB}\llbracket b\rrbracket$.

The Weakest Precondition of a *for* loop is the invariant when $i=a$. The cost of executing a *for* loop is $b-a$ times the cost of the loop body t, plus $b-a+1$ times the cost of evaluating $a<b$.

In figure 6.6 we show the VC rules for if and for. The rule for if remains exactly the same as in the original version (4.10).

The rule for for derives three VCs:

- $I[b/i]\rightarrow Q$, meaning that when i reaches value b the loop breaks and the postcondition Q is met.
- $I\wedge a\leq i<b\rightarrow wp(S,I)$, which guarantees the invariant is preserved and that before executing the *for* loop body, i must be a value between a and b.

- $I \land \neg(a < b) \to Q$, which states that if $a < b$ is not met, then the loop will not execute and the postcondition Q is true.

The rule also has a recursive call to VC applied to the body statement S.

$$
\begin{aligned}
\mathbf{VC}(\text{if } B \text{ then } S_1 \text{ else } S_2, Q) &= \mathbf{VC}(S_1, Q) \ \cup\ \mathbf{VC}(S_2, Q) \\[1em]
\mathbf{VC}(\text{for } i = a \text{ to } b \text{ do } S, Q) &= \{\underline{I}[b/i] \to Q\} \cup \\
&\quad \{(\underline{I} \land a \leq i < b) \to wp(S, \underline{I}[i+1/i])\} \cup \\
&\quad \{(\underline{I} \land \neg(a < b)) \to Q\} \cup \\
&\quad \mathbf{VC}(S, \underline{I}[i+1/i])
\end{aligned}
$$

Figure 6.6: VC Function for Exact Costs.

Soundness

We need to prove theorem 1, which states that the VCG algorithm is sound if the Verification Condition generated implies the Hoare triple we wish to prove,

$$\models VCG(\{P\}Q\{R\}) \implies \vdash \{P\}Q\{R\}$$

Proof. We prove $\implies$ by induction on the structure of Q and $\Leftarrow$ by induction in the derivation of $\vdash \{P\}Q\{R\}$.

Case if:

Induction Hypothesis:

$$\models VCG(\{P\}S_1\{Q|t_1\} \to \vdash \{P\}S_1\{Q|t_1\})$$

$$\models VCG(\{P\}S_2\{Q|t_2\} \to \vdash \{P\}S_2\{Q|t_2\})$$

Let us consider

- $wp_1, t_1 = wpc(S_1, Q)$
- $wp_2, t_2 = wpc(S_2, Q)$
- $wpc(\text{if } b \text{ then } S_1 \text{ else } S_2, Q) = (\mathcal{B}[\![b]\!] \to wp_1 \land \neg\mathcal{B}[\![b]\!] \to wp_2 \land t_1 = t_2, t_1 + \mathcal{TB}[\![b]\!])$

$$
\begin{aligned}
VCG(\{P\}\text{if } b \text{ then } S_1 \text{ else} S_2\{Q|T\}) = &\{P \to (b \to wp_1 \land \neg b \to wp_2 \land t_1 = t_2)\} \cup \\
&\{T = t_1 + \mathcal{TB}[\![b]\!]\} \cup \\
&VC(\text{if } b \text{ then } S_1 \text{ else} S_2, Q)
\end{aligned}
$$

Where $VC(\text{if } b \text{ then } S_1 \text{ else} S_2, Q) = VC(S_1, Q) \cup VC(S_2, Q)$.

Assuming $\models VCG(\{P\}\text{if } b \text{ then } S_1 \text{ else } S_2\{Q|T\})$.

- Since $P \wedge \mathcal{B}[\![b]\!] \to wp_1$, $t_1 = t_1$, and $VC(S_1, Q)$,

$$\models VCG(\{P\}S_1\{Q|t_1\})$$

- Since $P \wedge \neg\mathcal{B}[\![b]\!] \to wp_2$, $t_2 = t_2$, and $VC(S_2, Q)$,

$$\models VCG(\{P\}S_2\{Q|t_2\})$$

From our Induction Hypothesis, we have $\vdash \{P \wedge \mathcal{B}[\![b]\!]\}S_1\{Q|t_1\}$, and $\vdash \{P \wedge \neg\mathcal{B}[\![b]\!]\}S_2\{Q|t_2\}$.

By the *if rule*, we get

$$\{P\}\text{if } b \text{ then } S_1 \text{ else } S_2\{Q|t_1 + \mathcal{TB}[\![b]\!]\}$$

Since $T = t_1 + \mathcal{TB}[\![b]\!]$, by the *weak rule*

$$\vdash \{P\}\text{if } b \text{ then } S_1 \text{ else } S_2\{Q|T\}$$

Case for:

Induction Hypothesis:

$$\models VCG(\{P\}S\{Q|T\} \to \vdash \{P\}S\{Q|T\})$$

Let us consider

$$wpc(\text{for } i = a \text{ to } b \text{ do } S, Q) = (I[a/i], (b-a) \times (\mathcal{TA}[\![a]\!] + C_{ASSIGN\_V} + t) + (b-a+1) \times \mathcal{TB}[\![a \leq b]\!])$$

where $wp_S, t_S = wpc(S, I[i+1/i])$

$$\begin{aligned}
VC(\text{for } i = a \text{ to } b \text{ do } S, Q) =& \{I[b/i] \to Q\} \cup && (6.1)\\
& \{(I \wedge \neg(a < b)) \to Q\} \cup && (6.2)\\
& \{(I \wedge a \leq i < b) \to wp\} \cup && (6.3)\\
& VC(S, I[i+1/i]) && (6.4)
\end{aligned}$$

$$\begin{aligned}
VCG(\{P\}\text{for } i = a \text{ to } b \text{ do } S\{Q|T\}) =& \{P \to I[a/i]\} \cup && (6.5)\\
& \{T = (b-a) \times t + (b-a+1) \times \mathcal{TB}[\![a \leq b]\!]\} \cup && (6.6)\\
& VC(\text{for } i = a \text{ to } b \text{ do } S, Q) && (6.7)
\end{aligned}$$

Assuming $\models VCG(\{P\}\text{for } i = a \text{ to } b \text{ do } S\{Q|T\})$.

Given 6.3, , $t_S = t_S$, and VC(S,I[i+1/i]) then $\models VCG[\{I \wedge a \leq i < b\}S\{I[i+1/i]|t\}]$

From our Induction Hypothesis

$$\vdash \{I \wedge a \leq i < b\}S\{I[i+1/i]|t\}$$

By the *for* rule

$$\vdash \{I[a/i]\}\text{for } i = a \text{ to } b \text{ do } S\{I[b/i]|(b-a)\times(\mathcal{TA}[\![a]\!]+C_{ASSIGN\_V}+t)+(b-a+1)\times\mathcal{TB}[\![a \leq b]\!]\}$$

Given 6.5, 6.1 and 6.6 we get

$$\vdash \{P\}\text{for } i = a \text{ to } b \text{ do } S\{Q|T\}$$

□

Example: Range Filter

We now apply the VCG algorithm to the range filter example 6.4. We will use the same notation as in section 6.2 and refer to our precondition as P, our program as S, our postcondition as Q, and our tight cost as T. We will also refer to the *for* loop body statement as S_{if}. The algorithm starts with a call to the VCG function.

$$\begin{aligned} VCG(\{P\}S\{Q|T\}) = &\{P \rightarrow wp) \cup && (6.8)\\ &\{T = t\} \cup && (6.9)\\ &VC(S,Q) \end{aligned}$$

Where

$$wp, t = wpc(S,Q) = (I[0/i][0/j], n \times t_{if} + (n+1) \times \mathcal{TB}[\![0 < n]\!] + 2)$$

$$wp_{if}, t_{if} = wpc(S_{if}, I[i+1/i])$$

Then the VCG function calls the VC function for S. Since only for loops generate extra VCs, we will omit other calls to the VC function for simplicity.

$$\begin{aligned} VC(S,Q) = &\{(I \wedge 0 \leq i < n) \rightarrow wp_{if}\} \cup && (6.10)\\ &\{(I[n+1/i]) \rightarrow Q\} \cup && (6.11)\\ &\{(I[0/i] \wedge n < 0) \rightarrow Q\} \cup && (6.12)\\ &VC(S_f, I[i+1/i]) \end{aligned}$$

To prove our triple, we now simply need to prove all of the VCs generated by the algorithm (6.8 to 6.12), this can easily be done for all the conditions manually, or with the assistance of a theorem prover.

As would be expected, proving a Hoare triple by applying the VCG algorithm is simpler and more mechanic than proving it directly by applying our rules and deriving the inference tree.

Chapter 7

Implementation and Experimental Results

This chapter describes how we implemented our verification tool for all three versions of our logic. We will also present implementations of classic algorithms and how we prove their correctness and cost using our tool. We have implemented our verification system prototype in OCaml, and all the code and examples are in the GitHub repository https://github.com/carolinafsilva/time-verification.

7.1 Tool Architecture

Our goal is to write programs with annotation of correctness and time bounds and be able to prove these conditions. In figure 7.1 we show the architecture of our tool, with each of the steps that will allow us to meet our goal.

Program verification is conducted in three stages:

1. Annotation of the program by the programmer, who specifies the correctness conditions that must be met, as well as the cost upper bound.
2. Implementation of the Verification Condition Generator (VCG) which, given an annotated program generates a set of goals that need to be proved.
3. The proof stage: proof goals are passed to a theorem prover which attempts to prove them automatically. If it fails, some interaction with the user is needed to guide the proof.

Achieving the first stage involves defining the language Abstract Syntax Tree (AST), parsing the program and annotations, and implementing the operational semantics interpreter.

The second stage includes the implementation of our VCG algorithm and interaction with the oracle, to provide extra information about program loops. At the end of stage two, our tool

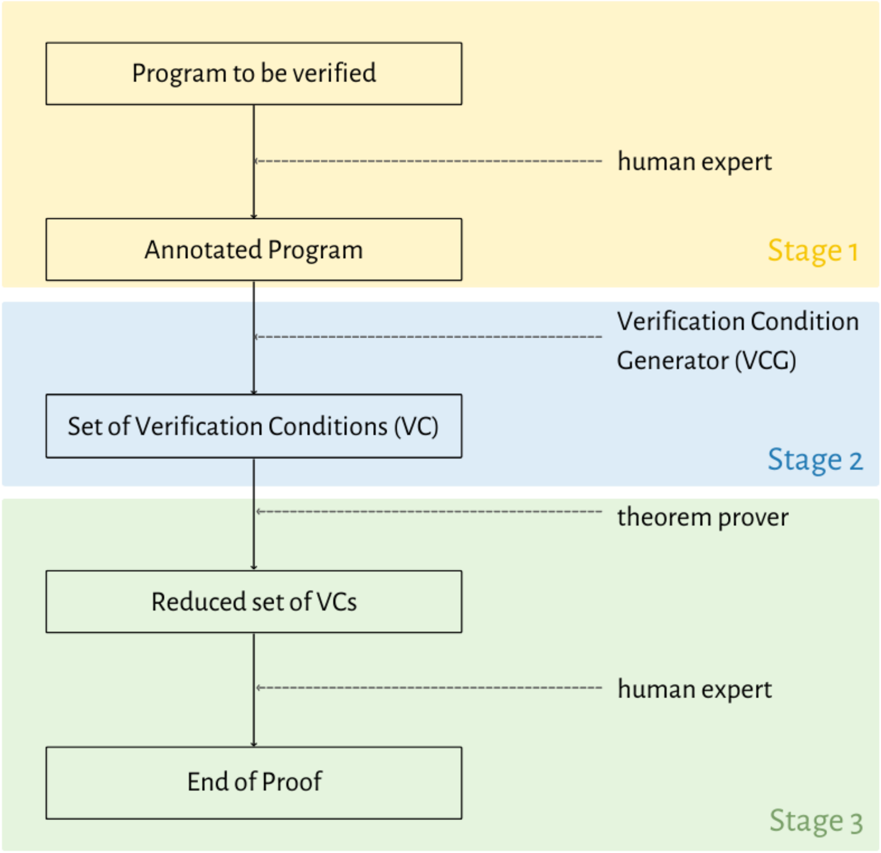

Figure 7.1: Architecture of our tool.

has generated a set of proof goals needed to ensure correction, termination, and resource usage of our input program.

Finally, in stage three, we discard these proof goals by sending them to an automatic prover (e.g., Easycrypt, why3) for validation. This step might need some assistance from the user since the generated VCs might be too complex to be automatically proved. If all our VCs are validated, we know our program is correct, and we have learned some restrictions on its execution time.

Verification is semi-automatic in the sense that in certain situations, the user has to give extra information to the program, either in the form of an oracle that defines some needed parameters or in the proof stage in situations where the proof is interactive.

7.2 Implementation Details

7.2.1 Cost Model

One essential part of our system is our cost model. We need to define a way to evaluate the cost of a program. We start by defining a map to store the cost of atomic operations. For instance, the cost of a sum (C_+) might be defined as 1, and the cost of multiplication (C_*) as 3. Besides our dictionary, we implement a semantic to define the cost of evaluating arithmetic expressions, boolean expressions, and statements. Our operational semantics uses the cost model to compute the real execution cost.

7.2.2 Oracle

Programs with loops require additional information to prove correctness, termination, and cost bounds. This information is provided through an oracle. This oracle will request user input when necessary to complete the VCG algorithm. Since information such as invariants is needed multiple times throughout the algorithm, we need to store this information to be easily accessible. To achieve this, we assign a unique identifier to each *while-loop* and create an oracle hashtable to store the oracle information for each loop. The code for our oracle can be seen in listing 7.1.

```
let oracle_hashtbl = Hashtbl.create 43

let parse_info name parser_function =
  Printf.printf name ;
  let input = read_line () in
  Lexing.from_string input |> parser_function Lexer.token

let oracle () =
  try
    let inv = parse_info "Invariant: " Parser.annot_start in
    let f = parse_info "Variant: " Parser.aexp_start in
    let n = parse_info "Number of iterations: " Parser.aexp_start in
    let t = parse_info "Cost Function of While: " Parser.lambda_start in
    (inv, f, n, t)
  with _ -> failwith "Oracle Error\n"

let get_oracle id =
  if Hashtbl.mem oracle_hashtbl id then Hashtbl.find oracle_hashtbl id
  else
    let inv, f, n, t = oracle () in
    Hashtbl.add oracle_hashtbl id (inv, f, n, t) ;
    (inv, f, n, t)
```

Listing 7.1: Oracle Implementation.

7.2.3 VCG

Let us look at our VCG implementation. We implemented this algorithm as similar to the theoretical definitions as possible to guarantee soundness, as per our proofs. We presented three theoretical definitions of our logic, one classic for upper bounds, one which uses amortized analysis to further refine our upper-bound estimation, and one that proves the exact costs of a restricted version of our language.

Consider the implementation of the Weakest Precondition (wp) function in listing 7.2. Note how in the while case we start by calling the *get_oracle* function to get our loop information.

```
let rec wpc s phi =
  match s with
  | Skip ->
      (phi, Var "Skip")
  | Assign (x, a) ->
      (subst phi x a, Sum (Var "Assign", time_aexp a))
  | ArrAssign (x, a1, a2) ->
      let t’ = Sum (time_aexp a1, time_aexp a2) in
      (subst_arr phi x a1 a2, Sum (Var "Assign", t’))
  | Seq (s1, s2) ->
      let phi’, t2 = wpc s2 phi in
      let phi, t1 = wpc s1 phi’ in
      (phi, Sum (t1, t2))
  | If (b, s1, s2) ->
      let wp1, t1 = wpc s1 phi in
      let wp2, t2 = wpc s2 phi in
      let v_b = annot_of_bexp b in
      let tb = time_bexp b in
      (AAnd (AImpl (v_b, wp1), AImpl (ANeg v_b, wp2)), Sum (Sum (t1, t2), tb))
  | While (id, b, _) ->
      let inv, f, n, t = get_oracle id in
      let time =
        Sum
          ( Mul (Sum (n, Cons 1), time_bexp b)
          , Sigma ("k", 0, Sub (n, Cons 1), lambda_app t (Var "k")) )
      in
      (AAnd (inv, AGe (f, Cons 0)), time)
```

Listing 7.2: Weakest Precondition Implementation.

Similarly, we can see the VC function implementation in listing 7.3. The *while* case calls the *get_oracle* function again to retrieve the information about loop invariant, variant, and cost.

Finally, we show the entry point function VCG in listing 7.4. This function calls both *wpc* and VC and combines all the VCs together.

The VCG implementation for amortized costs differs from the previous one, only for the *while* case. The oracle will also request new information in this version, an amortized cost and a

```
let rec vc s phi: annot list =
  match s with
  | Skip | Assign (_, _) | ArrDef (_, _) | ArrAssign (_, _, _) ->
      []
  | Seq (s1, s2) ->
      vc s1 (wp s2 phi) @ vc s2 phi
  | If (_, s1, s2) ->
      vc s1 phi @ vc s2 phi
  | While (id, b, s’) ->
      let inv, f, n, t = get_oracle id in
      let b = annot_of_bexp b in
      let wp, t’ = wpc s’ (AAnd (inv, AGt (f, Var "k"))) false in
      AForall ("k", AImpl (AAnd (inv, AAnd (b, AEq (f, Var "k"))), wp))
      :: AImpl (AAnd (inv, AAnd (b, AEq (f, Var "k"))), AGe (lambda_app t (Var
          "k"), t’))
      :: AImpl (AAnd (inv, ANeg b), phi)
      :: AImpl (AAnd (inv, b), ALe (f, n))
      :: vc s’ (AAnd (inv, ALe (f, Var "k")))
```

Listing 7.3: VC Implementation.

```
let vcg pre s t pos =
  let wp, ts = wpc s pos true in
  AImpl (pre, ALe (ts, t)) :: AImpl (pre, wp) :: vc s pos
```

Listing 7.4: VCG Implementation.

potential function instead of a function of cost.

The exact cost version of our logic requires additional changes. We start by extending our language with for-loops and implementing the required adaptations to our interpreter. A restriction will be added to *if* statements to ensure equal run time for both branches. Our VCG algorithm will now be extended to deal with *for*-loops and the inequality operator in the cost assertion will now be replaced with an equality operator to prove the exact-time bound.

7.3 Examples

Let us now analyze some working examples implemented in this language and the conditions generated by our VCG algorithm. Particularly we will be able to look at the implementation and results we have already analyzed in previous chapters. For simplicity, we defined the cost of all atomic operations as 1 in our cost dictionary. In table 7.1 we show how many VCs each example generated and which of the three versions of our logic was used.

7.3.1 Insertion Sort

Our first example is of a classic sorting algorithm, insertion sort. The implementation is presented in listing 7.5.

```
{ n > 0 }
i = 1;
while i < n do
  key = x[i];
  j = i - 1;
  while j >= 0 and x[j] > key do
    x[j + 1] = x[j];
    j = j - 1
  end;
  x[j + 1] = key;
  i = i + 1
end
{ forall k. (0<=k and k<n) => x[k] >= x[k-1] | 9*n*n + 27*n + 13 }
```

Listing 7.5: Insertion Sort Implementation with Annotations.

The precondition simply states that n is a positive number. The postcondition says that our final array is in ascending order. Since the implementation has two *while* loops, we will have two calls to the oracle.

For the external loop, we define the maximum number of iterations as n. The variant is the i variable since it always increases until it reaches the value of n. The invariant states that the array is always ordered from the first position until the $(i-1)$-th position: $\forall k.(0 < k \wedge k < i) \rightarrow (x[k-1] \leq x[k])$. The cost of the body of the external *while* is not the same for all iterations, since we have a nested while. We define this cost with the function: $t(i) = 9 \times i + 15$.

For the internal loop, it will iterate i times. The variant is the increasing expression $i - j$. The invariant is that all elements between positions i and j are greater than the key and that from the first position until $i-1$ the array is sorted, excluding the element on position j: $(\forall k.(j < k \wedge k < i) \rightarrow x[k] > key) \rightarrow \forall k1, k2.0 \leq k1 \wedge k1 \leq k2 \wedge k2 < i \wedge \neg(k1 = j) \wedge \neg(k2 = j) \rightarrow x[k1] \leq x[k2]$. Note that one can define a cost function $t(k)$ that would allow us to derive an exact cost for the *while* rule, however, our logic does not allow proving that this bound is tight.

Given this information, our VCG generates the conditions needed to prove the termination, correctness, and cost bound of our program.

7.3.2 Binary Search

Our next example is another classic algorithm, Binary Search. Here we want to prove that, not only our implementation is correct and terminates, but also that the algorithm runs in logarithmic time in the size of the array. The specification can be seen in listing 7.6

```
{(forall i. (0 <= i and i < n) => a[i] < a[i+1]) and (exists j. a[j] = v)}
l = 0;
u = n - 1;
while l <= u do
  m = l + ((u - l) / 2);
  if a[m] < v then
    l = m + 1
  else
    if a[m] > v then
      u = m - 1
    else
     result = m;
     l = u + 1
    end
  end
end
{0 <= result and result < n and a[result] = v | 43 * log(n) + 10}
```

Listing 7.6: Binary Search Implementation with Annotations.

The precondition states that the array a is sorted, and that value v is in the array. The postcondition says that $result$ is a valid position in a and it corresponds to the position of v in a, $a[result] = v$.

To prove the execution time bound, we provide to the oracle the maximum number of iterations as being $log(n)$ and a constant value as the cost of each loop body iteration. To prove termination we must also provide $n - u + l$ as a variant. Our invariant says that the position we are looking for is between l and u, $0 \leq l \land u < n \land (\forall i.(0 \leq i \land i < n \land a[i] = v) \rightarrow l \leq i \land i \leq u)$ as invariant.

7.3.3 Binary Counter

In the binary counter algorithm, we represent a binary number as an array of zeros and ones. We start with an array with every value at zero, and with each iteration, we increase the number by one until we reach the desired value. Our implementation can be seen in listing 7.7.

Unlike in previous examples, we applied our amortized logic to prove the bound of the binary counter algorithm. Our precondition says that n is a positive value, size is $log(n)$ and that all elements in array B from 0 to $size$ start at zero. Our postcondition says that at the end of the program, array B is a binary representation of decimal number n. In order to prove this assertion, we must provide the oracle with the amortized cost $(2c)$ and a potential function denoting the number of ones in the array at each iteration. We must also specify the invariant $i = sum(k, 0, size, B[k] * 2^k)$, the variant i, and the maximum number of iterations $size$, to prove correctness and termination respectively.

```
{n >= 0 and size = log(n)}
i = 0;
while i < n do
  j = 0;
  while B[j] = 1 do
    B[j] = 0;
    j = j + 1
  end;
  B[j] = 1;
  i = i + 1
end
{n = sum(i,0,log(n) - 1, B[i]*2^i) | 20*c*n + 3*n + 30}
```

Listing 7.7: Binary Counter Implementation with Annotations.

If we were to use a worst-case analysis on this implementation, we would get that this algorithm is $\mathcal{O}(n\ logn)$, meaning we would flip every bit $(logn)$ a total of n times. However, this is not the case. While the first bit $(B[0])$ does flip every iteration, the second bit$(B[1])$ flips every other iteration, the third $(B[2])$ every 4th iteration, and so on. We can see a pattern where each bit $B[i]$ flips every 2^ith iteration. This will mean that, at most, we have $2n$ bit flips, meaning our algorithm is actually $\mathcal{O}(n)$, as we successfully proved with our algorithm. we can define a potential function as:

7.3.4 Range Filter

In our last example, we implement a simple filter where, given an array (a) and a range [l..u], we use an auxiliary array (b) to filter if the elements in a are within the range, listing 7.8.

```
{ 0 <= l and l < u and n >= 0}
j = 0;
for i=0 to n do
  if (l <= a[i] and a[i] <= u)
  then
    b[j] = a[i];
    j = j + 1
  else
    b[j] = b[j];
    j = j + 0
  end
end
{ forall i. (0<=i and i<n) => (l <= a[i] and a[i] <= u => exists j. B[j] = a[i]
    )
and (l > a[i] and a[i] > u => not (exists j. b[j] = a[i]) ) | 13*n + 10 }
```

Listing 7.8: Range Filter Implementation with Annotations.

Our pre-condition states that l and n are positive values, and u is greater than l. Our postcondition says that for every i element in a in the range [l..u], i will also be in B. And for every i element in a not in the range [l..u], i will not be in B.

We provide the invariant $\forall k.(0 \leq k \wedge k < i) \rightarrow (l \leq a[k] \wedge a[k] \leq u \rightarrow \exists k.b[j] = a[i]) \wedge (l > a[k] \wedge a[k] > u \rightarrow \neg(\exists j.b[k] = a[i]))$ in order to prove correctness. Using our VC generator and EasyCrypt we prove that not only is this algorithm correct, the cost we provide of $13n + 10$ is the exact cost of this program. This result allows us to conclude the time it takes to run depends only on the size of the array, and not on its values.

Algorithm	Logic	Number of VCs Generated
Insertion Sort	Classic	10
Binary Search	Classic	6
Binary Counter	Amortized	17
Range Filter	Exact	5

Table 7.1: Logic Used and Number of VCs generated by each example.

Chapter 8

Conclusion

The topic of static cost analysis is not new. There is a lot of previous research on how to get reasonable estimations of cost or worst-case scenario costs, either by using type systems or using an axiomatic semantics. Our work continues on this but aims to produce tighter bounds than what we found so far in the literature.

We first extended the traditional logic of worst-case cost to use amortized analysis, giving better results for programs that fit the amortized analysis scenario.

Then, we further extended our logic to a restricted version of our language, where one can prove the exact cost of execution. As far as we know, this is a novel logic. This result is rather significant if we consider the application to critical systems where the worst-case cost is not enough to guarantee all safety goals. It is also relevant if applied to cryptographic implementations, where timing leakage might be a security concern.

8.1 Future Work

One of the first improvements we are aiming towards is to develop a single system capturing all of the cost logics together, creating a more cohesive, powerful tool.

We primarily focused on theoretical definitions and guaranteeing a sound logic that produced reasonable bounds. Our implementation is a simple prototype that serves as proof of concept of these definitions. Therefore many improvements can be made to our tool concerning efficiency, transforming it from a conceptual tool to a practical one.

We also defined our logic using a simple language, which allowed us to focus on the cost estimation aspect without having to worry so much about language details. In the future, we want to extend our language with more features, such as functions, to improve the expressiveness of programs. We would also like to expand the application of our logic to more extensive and complex case studies, namely cryptographic implementations.

Our work started as an adaptation of the EasyCrypt cost logic developed in [5]. In the future, we would like to propose an extension to the EasyCrypt tool with our logic.

Bibliography

[1] E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, G. Puebla, D. Ramírez, G. Román, and D. Zanardini. Termination and cost analysis with costa and its user interfaces. *Electronic Notes in Theoretical Computer Science*, 258(1):109–121, 2009. ISSN: 1571-0661. Proceedings of the Ninth Spanish Conference on Programming and Languages (PROLE 2009). doi:https://doi.org/10.1016/j.entcs.2009.12.008.

[2] José Almeida, Maria Frade, Jorge Pinto, and Simão Sousa. *Rigorous Software Development. An Introduction to Program Verification*. Springer Verlag, 01 2011. doi:10.1007/978-0-85729-018-2.

[3] Robert Atkey. Amortised resource analysis with separation logic. *Logical Methods in Computer Science*, 7(2), jun 2011. doi:10.2168/lmcs-7(2:17)2011.

[4] Martin Avanzini and Ugo Dal Lago. Automating sized-type inference for complexity analysis. *Proc. ACM Program. Lang.*, 1(ICFP), aug 2017. doi:10.1145/3110287.

[5] Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, and Pierre-Yves Strub. Mechanized proofs of adversarial complexity and application to universal composability. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 2541–2563, New York, NY, USA, 2021. Association for Computing Machinery. ISBN: 9781450384544. doi:10.1145/3460120.3484548.

[6] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. Alternating runtime and size complexity analysis of integer programs. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 140–155, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN: 978-3-642-54862-8.

[7] Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. End-to-end verification of stack-space bounds for c programs. *SIGPLAN Not.*, 49(6):270–281, jun 2014. ISSN: 0362-1340. doi:10.1145/2666356.2594301.

[8] Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. Compositional certified resource bounds. *SIGPLAN Not.*, 50(6):467–478, jun 2015. ISSN: 0362-1340. doi:10.1145/2813885.2737955.

[9] Stephen A Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing*, 7(1):70–90, 1978.

[10] Edsger W Dijkstra. *Usability engineering.* Englewood Cliffs, N.J. : Prentice-Hall, 1976. ISBN: 013215871X 9780132158718.

[11] Robert W. Floyd. Assigning meanings to programs. In *Proceedings of Symposia in Applied Mathematics Vol. 19*, pages 19–32. Amer. Math. Soc., 1967.

[12] Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. Speed: Precise and efficient static estimation of program computational complexity. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, page 127–139, New York, NY, USA, 2009. Association for Computing Machinery. ISBN: 9781605583792. doi:10.1145/1480881.1480898.

[13] Maximilian P. L. Haslbeck and Tobias Nipkow. Hoare logics for time bounds. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 155–171, Cham, 2018. Springer International Publishing. ISBN: 978-3-319-89960-2.

[14] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10): 576–580, oct 1969. ISSN: 0001-0782. doi:10.1145/363235.363259.

[15] Jan Hoffmann and Martin Hofmann. Amortized resource analysis with polynomial potential. In Andrew D. Gordon, editor, *Programming Languages and Systems*, pages 287–306, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN: 978-3-642-11957-6.

[16] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. Towards automatic resource bound analysis for ocaml. *SIGPLAN Not.*, 52(1):359–373, jan 2017. ISSN: 0362-1340. doi:10.1145/3093333.3009842.

[17] Martin Hoffmann and Steffen Jost. Type-based amortised heap-space analysis. In Peter Sestoft, editor, *Programming Languages and Systems*, pages 22–37, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN: 978-3-540-33096-7.

[18] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. Weakest precondition reasoning for expected runtimes of randomized algorithms. *J. ACM*, 65(5), aug 2018. ISSN: 0004-5411. doi:10.1145/3208102.

[19] James Cornelius King. *A program verifier.* Carnegie Mellon University, 1970.

[20] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, jul 2009. ISSN: 0001-0782. doi:10.1145/1538788.1538814.

[21] Zohar Manna. The correctness of programs. *Journal of Computer and System Sciences*, 3 (2):119–127, 1969.

[22] Hanne Riis Nielson. A hoare-like proof system for analysing the computation time of programs. *Science of Computer Programming*, 9(2):107–136, 1987. ISSN: 0167-6423. doi:https://doi.org/10.1016/0167-6423(87)90029-3.

[23] H.R. Nielson and F. Nielson. *Semantics with Applications: An Appetizer*. Undergraduate Topics in Computer Science. Springer London, 2007. ISBN: 9781846286919.

[24] Ivan Radiček, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Florian Zuleger. Monadic refinements for relational cost analysis. *Proc. ACM Program. Lang.*, 2(POPL), dec 2017. doi:10.1145/3158124.

[25] Alejandro Serrano, Pedro López-García, and Manuel V. Hermenegildo. Resource usage analysis of logic programs via abstract interpretation using sized types. *Theory and Practice of Logic Programming*, 14, 05 2014. doi:10.1017/S147106841400057X.

[26] Hugo Simões, Pedro Vasconcelos, Mário Florido, Steffen Jost, and Kevin Hammond. Automatic amortised analysis of dynamic memory allocation for lazy functional programs. *SIGPLAN Not.*, 47(9):165–176, sep 2012. ISSN: 0362-1340. doi:10.1145/2398856.2364575.

[27] Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985.

[28] Pedro B. Vasconcelos, Steffen Jost, Mário Florido, and Kevin Hammond. Type-based allocation analysis for co-recursion in lazy functional languages. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Proceedings*, volume 9032 of *LNCS*, pages 787–811. Springer, 2015.