Abstract

Modular and Certified Resource-Bound Analyses

Quentin Carbonneaux

2017

Today, taming the resource consumption of programs is an important problem. Excessive resource usage *costs money* by requiring ever larger computing infrastructures. Imprecise resource usage *compromises the safety* of hard real-time systems and puts human lives at risk. Input-dependent resource usage *compromises the security* of communication protocols by allowing side-channel attacks.

In this work, we present *modular* and *certified* solutions to the resource-bound problem; this problem consists in finding bounds on the resource requirements of a program. To prove the practicality of our approach, we implement multiple systems with automation support and evaluate them on examples and benchmarks against state-of-the-art tools.

The modularity of our approach is provided at three different levels. First, our systems are compositional, allowing to split the resource analysis of a program in smaller units. Second, the design of the analysis enables modular implementations split in orthogonal components, making the systems more easily adaptable to new languages and new automation techniques. Third, our systems allow modular use of automation by providing a unified setting in which automatically-derived proofs can be combined with manual reasoning.

To provide the highest confidence level for the inferences made in our resource-analysis systems, we use machine-checkable certificates. The certificates produced contain a theorem asserting the validity of the bounds inferred with respect to resource-aware semantics of the language. They are free from any assumptions and checkable without modifications by the Coq proof assistant.

In addition to meeting the design requirements above, our systems provide multiple original contributions to the field of resource analysis. We show how to adapt automatic amortized resource analysis to imperative programs with possibly-recursive procedure calls. Before, it was exclusively available for functional languages. We show how resource analysis can be done at the source level while still having formal guarantees on the compiled program. We implement the first automated resource-analysis systems with which a user can soundly interact. We implement the first automated resource-analysis system that generates machine-checkable certificates for the bounds it inferred.

Modular and Certified Resource-Bound Analyses

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Quentin Carbonneaux

Dissertation Director: Zhong Shao

December 2017
Draft of July 13, 2017

Contents

Chapter 1

Introduction

1.1 The Resource-Bound Problem

In this dissertation, I study the *resource-bound problem*; this problem consists in finding bounds on the resource requirements of a program. Examples of resources include time, memory, energy, bytes of data exchanged on a network, or calls to a cloud API.

The resource-bound problem arises naturally when building systems. For example, the resource requirements of a program are often related to the *cost of the equipment* to run it. Thus, companies like Google optimize the efficiency of their software to be able to compute with smaller data centers. Similarly, mass-produced embedded systems use efficient programs that run on the smallest computer possible in order to minimize production costs.

With hard real-time systems, even though cost is not a primary concern, the resource behavior of programs is of paramount importance for *safety.* For example, it is critical that a fly-by-wire system answers to external input from the airplane's stick in a timely manner. Similarly, an anti-lock breaking system needs to act as soon as the wheels of the car are blocked.

Side-channel attacks, by observing the resource-usage patterns of programs, can infer parts of their private state, comprimising their security. Resource bounds can help develop so-called constant-time implementations that are robust to such attacks and hence, help create more *secure software.*

In general, quantitative information about the resource behavior of programs is a *useful feedback* to the developer, and more and more companies include diagnostic tools for the resource behavior of programs as part of their continuous integration process.

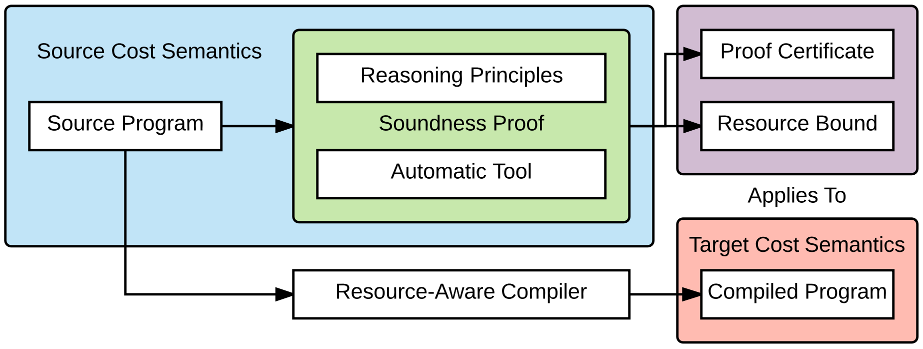

Figure 1.1: Overview of a complete solution to the resource-bound problem.

1.2 Key Components for Resource-Bound Analysis

A complete solution to the resource bound program requires multiple interacting components. In this section, I list them and describe what is their role in the overall solution. Each of the components appears in the overview chart in Figure 1.1.

Cost semantics. To talk about the resource behavior of a program, it is necessary to know its execution environment. A description of this execution environment suitable for the resource-bound problem should specify how each action of the program is executed, and what is its cost. To this end, we use *cost semantics*; cost semantics give a mathematical definition of what it means to run a program and how it affects the resources available. It is important to note that cost semantics are specific to the language programs are written in, thus, the cost semantics of an assembly program will be different from the one of a C program. In Figure 1.1, two cost semantics appear, one for the program written by the programmer (Source Cost Semantics), and one for the program that runs on the machine (Target Cost Semantics). The link between these two semantics is done by the resource-aware compiler.

Resource-aware compilers. Programs are usually written in a high-level source language and then compiled to the low-level target language understood by the computer. High-level languages are more suitable for reasoning, but it is low-level languages that are effectively executed by the computer. Ideally, one would want to reason at the source level and obtain guarantees on the compiled program. This is precisely what formally verified compilers [KN06, Chl07] such as CompCert [Ler09] enable. However, important quantitative properties such as memory and time consumption are not modeled nor preserved by those formally verified compilers. This has led researchers to believe that it is necessary to reason about these properties at the assembly level "since only at this level is all necessary information available" [WEE$^+$08]. But we claim otherwise: to enable reasoning about the resource consumption of programs at the source level, we need *resource-aware compilers* that preserve the resource behavior of the programs compiled. In Figure 1.1 (p. 9), the resource-aware compiler bridges the gap between the source cost semantics and the target cost semantics. Additionally, it allows the resource bound inferred from the source program to apply to the compiled program.

Reasoning principles. Similarly to how Floyd [Flo67] and Hoare [Hoa69] proposed to formally assign meaning to programs, it is necessary to provide a formal setting in which proofs about the resource consumption of programs can be be established. This setting is what I call *reasoning principles*. In the case of the resource-bound problem, reasoning principles find two uses. First, they let the programmer formally derive correct resource bounds for his program manually. Second, they are used to justify the soundness of automatic tools. These two roles are displayed in Figure 1.1 (p. 9). In the figure, the reasoning principles are displayed over the "Source Cost Semantics" box to emphasize that they have to be designed consistent with those semantics.

Automatic tools. The analysis of the resource consumption of programs can be tedious, time comsuming, and complex, even for experts. To alleviate this burden, *automatic tools* are desirable. In the general case, complete automation of the resource-bound problem is impossible—or, more formally, undecidable. Fortunately, in practice, this limitation does not prevent the implementation of useful tools that work for many programs. In Figure 1.1 (p. 9), the automated tool helps derive a resource bound from the input source program. When the automation cannot fully handle the source program, the user of the system can use the reasoning principles manually to fill the gaps. A soundness proof links the automated tool to the reasoning principles, ensuring that the bounds generated are correct.

Proof certificates. Automatic resource-analysis tools have complex implementations and rely on subtle invariants for their correctness. To provide the highest level of confidence to its user, a resource-bound system can generate *proof certificates*. Proof certificates are logical objects that justify the validity of one resource bound. They can be validated by an external trustworthy checker, and hence remove the automatic tool from the trusted computing base.

1.3 Some Challenges for Resource-Bound Analysis

The development of a solution for the resource-bound problem involves multiple challenges.

First, resource analysis must be compositional. In programming languages, compositionality is the idea that bigger programs are composed by aggregating smaller sub-programs. For example, multiple statements can be sequenced in a block and blocks can be abstracted as procedures. A compositional resource-bound analysis must embrace this concept by splitting the reasoning done for one program into multiple smaller pieces for each component of the program.

This design goal is however hard to attain in practice. First, many of the techniques to obtain resource bounds are based on some form of *ranking function* or *counter instrumentation* that is linked to a local analysis. As a result, loop bounds are expressed in terms of the values of variables just before the loop. This makes it hard to give a resource bound on a sequence of loops and function calls in terms of the input parameters of a function. Second, while all popular imperative programming languages provide a function or procedure abstraction, many techniques are not able to abstract resource behavior; instead, they have to inline the procedure body to perform their analysis.

Second, resource analysis must be modular. Ideally, one would like a resource analysis system as described in Section 1.2, where each of the components can be changed freely. In particular, it is desirable to keep the techniques on which the automatic tools are based orthogonal with the more general reasoning principles of the programming language. With such a separation, one can adapt the automatic tool to a new language more simply than if the automation technique is tangled with the language it applies to. Moreover, this separation is also a way to provide user interaction in case the automatic analysis fails, offering one solution to the challenge that I describe next.

Beyond reasoning principles, there does not exist a framework in which automatic resource-bound tools can be implemented and extended. The potential method, on which multiple automatic systems [HDW17, HJ03, HJ06, CHS15] are based, seems to be a good candidate for this framework. However, none of the previous work gives a roadmap on how extensions should be implemented. Additionally, even though these work claim to be all based on the same potential method, this method remains a *design pattern* and never graduated to a proper *formal setting*.

Third, user interaction must be possible. Tarjan has shown that even a simple algorithm can have a very complicated running time [Tar75]. When a complex reasoning is necessary to perform the resource analysis of a program, automated tools must be complemented or replaced with manual proofs. However, the vast majority of automated tools does not provide any mean for interaction. Some tools based on a type system come close the goal [HDW17, HJ03, JLH$^+$09, HJ06, JHLH10]; however, the truncated parts of the typing derivation must be provided as axioms by the user and remain unverified. Finally, no state-of-the-art automatic resource-bound system offers a mechanism for the user to provide hints when the tool fails to find a bound. This is in contrast to state-of-the-art program verification tools like Dafny [Lei10] where the user can provide invariants and helper lemmas.

Finally, bounds should come with verifiable certificates. As argued earlier, imprecise resource bounds can have consequences on both safety and security of computer-controlled systems. Thus, in the same way that certified software [Sha10] comes with a proof that it respects its specification, resource bounds should come with a certificate justifying their correctness. However, no state-of-the-art resource-bound tool is able to produce certificates. Most tools that work on imperative programs [BEF$^+$16, AAG$^+$12, ADFG10, FMH14, SZV15] first abstract the program source as some kind of transition system; this transformation, similar to a compilation pass, is never specified and makes it hard to relate the bounds computed to the original program. Many papers about tools based on the potential method and linear programming argue that it is possible to generate checkable certificates [HDW17, HJ03, JLH$^+$09, HJ06, JHLH10, CHS15]; however none of the implemented tools supports the generation of certificates.

1.4 Contributions

In this thesis, *I argue that it is possible to design modular systems that infer certified resource bounds.* By modular, I mean that such systems

— are compositional, and hence enable *modular reasoning*;

— have automation procedures separated from and built upon formal reasoning principles, and hence enable *modular implementations*;

— offer user-interaction with automation procedures, and hence enable *modular use of automation.*

To prove this point, I design three formal resource-analysis systems in a disciplined incremental bottom-up way that have the potential method at their core. The complete systems are then evaluated on real code and benchmarks against competing systems. As parts of these three systems, multiple features appear for the first time.

— We implement the first resource-aware compiler for stack usage and successfully use it to prove bounds on compiled programs.

— I implement the first automatic tool that interfaces in a well-defined way with manual proofs.

— I implement the first automatic resource-bound tool that generates proof certificates.

Quantitative logics. My first contribution is the keystone enabling modular inference of worst-case resource bounds for structured programs: *quantitative logics.* Quantitative logics are program logics specifically designed for the resource-bound problem. They find their roots in two distinct works: Hoare-style program logics and the potential method for the analysis of algorithms.

Hoare logic [Hoa69] is an immensely influential work that proposes an axiomatic way to proving properties about computer programs. By decomposing the program into smaller units and assembling back the various pieces of reasoning, the user of this logic can derive facts about programs in a compositional way.

The potential technique was first introduced to evaluate the amortized complexity of data structures [Tar85]. Amortized complexity is the analysis of the average cost over time of multiple operations on a data structure. The potential technique was defined with this goal in mind and permits compositional reasoning on a sequence of operations by offsetting the cost of some computationally expensive operations into many simple ones.

Quantitative logics merge these two works by providing a new axiomatic way to manipulate potential functions. The combination of these two ideas yields many benefits. First, using quantitative logics, the classic Hoare-style reasoning can be used to infer properties about the resource behavior of programs. Second, the power of the potential-based reasoning allows to derive resource bounds on many complex examples that require amortization. Third, the derivation of potential functions is now guided by the axiomatic rules of the logic; this is in contrast to Tarjan's original paper where the construction of potential functions is fully left to the ingenuity of the user.

The quantitative logics are formalized in the Coq proof assistant and proved sound and complete.

Flexible automated systems. My second contribution is to provide a toolkit to build automatic procedures using potential-based reasoning and linear programming. This modular approach to building automated systems is enabled by the clean separation of three orthogonal components.

1. *Potential functions.* To enable automation with linear programming, we restrict the potential functions to be linear combinations of more elementary base functions. However, the choice of what these base functions are is for the system designer to take. In this dissertation, I use both interval of program variables and general monomials of program variables.

2. *Programs and their reasoning principles.* Automated systems can be designed both for structured programs and for unstructured control-flow graphs. Regardless of the representation chosen, the program semantics are interfaced cleanly using a suitable resoning principle. In this dissertation, I use quantitative logics and admissible annotations as two reasoning principles for structured programs and control-flow graphs, respectively.

3. *The type of analysis.* The potential method is suitable to proving both resource bounds and size-change properties. Only a simple additional constraint can change a potential-based size-change system into a resource-bound system. Finally, I show that the concept of framing provides an elegant way to combine the two kinds of system.

One key to this flexibility is the clear separation of the reasoning principles from the specificities related to the automated system. In the soundness proof of the automated systems, each rule is clearly related to its more general version in the reasoning principle. This approach in two stages also makes sound user interaction possible. Indeed, by using general reasoning principles as lingua franca, automated systems and manual inferences can coexist in a coherent setting.

Practical implementations. The third contribution of this dissertation is two automated tools, C4B and Pastis. C4B can infer linear resource bounds on C-like programs. Pastis can automatically infer resource bounds and size change information on programs expressed in LLVM bitcode. Both

Automatic System	Potential Functions	Programs	Analysis
C4B	Interval based	Structured programs	Resource bound
Pastis	Monomials	Interprocedural CFGs	Size change

Table 1.1: Orthogonal components of the automated systems of this dissertation.

systems are implementated following the blueprint detailed in the previous paragraph. Table 1.1 (p. 14) summarizes how each component was picked for both systems.

C4B is compared to four other tools on a benchmark of 33 complex example programs, some are collected from previous work and some are original. C4B is also evaluated on larger open source code from the cBench benchmark suite.

Pastis is compared to four other tools in an extensive experimental evaluation consisting in more than 200,000 lines of C and performs better than three of them. Unlike other tools, Pastis can handle arbitrary (mutually) recursive functions, can be extended to support more complex bounds (e.g., logarithmic) by leveraging the flexibility of the toolkit it is based upon, and allows user interaction in the case where a bound cannot be derived completely automatically.

Proof certificates. In addition to being competitive on state-of-the-art benchmarks, Pastis can generate proof certificates that ensure the validity of the generated bounds. This feature is enabled by the formal reasoning principle used for its soundness proof and by the ability of linear-programming solvers to return satisfying assignments. The proof certificates required only a small implementation effort, do not make any assumptions, and can be checked without any intervention of the user by the Coq proof assistant.

A certified resource-aware compiler. Finally, for the special case of the stack resource, we developed a new resouce-aware compiler in the Coq proof assistant. This compiler ensures that the stack usage of the compiled program is less than the one of the original program. The new resource-aware compiler allows to derive stack-usage bounds on an assembly program while offering the convenience of reasoning at the source level.

The new resource-aware compiler is based on the CompCert compiler [Ler09]. In addition to preserving the stack usage of programs, it addresses the imprecise model of the stack in CompCert. This imprecise model was a long-standing limitation that was part of some unverified pretty-printing code. In the resource-aware compiler, the stack is correctly modeled as a single finite block of memory.

1.5 Dissertation Overview

Contents of the Chapters The dissertation starts with a presentation of invariant logics and their metatheory in Chapter 2. This chapter also contains definitions reused many times after.

Invariant logics appear first because they are reused multiple times in other proofs of soundness and completeness in the rest of the dissertation.

In Chapter 3, I introduce the essential concept of resource safety and show how to use it together with an invariant logic to solve the resource-bound problem for stack usage. Then, I define the first quantitative logic of this dissertation to prove stack-usage bounds and prove it sound and complete. Finally, I desribe how Quantitative CompCert, the new stack-aware compiler, was implemented on top of vanilla CompCert and how we linked its main theorem with the bounds proved using the quantitative logic.

In Chapter 4, I shift the focus on automation. I define a second quantitative logic which is not specific to the stack usage and I prove it to be sound. Then, I present an automated system and prove it sound using the quantitative logic as reasoning principle. This system, implemented in the new tool C4B, tracks the size of intervals of program variables and can derive linear bounds for multiple intricate examples, including ones requiring amortized reasoning.

In Chapter 5, I present a more ambitious generic framework to infer size-change properties automatically on low-level programs. I also give a detailed account of the link between size-change information and the resource-bound problem. Finally, I present Pastis, one practical implementation of this framework, and I evaluate it using an extensive experimental evaluation.

Finally, in Chapter 6, I describe how the certificate-extraction feature of Pastis was implemented.

A note about the presentation. In Section 1.4, I mention that this dissertation contributes a flexible toolkit to build automatic systems based on the potential method. In the dissertation, I made the choice to present it by using two examples. The interval system in Chapter 4 (implemented in C4B), and a more generic framework in Chapter 5 (implemented in Pastis).

I believe this presentation is more natural than a monolithic top-down lecture. It allows me to introduce many ideas gradually and shows the reader a first simple yet complete system before diving into more advanced matter (e.g., polynomial bounds and resource-polymorphic recursion).

Table 1.1 (p. 14) reuses the terminology of Section 1.4 to summarize the characteristics of the two automated systems of this dissertation. Mixing and matching their features differently permits to create up to 6 more automatic systems. The two systems presented are the ones that were published in conference papers.

1.6 Published Work

This dissertation contains work that was published in three different papers.

1. *End-to-end Verification of Stack-space Bounds for C Programs* [CHRS14] with Jan Hoffmann, Tahina Ramananandro, and Zhong Shao; in PLDI '14.

   This work matches the Chapter 3.

2. *Compositional Certified Resource Bounds* [CHS15] with Jan Hoffmann, and Zhong Shao; in PLDI '15.

   This work matches the Chapter 4.

3. *Automated Resource Analysis with Coq Proof Objects* [CHRS17] with Jan Hoffmann, Thomas Reps, and Zhong Shao; in CAV '17.

   This work matches the Chapters 5 and 6.

We submitted implementations together with those three papers and they were all validated by the Artifact Evaluation Committees.

The invariant logics presented in Chapter 2 and their soundness and completeness proofs were never published but appeared in a talk entitled *Invariant logics and their theory* during the Yale PL day in November 2015.

Chapter 2

Invariant Logics

2.1 Safety v.s. Functional Correctness

Safety and functional correctness are two essential properties for computer programs. *Safety* ensures that all the steps taken by a program never lead to a "bad" state. *Functional correctness* on the other hand specifies the behavior of a program by relating initial and final states. These two concepts are connected: It makes little sense to talk about the functional correctness of a program that could go wrong; and reciprocally safety often relies on understanding the functional behavior of code.

Both safety and functional correctness are often reasoned about using program logics. And in both cases, the program logics are syntactically often similar. But surprisingly, the semantic accounts of these logics differ wildly, and results from works of one side seldom account for the other side. Below, I state the biggest differences in these two accounts.

— The soundness of program logics for safety is often proved using a *step-indexed* safety predicate and using a *small-step semantics* to describe the programming language. Tools to prove safety can offer guarantees on *non-terminating* programs. *Completeness results* are often missing from the meta-theory of safety logics.

— The soundness of program logics for functional correctness is typically proved using *big-step semantics* to describe the programming language, and no step-indexing techniques are required. Functional specification tools are not able to *distinguish non-termination from divergence.* *Completeness results* are often proved and can be found in many textbooks.

In this chapter, I propose to reconcile the treatment of these two problems past the syntax of the program logic. I first present the syntax of *invariant logics*. Then, I give a unified semantic treatment of this logic presented as two generic theorems. These two theorems provide a generic way to show *soundness and completeness* for at least the safety and functional correcntess interpretations of the logic.

I believe that the proof techniques are a contribution of this dissertation. They work on small-step semantics but do not require step indexing. The completeness result is the only one I known that works with small-step semantics. All the proofs are formalized in the Coq proof assistant in less than 700 lines; I believe that the size of this formalization demonstrates the simplicity and efficiency of the proof techniques.

2.2 Abstract Language Definition

The small structured imperative language I present in this section is slightly idealized, but I believe that it contains all the necessary features—except for function calls, that are added in Section 2.6—to be scaled to a larger and more realistic first-order imperative programming language without technical difficulties.

2.2.1 Syntax

Commands are terms given by the following syntax:

$$c := \mathsf{skip} \mid b \mid c_1; c_2 \mid c_1 {+} c_2 \mid \mathsf{loop}\ c \mid \mathsf{break}.$$

I use b to denote a base statement abstractly, we will see in the operational semantics that these base statements can be non-deterministic and also stop the execution—for example, they can be guard statements that block the execution under certain conditions. The sequential composition of two commands is traditionally written $c_1; c_2$. The non-deterministic composition of two commands $c_1{+}c_2$ will execute either c_1 or c_2. A command can be repeated an arbitrary number of times by prefixing it with loop in the syntax. Loops are exited using the break statement. Finally, skip is the empty command; it does not have any effect.

Continuations are defined by the following syntax:

$$k := \mathsf{kstop} \mid \mathsf{kseq}\ c\ k \mid \mathsf{kloop}\ c\ k.$$

Continuations represent computations that are left to do. The empty continuation kstop signifies that nothing else is to do. The sequence continuation $\mathsf{kseq}\ c\ k$ is a continuation that will first run the command c, then execute the rest k. Finally, $\mathsf{kloop}\ c\ k$ will repeat the command c until it uses break, then follow with the rest k.

$$\frac{\sigma[\![b]\!]\sigma'}{(\sigma, b, k) \to (\sigma', \mathsf{skip}, k)}\ \text{(S:B}\textsc{ase}\text{)}$$

$$(\sigma, c_1; c_2, k) \to (\sigma, c_1, \mathsf{kseq}\ c_2\ k)\ \text{(S:S}\textsc{eq}\text{1)}$$

$$(\sigma, \mathsf{skip}, \mathsf{kseq}\ c_2\ k) \to (\sigma, c_2, k)\ \text{(S:S}\textsc{eq}\text{2)} \qquad (\sigma, \mathsf{break}, \mathsf{kseq}\ c_2\ k) \to (\sigma, \mathsf{break}, k)\ \text{(S:S}\textsc{eq}\text{3)}$$

$$(\sigma, c_1{+}c_2, k) \to (\sigma, c_1, k)\ \text{(S:A}\textsc{lt}\text{1)} \qquad (\sigma, c_1{+}c_2, k) \to (\sigma, c_2, k)\ \text{(S:A}\textsc{lt}\text{2)}$$

$$(\sigma, \mathsf{loop}\ c, k) \to (\sigma, c, \mathsf{kloop}\ c\ k)\ \text{(S:L}\textsc{oop}\text{1)} \qquad (\sigma, \mathsf{skip}, \mathsf{kloop}\ c\ k) \to (\sigma, \mathsf{loop}\ c, k)\ \text{(S:L}\textsc{oop}\text{2)}$$

$$(\sigma, \mathsf{break}, \mathsf{kloop}\ c\ k) \to (\sigma, \mathsf{skip}, k)\ \text{(S:L}\textsc{oop}\text{3)}$$

Figure 2.1: Continuation-based semantics for the minimal structured imperative language.

2.2.2 Semantics

The complete small-step semantics is given in Figure 2.1. It is defined as a transition rule $(\sigma, c, k) \to (\sigma', c', k')$ on configurations. A configuration is a triple of a memory state $\sigma \in \Sigma$, a command c, and a continuation k. Like base statements, memory states Σ are also left abstract. In the semantics, I assume that there is a semantic function $[\![\cdot]\!]$ that maps base statements to relations on memory states $\mathcal{P}(\Sigma \times \Sigma)$. If two states σ and σ' are related by $[\![b]\!]$, I write $\sigma[\![b]\!]\sigma'$.

The rule S:Base means that executing a base statement b updates the memory state according its semantic $[\![b]\!]$; the command part of the configuration is changed to skip by the rule to signify that the execution of b is complete. The rule S:Seq1 encodes how the sequence command $c_1; c_2$ is executed: We shift the focus over to the first command c_1 and stack c_2 on the continuation to do it later. This "later" is precisely explained using the rule S:Seq2. Indeed, when the execution of c_1 is complete, the command part of the configuration becomes the empty command skip; in that case

S:Seq2 pops the next thing to do from the continuation $\mathsf{kseq}\ c_2\ k$. The rule S:Seq3 together with S:Loop3, encodes that break jumps out of the innermost enclosing loop. The two other loop rules, S:Loop1 and S:Loop2 are similar to the sequence rule aside from the fact that they repeat the execution of the "loop body" stored in the continuation. Finally S:Alt1 and S:Alt2 mean that a non-determinstic composition is executed by running either side, and discarding the other.

The rules are shown in action in the example below. In the example, base statements are variable assignments and memory states are maps from variables to numbers.

$$\begin{aligned}
([x \mapsto 0], x := 41; x := x+1, \mathsf{kstop}) &\to ([x \mapsto 0], x := 41, \mathsf{kseq}\ (x := x+1)\ \mathsf{kstop}) \\
&\to ([x \mapsto 41], \mathsf{skip}, \mathsf{kseq}\ (x := x+1)\ \mathsf{kstop}) \\
&\to ([x \mapsto 41], x := x+1, \mathsf{kstop}) \\
&\to ([x \mapsto 42], \mathsf{skip}, \mathsf{kstop})
\end{aligned}$$

In the last configuration, the value of the variable x is 42, as expected. To start the execution, I used the empty continuation kstop. The final configuration $([x \mapsto 42], \mathsf{skip}, \mathsf{kstop})$ is typical of a terminated execution: both the command part and the continuation indicate that nothing is left to do. In the following, I say that an execution terminates successfully if it ends in a configuration of the form $(\_, \mathsf{skip}, \mathsf{kstop})$.

Concrete languages. In the rest of this dissertation I will use multiple instantiations of the simple generic language just defined. Such a concrete instantiation is provided by

— a set of allowed base statements,

— a semantic interpretation function $[\![\cdot]\!]$ for base statements,

— a set of memory states σ, and

— two "calling convention" relations on memory states entry and exit (c.f., Section 2.6).

2.3 Inference Rules

Invariant logics are program logics obtained with a minor variations on Hoare logic, they can prove the preservation of invariants all along the execution. Safety-critical programs often need to have such invariants proved, for instance, a car-controlling software probably wants to make sure that the car is not braking while the gas pedal is down. But the class of invariants that is most relevant to this dissertation is resource invariants. An example of resource invariant would be that the stack consumption always stays within the range permitted by the hardware. Multiple resource invariants will be used in the following chapters.

Formally, invariants are defined as predicates on memory states. For a given invariant I, the corresponding invariant logic gives a set of rules to derive triples $\vdash_I \{P\}\ c\ \{Q\}$. When the invariant I is trivial (contains all memory states) the logic degenerates to regular Hoare logic and I omit the invariant I from the triple notation. In the triple $\vdash_I\ \{P\}\ c\ \{Q, B\}$, c is a command and the *precondition* P and the *postconditions* Q and B are assertions (i.e., sets of memory states). The intuitive meaning of that triple is that, starting from an initial state σ such that $\sigma \in I \cap P$, the command c will safely execute (i.e., preserve the invariant at all steps), and if it terminates successfully with a memory state σ', then we have $\sigma' \in I \cap Q$.

$$\frac{}{\vdash_I \{Q\}\ \mathsf{skip}\ \{Q, \bot\}}\ (\text{L:}\textsc{Skip}) \qquad \frac{}{\vdash_I \{\sigma \mid \forall \sigma'.\, \sigma \in I \wedge \sigma[\![b]\!]\sigma' \Longrightarrow \sigma' \in I \cap Q\}\ b\ \{Q, \bot\}}\ (\text{L:}\textsc{Base})$$

$$\frac{\vdash_I \{P\}\ c_1\ \{Q, B\} \qquad \vdash_I \{P\}\ c_2\ \{Q, B\}}{\vdash_I \{P\}\ c_1{+}c_2\ \{Q, B\}}\ (\text{L:}\textsc{Alt}) \qquad \frac{}{\vdash_I \{Q\}\ \mathsf{break}\ \{\bot, Q\}}\ (\text{L:}\textsc{Break})$$

$$\frac{\vdash_I \{P\}\ c_1\ \{Q, B\} \qquad \vdash_I \{Q\}\ c_2\ \{R, B\}}{\vdash_I \{P\}\ c_1; c_2\ \{R, B\}}\ (\text{L:}\textsc{Seq}) \qquad \frac{\vdash_I \{P\}\ c\ \{P, B\}}{\vdash_I \{P\}\ \mathsf{loop}\ c\ \{B, \bot\}}\ (\text{L:}\textsc{Loop})$$

$$\frac{P' \subseteq P \qquad \vdash_I \{P\}\ c\ \{Q, B\} \qquad Q \subseteq Q' \qquad B \subseteq B'}{\vdash_I \{P'\}\ c\ \{Q', B'\}}\ (\text{L:}\textsc{Conseq})$$

Figure 2.2: Invariant logic derivation rules for commands.

The full set of rules is given in Figure 2.2. Postconditions are split in two parts $\{Q, B\}$, the Q assertion treats with normal (fallthrough) termination of the command and the B assertion handles breaking out of loops. The role of these two assertions can be observed on the L:Loop rule, where the breaking assertion of the inner command becomes the fallthrough assertion of the whole loop. The invariant preservation is only enforced in the L:Base rule since base statements are the only mean there is to changing the memory state.

$$\frac{}{\vdash_I \{Q, \top\}\ \mathsf{kstop}\ \{Q\}}\ (\text{LK:}\textsc{Stop}) \qquad \frac{\vdash_I \{P\}\ c\ \{Q, B\} \qquad \vdash_I \{Q, B\}\ k\ \{R\}}{\vdash_I \{P, B\}\ \mathsf{kseq}\ c\ k\ \{R\}}\ (\text{LK:}\textsc{Seq})$$

$$\frac{\vdash_I \{P\}\ c\ \{P, B\} \qquad \vdash_I \{B, \bot\}\ k\ \{R\}}{\vdash_I \{P, B\}\ \mathsf{kloop}\ c\ k\ \{R\}}\ (\text{LK:}\textsc{Loop})$$

$$\frac{P' \subseteq P \qquad B' \subseteq B \qquad \vdash_I \{P, B\}\ k\ \{Q\} \qquad Q \subseteq Q'}{\vdash_I \{P', B'\}\ k\ \{Q'\}}\ (\text{LK:}\textsc{Conseq})$$

Figure 2.3: Invariant logic derivation rules for continuations.

One key to use small-step semantics in the soundness proof is to associate logical informations to not only commands but also continuations. An informal justification would go like this. Suppose that we are executing the sequenced command $c_1; c_2$ for which we have the triple $T := \vdash_I \{P\}\ c_1; c_2\ \{Q, B\}$. Then, the first step of any execution for this command is the rule S:Seq1 that transitions to the configuration $C := (\sigma, c_1, \mathsf{kseq}\ c_2\ k)$ where σ and k are, respectively, the initial state and continuation. If logical information is only associated with the commands, we lost what the triple T knew about the command c_2 because it is now in the continuation.

As far as I know, this is the first time logical state is attached to continuations, and as we will see in further sections, it is critical to the soundness proof. Dually to command triples, continuation triples $\vdash_I \{Q, B\}\ k\ \{R\}$ have two preconditions and one postcondition. The two preconditions are necessary because a continuation can be started in two modes: either by breaking out of it, or by falling through it. Similarly to command triples, continuation triples are indexed with an invariant I. Derivation rules for continuations are presented in the Figure 2.3.

2.4 Triple Preservation

In this section, I present the first theorem relating the triples to the operational semantics of the language. The main theorem is similar to the classic subject reduction result used when presenting type safety of functional languages; it shows that the operational semantics preserve logical information.

2.4.1 Theorem

Theorem 1 (Triple preservation)**.** *For any command c, continuation k, and memory state σ, such that*

$$\sigma \in I \cap P,\ \textit{and} \qquad \vdash_I \{P\}\ c\ \{Q, B\},\ \textit{and} \qquad \vdash_I \{Q, B\}\ k\ \{R\};$$

if $(\sigma, c, k) \to (\sigma', c', k')$, then there is P', Q', and B', such that

$$\sigma' \in I \cap P',\ \textit{and} \qquad \vdash_I \{P'\}\ c'\ \{Q', B'\},\ \textit{and} \qquad \vdash_I \{Q', B'\}\ k'\ \{R\}.$$

To prove the theorem, a few inversion lemmas are necessary on the derivation rules of Figures 2.2 and 2.3. I give the flavor of those administrative lemmas with a few examples below. They are proved by induction on the derivations and essentially get rid of stacked weakening applications.

Lemma 1 (L:Skip inversion)**.** *If $\vdash_I \{P\}$ skip $\{Q, B\}$, then $P \subseteq Q$.*

Lemma 2 (L:Loop inversion)**.** *If $\vdash_I \{P\}$ loop c $\{Q, B\}$, then there is P' and B' such that $\vdash_I \{P'\}\ c\ \{P', B'\}$ and $P \subseteq P'$ and $B' \subseteq Q$.*

Proof of Theorem 1. By case analysis on the small-step relation. □

2.4.2 Applications

In this section, I give two applications of Theorem 1. The first one is a functional correctness result that is only applicable to terminating executions. The second one proves that the invariant logic indeed enforces safety and is applicable to both terminating and non-terminating executions.

The typical partial correctness meaning of a Hoare triple is that, started in a state satisfying the precondition, if the command terminates successfully, the final state satisfies the postcondition. The folloing corollary states this results formally.

Corollary 1 (Hoare-logic-like soundness)**.** *If $\vdash \{P\}\ c\ \{Q, B\}$, $\sigma \in P$, and $(\sigma, c, \mathsf{kstop}) \to^* (\sigma', \mathsf{skip}, \mathsf{kstop})$, then $\sigma' \in Q$.*

Proof.

1. By induction on $\to^*$, it is simple to extend the triple preservation Theorem 1 to longer executions.

2. Remark that $\vdash \{Q, B\}$ kstop $\{Q\}$ is derivable for any B using the rules LK:Conseq and LK:Stop.

3. Then, by the extended triple preservation result of step 1, there exists P', Q', B' such that $\vdash \{P'\}$ skip $\{Q', B'\}$ and $\vdash \{Q', B'\}$ kstop $\{Q\}$ and $\sigma' \in P'$.

4. By inversion of the two previous triples, we get $P' \subseteq Q' \subseteq Q$, and thus $\sigma' \in Q$. □

Thanks to the triple preservation result, the proof above is quite simple. However, attempts to prove it directly by induction on the command c, like in textbook soundness proofs, will not work. The direct method fails because, unless continuations get associated with some logical state, the result is not compositional. The continuations triples of Figure 2.3 (p. 23) provide an elegant way to solve that issue.

The previous theorem did not use the invariant I of the logic. I now give a second corollary to justify that if a derivation exists then all the steps taken by the command preserve the invariant.

Corollary 2 (Safety soundness)**.** *If* $\vdash_I \{P\}\ c\ \{Q, B\}$*,* $\sigma \in I \cap P$*, and* $(\sigma, c, \mathit{kstop}) \to^* (\sigma', \_, \_)$*, then* $\sigma' \in I$*.*

Proof. Like in the previous proof, it is consequence of the triple preservation theorem generalized to the $\to^*$ relation. □

As opposed to the first corollary, this safety property applies even to commands that do not terminate. It also states a result on *all* reachable memory states, and not only the final one.

Note in the previous result that the postcondition part of the triple finds no use. However, even to derive pure safety properties, postconditions are useful to the overall compositionality of the logic (e.g., to prove safety of sequentially composed commands). Similarly, in the first corollary, the break part of the postcondition is not used but critical to reason on commands with loops.

I believe that the simplicity of the proofs of the two corollaries in this section and the ease of treatment of non-terminating reactive programs makes a case for using small-step semantics. The only keys unlocking these results were a proper logical accounting of continuations and the triple preservation theorem.

2.5 Generic Completeness and Forward-Closed Properties

In this section, I introduce *forward-closed properties*, a class of properties for which invariant logics are complete. The completeness result is generic and, like in the previous section, I show two possible applications.

2.5.1 Theorem

Definition 1 (Forward-closure)**.** *A property P on runtime configurations is said forward-closed (with respect to the invariant I), when for every pair of runtime configurations such that $(\sigma, c, k) \to (\sigma', c', k')$ and $(\sigma, c, k) \in P$, we have $(\sigma', c', k') \in P$ and $\sigma' \in I$.*

Forward-closed properties are thus preserved by execution and maintain the program invariant I. In fact, the reader is already familiar with forward-closure! Indeed, squinting at the above definition should remind the triple preservation result. Theorem 1 is in fact a proof that the property $\hat{R}$ defined below is forward closed.

$$\hat{R} := \{(\sigma, c, k) \mid \exists P\, Q\, B.\ \ \sigma \in I \cap P \wedge \vdash_I \{P\}\ c\ \{Q, B\} \wedge \vdash_I \{Q, B\}\ k\ \{R\}\}.$$

For each forward-closed property P, continuation k, and command c, define the assertion P_c^k to be $\{\sigma \mid (\sigma, c, k) \in P\}$. Using this assertion, let the P-triple of a command c be $\vdash_I \{P_c^k\}\ c\ \{P_{\mathsf{skip}}^k, P_{\mathsf{break}}^k\}$. The following theorem states that P-triples are always derivable for forward-closed properties.

Theorem 2 (Completeness)**.** *For every forward-closed property P, every command c and every continuation k, the P-triple $\vdash_I \{P_c^k\}\ c\ \{P_{\mathsf{skip}}^k, P_{\mathsf{break}}^k\}$ is derivable.*

Proof. By induction on the command. The fact that the induction hypothesis is applicable for any continuation k is critical for the proof to work. □

2.5.2 Applications

Let us consider the completeness for functional correctness first. To that end, we apply the Theorem 2 to the predicate

$$\hat{Q} := \{(\sigma, c, k) \mid \forall \sigma'.\, (\sigma, c, k) \to^* (\sigma', \mathsf{skip}, \mathsf{kstop}) \implies \sigma' \in Q\}.$$

By definition of $\to^*$, it is simple to prove that this predicate is forward-closed regardless of the assertion Q for the trivial invariant $I = \top$. Intuitively, this is because if a configuration C terminates in a final state satisfying Q, then so does any configuration C' such that $C \to C'$. We are now in position to state the following completeness result.

Corollary 3 (Hoare-logic-like completeness)**.** *If P is an assertion that ensures the postcondition Q for the command c—that is, if for any $\sigma \in P$ such that $(\sigma, c, \mathsf{kstop}) \to^* (\sigma', \mathsf{skip}, \mathsf{kstop})$ we have*

$\sigma' \in Q$—*then* $\vdash \{P\}\ c\ \{Q, \top\}$ *is derivable.*

Proof. 1. By the Theorem 2 applied to $\hat{Q}$, we get that $\vdash \{\hat{Q}_c^{\mathsf{kstop}}\}\ c\ \{\hat{Q}_{\mathsf{skip}}^{\mathsf{kstop}}, \hat{Q}_{\mathsf{break}}^{\mathsf{kstop}}\}$ is derivable.

2. The hypothesis that P ensures the postcondition Q for the command c can be rephrased as the inclusion $P \subseteq \hat{Q}_c^{\mathsf{kstop}}$.

3. Similarly, by definition of the predicate $\hat{Q}$, we have $\hat{Q}_{\mathsf{skip}}^{\mathsf{kstop}} \subseteq Q$.

4. Finally, conclude that $\vdash \{P\}\ c\ \{Q, \top\}$ is derivable by applying one weakening using all the previous steps. □

The Theorem 2 does most of the heavy lifiting in this proof and leaves us with only one simple weakening to apply. Similarly to the soundness result proved earlier, a direct proof by induction on c would fail. The compositionality is provided in Theorem 2 by the universal quantification on the continuation k.

Safety with respect to an invariant I is also a forward-closed property. More formally, the forward-closed predicate is

$$S := \{(\sigma, c, k) \mid \forall \sigma'.\, (\sigma, c, k) \to^* (\sigma', \_, \_) \implies \sigma' \in I\}.$$

Leveraging this, we derive a second completeness result, this time stating completeness of the invariant logics with respect to safety properties.

Corollary 4 (Safety completeness)**.** *If P is an assertion that makes the command c safe—that is, if for any states σ and σ' such that $\sigma \in P$ and $(\sigma, c, \mathsf{kstop}) \to^* (\sigma', \_, \_)$, we have $\sigma' \in I$—then $\vdash_I \{P\}\ c\ \{\top, \top\}$ is derivable.*

Proof. 1. By the Theorem 2 applied to S, we get that $\vdash_I \{S_c^{\mathsf{kstop}}\}\ c\ \{S_{\mathsf{skip}}^{\mathsf{kstop}}, S_{\mathsf{break}}^{\mathsf{kstop}}\}$ is derivable.

2. The hypothesis that P makes the command c safe can be rephrased as the inclusion $P \subseteq S_c^{\mathsf{kstop}}$.

3. Finally, conclude that $\vdash_I \{P\}\ c\ \{\top, \top\}$ is derivable using one weakening application. □

2.6 Procedure Calls

In this section, I add procedure calls to the language and explain how the soundness and completeness results are modified.

2.6.1 Syntax and Semantics

A complete program is now a map from procedure names (an arbitrary set) to commands. Commands are extended to include the new procedure call $\mathsf{call}\ p$, where p is a procedure name.

$$c \coloneqq \dots \mid \mathsf{call}\ p.$$

We also extend continuations to be able to represent call frames.

$$k \coloneqq \dots \mid \mathsf{kcall}\ k.$$

For simplicity, I did not add a return statement. Their handling both in the semantics and in the logic would be very similar to how break is handled. The execution of a procedure returns when the body has been fully executed. I also only considered procedure without arguments and return valuues. Those would only artificially complicate the operational semantics and the reasoning later on without adding any insight. Moreover, arguments and return values can be simulated using global variables.

From now on, I will assume that the program under consideration is Δ—for any procedure name p, $\Delta(p)$ is the procedure body of p. To define the semantics of procedure calls, I will use two "calling convention" relations on memory states, the entry and exit relations. In a concrete instance of this language, the entry relation would save the value of the current local variables on a stack and create a fresh environment for the callee to run. The exit relation would restore the value of the caller's local variables by popping them out of the stack. Such an example is given in Chapter 4.

$$\frac{\mathsf{entry}\ \sigma\ \sigma'}{(\sigma, \mathsf{call}\ p, k) \to (\sigma', \Delta(p), \mathsf{kcall}\ k)}\ (\text{S:Call1}) \qquad \frac{\mathsf{exit}\ \sigma\ \sigma'}{(\sigma, \mathsf{skip}, \mathsf{kcall}\ k) \to (\sigma', \mathsf{skip}, k)}\ (\text{S:Call2})$$

Figure 2.4: Extra rules for the small-step semantics to handle procedure calls.

Using these two new relations, I extend the rules of the operational semantics of Figure 2.1 (p. 20) with the two new rules of Figure 2.4.

2.6.2 New Inference Rules

The biggest modification to the invariant logics required to handle procedure calls is to add a *context*. Triples are now of the shape $\Gamma \vdash_I \{P\}\ c\ \{Q, B\}$, where Γ maps procedure names to a set of procedure specifications. A *procedure specification* is a triple containing an arbitrary set Z and two assertions P_z and Q_z parameterized by a variable $z \in Z$. We write this triple as

$$[\forall z \in Z.\, \{P_z\}\{Q_z\}].$$

The variable z is called *auxiliary state*, and helps linking the precondition P_z to the postcondition Q_z. The choice of the set Z containing the auxiliary variable is left to the user of the logic. Using this auxiliary state, an example specification for a function incrementing a global variable i would be $[\forall z \in \mathbb{N}.\, \{i = z\}\{i = z + 1\}]$.

All the rules presented for commands in Figure 2.2 (p. 22) remain the same except for the fact that they now carry uniformly a fixed context Γ. One new rule L:Call is added to handle function calls:

$$\frac{\begin{array}{c}[\forall z \in Z.\, \{P_z\}\{Q_z\}] \in \Gamma(p) \qquad z \in Z \\ P := \{\sigma \mid \forall \sigma'.\, \mathsf{entry}\ \sigma\ \sigma' \implies \sigma' \in P_z \cap I\} \qquad Q := \{\sigma \mid \exists \sigma'.\, \mathsf{exit}\ \sigma'\ \sigma \wedge \sigma' \in Q_z\}\end{array}}{\Gamma \vdash_I \{P\}\ \mathsf{call}\ p\ \{Q, \bot\}}\ (\text{L:Call}).$$

Note: Add comments about the rule.

Finally, the *validity* of a context expresses that all the procedure specifications it contains are valid specifications for the function bodies. I write $\vdash \Gamma$ to denote that the context Γ is valid and define the notation to mean,

$$\vdash \Gamma := [\forall z \in Z.\, \{P_z\}\{Q_z\}] \in \Gamma(p) \implies \forall z \in Z. \quad \Gamma \vdash_I \{P_z\}\ \Delta(p)\ \{Q_z \cap \mathsf{canexit}, \top\}.$$

Where $\mathsf{canexit} := \{\sigma \mid \forall \sigma'.\, \mathsf{exit}\ \sigma\ \sigma' \implies \sigma' \in I\}$ ensures that when the procedure returns, the invariant is satisfied.

Two things are worth noting in this definition. First, when proving the validity of one specification in the context Γ, the whole context can be taken as hypothesis, i.e., assumed to be true. This is necessary when dealing with mutually recursive procedures. If procedure p calls procedure q and vice versa, their two specifications have to be proved simultaneously. Second, the validity of contexts uses a quantification "$\forall z \in Z$" of the meta-language. Auxiliary state (or meta-variables in other works) is notoriously complicated to handle in program logics and there is a long history of mistakes in its treatment [Nip02]. Using a quantifier of the meta-language is one simple approach that I first saw in a program logic for C by Krebbers [Kre14]. It provides a simple handling of auxiliary state and remains sound and complete, as we will see later.

The meta-language quantification is very natural in the Coq development. Here is the definition for the validity of contexts:

```
Definition valid (Γ: proc_id → proc_spec → Prop): Prop :=
  ∀ p spec (InContext: Γ p spec), (* p is the proc. name and spec is a specification for p *)
  ∀ z, triple Γ (pre p spec z) (P p) (post p spec z ∧ canexit p) (top).
```

With this additional notion of validity for contexts, a complete proof of program for a command c with procedure calls is a pair of one triple $\Gamma \vdash_I \{P\}\ p\ \{Q, B\}$ and one validity proof $\vdash \Gamma$ for the context. The triple justifies that the command implements a specification (the pre and postcondition P and Q) with respect to some assumptions (in the context Γ), and the validity judgement for the context ensures that those assumptions are correct.

Finally, we also need to change the logic rules for continuations in Figure 2.3 (p. 23). Like the rules for commands, they now carry uniformly one fixed context Γ. A new rule LK:Call given below can account for the new type of continuations $\mathsf{kcall}\ p\ k$.

$$\frac{\Gamma \vdash_I \{P, \bot\}\ k\ \{Q\}}{\Gamma \vdash_I \{\sigma \mid \forall \sigma'.\, \mathsf{exit}\ \sigma\ \sigma' \implies \sigma' \in P \cap I, \top\}\ \mathsf{kcall}\ p\ k\ \{Q\}}$$

Note: Discuss the $\top$ and $\bot$ choices.

2.6.3 Triple Preservation

The triple preservation theorem is similar with procedure calls. It simply has one extra hypothesis assuming the validity of the context Γ used in the derivation.

Theorem 3 (Triple preservation with calls)**.** *For any valid context $\vdash \Gamma$, command c, continuation k, and memory state σ, such that*

$$\sigma \in I \cap P, \text{ and } \qquad \Gamma \vdash_I \{P\}\ c\ \{Q, B\}, \text{ and } \qquad \Gamma \vdash_I \{Q, B\}\ k\ \{R\};$$

if $(\sigma, c, k) \to (\sigma', c', k')$*, then there is* P'*,* Q'*, and* B'*, such that*

$$\sigma' \in I \cap P', \text{ and } \qquad \Gamma \vdash_I \{P'\}\ c'\ \{Q', B'\}, \text{ and } \qquad \Gamma \vdash_I \{Q', B'\}\ k'\ \{R\}.$$

Proof. Also by case analysis on the small-step relation. For all the rules but S:Call1 and S:Call2 the proof is exactly the same.

For S:Call1, we reason by inversion of the judgement $\Gamma \vdash_I \{P\}\ \mathsf{call}\ p\ \{Q, B\}$. We set $P' := P_z$, $Q' := Q_z \cap \mathsf{canexit}_p$, and $B' := \top$, where P_z and Q_z are parts of the procedure specification of p used in the L:Call rule. We conclude the existence of the command triple using the validity of the context and the existence of the continuation triple using the LK:Call rule.

For S:Call2, we reason by inversion of the judgement $\Gamma \vdash_I \{Q, B\}\ \mathsf{kcall}\ p\ k\ \{R\}$. There must be Q' such that $(\star)$ $\Gamma \vdash_I \{Q', \bot\}\ k\ \{R\}$, we set $P' := Q'$, $Q' := Q'$, and $B' := \bot$; then we conclude using L:Skip and $(\star)$. □

Using this result, it is straightforward to generalize the applications of Section 2.4.2 to commands containing procedure calls. We simply need to add a context Γ to the derivation and carry the extra hypothesis that it is valid.

Corollary 5 (Hoare-logic-like soundness with calls)**.** *If* $\vdash \Gamma$*,* $\Gamma \vdash \{P\}\ c\ \{Q, B\}$*,* $\sigma \in P$*, and* $(\sigma, c, \mathit{kstop}) \to^* (\sigma', \mathit{skip}, \mathit{kstop})$*, then* $\sigma' \in Q$*.*

Corollary 6 (Safety soundness with calls)**.** *If* $\vdash \Gamma$*,* $\Gamma \vdash_I \{P\}\ c\ \{Q, B\}$*,* $\sigma \in I \cap P$*, and* $(\sigma, c, \mathit{kstop}) \to^* (\sigma', \_, \_)$*, then* $\sigma' \in I$*.*

Proof of the two corollaries. Same as in Section 2.4.2 using the extra hypothesis $\vdash \Gamma$. □

2.6.4 Generic Completeness

The notion of forward closure does not need to be modified to generalize the completeness result to commands with calls. From now on, we assume a forward-closed predicate P.

The extra complexity of the completeness result lies in the choice of the context Γ to use for the P-triples. The solution is, in essence, to use the P-triple of the function body. However, the continuations have to be treated carefully to permit induction. Let the *P-context* Γ_P be defined as follows:

$$\Gamma_P(p) := [\forall k, \{P_{\Delta(p)}^{\mathsf{kcall}\ p\ k}\}\{P_{\mathsf{skip}}^{\mathsf{kcall}\ p\ k}\}].$$

As in Section 2.5, we used the notation P_c^k for the assertion $\{\sigma \mid (\sigma, c, k) \in P\}$. The auxiliary state in the procedure specifications of the P-context is a continuation k. Intuitively, this continuation is the context in which the specified procedure is going to be executed. The use of $\mathsf{kcall}\ p\ k$ in the assertions can be explained by the fact that any englobing continuation for the procedure body $\Delta(p)$ will always be a kcall continuation. At this point, I really must mention that the use of a proof assistant was key in *finding* this somewhat convoluted definition and *trusting* that it is indeed what is needed.

Using a P-context, we can generalize the generic completeness Theorem 2 as follows.

Theorem 4 (Completeness with calls). *For every forward-closed property P, every command c and every continuation k, the P-triple $\Gamma_P \vdash_I \{P_c^k\}\ c\ \{P_{\mathit{skip}}^k, P_{\mathit{break}}^k\}$ is derivable.*

Proof. There is only one new case compared to the proof of Theorem 2, the call case. We treat it using the definition of the P-context Γ_P, and the forward closure of P on $(\sigma, \mathsf{call}\ p, k) \to (\sigma', \Delta(p), \mathsf{kcall}\ p\ k)$ and $(\sigma, \mathsf{skip}, \mathsf{kcall}\ p\ k) \to (\sigma', \mathsf{skip}, k)$. □

The previous theorem alone is not quite enough to have a satisfactory completeness result, indeed, the context Γ_P might contain invalid assumptions. Maybe surprisingly, the proof that Γ_P is valid boils down to an application of Theorem 4.

Theorem 5 (Validity of P-contexts). *The P-context Γ_P is valid: $\vdash \Gamma_P$.*

Proof. For any procedure p, we have to prove the triple $\Gamma_P \vdash_I \{P_{\Delta(p)}^{\mathsf{kcall}\ p\ k}\}\ \Delta(p)\ \{P_{\mathsf{skip}}^{\mathsf{kcall}\ p\ k} \cap \mathsf{canexit}_p, \top\}$. This is pretty much exactly what Theorem 4 provides us. We prove that the $P_{\mathsf{skip}}^{\mathsf{kcall}\ p\ k} \subseteq \mathsf{canexit}_p$ by using the forward closure of P. □

Combining the two previous results we prove, like in Section 2.5.2, completeness results for both the safety and functional correctness interpretations of triples. The results are exactly the same as the ones proved before but they assert the existence of a valid context in which the completeness triple holds.

Corollary 7 (Safety completeness with calls). *If P is an assertion such that for any states σ and σ' such that $\sigma \in P \cap I$ and $(\sigma, c, \mathit{kstop}) \to^* (\sigma', \_, \_)$ we have $\sigma' \in I$, then there is a valid context $\vdash \Gamma$ such that $\Gamma \vdash_I \{P\}\ c\ \{\top, \top\}$ is derivable.*

Corollary 8 (Hoare-logic-like completeness with calls). *If P is an assertion such that for any $\sigma \in P$ such that $(\sigma, c, \mathit{kstop}) \to^* (\sigma', \mathit{skip}, \mathit{kstop})$ we have $\sigma' \in Q$, then there is a valid context $\vdash \Gamma$ such that $\Gamma \vdash \{P\}\ c\ \{Q, \top\}$ is derivable.*

Proof of the two corollaries. Same as in Section 2.5.2 but using Γ_P with P respectively set to the forward-closed predicates S and $\hat{Q}$. The validity of the P-context is proved by Theorem 5 and the existence of the triple is proved by Theorem 4. $\square$

Chapter 3

Low-Level Stack Bounds Using a High-Level Logic

In this chapter, I present the first framework for deriving formally verified end-to-end stack-space bounds for C programs. Stack bounds are particularly interesting because stack overflow is "one of the toughest (and unfortunately common) problems in embedded systems" [Exp14]. Moreover, stack-memory is the only dynamically allocated memory in many embedded systems and the stack usage depends on the implementation of the compiler.

I begin by introducing the concept of resource safety and use it, in combination with an invariant logic, to prove bounds on the stack usage of high-level programs. Then I motivate and define a new quantitative logic designed to prove stack bounds on the same high-level programs. Used in combination with Quantitative CompCert, a resource-aware compiler for C programs, the bounds derived using the quantitative logic at the source level can be transported to the assembly program that results from the compilation. In addition to the conference version of this work, I present a proof of completeness of the program logic.

3.1 Language Definition

Like in Chapter 2, I am going to use a simple abstract language for the presentation of this chapter. In the Coq implementation of the stack analysis system, a large subset of Clight is suported. For any concrete instance L_0 of the abstract language of Section 2.2, we are going to define another concrete language L that tracks the stack consumption. Remember that L_0 is given by a set of base statements, a set of states Σ_0, a semantic function for base statements $[\![\cdot]\!]_0$, and a procedure entry and exit relations entry_0 and exit_0. Then, the stack-tracking language L is given by:

— a set of base statements identical to the one of L_0;

— a set of states $\Sigma = \Sigma_0 \times \mathbb{Z}$;

— a semantic for base statements $[\![\cdot]\!]$ such that $(\sigma, n)[\![b]\!](\sigma', n')$ when $n = n'$ and $\sigma[\![b]\!]_0\sigma'$;

— a procedure entry relation such that $\mathsf{entry}\ p\ (\sigma, n)\ (\sigma', n')$ when $n' = n - M(p)$ and $\mathsf{entry}_0\ p\ \sigma\ \sigma'$;

— a procedure exit relation such that $\mathsf{exit}\ p\ (\sigma, n)\ (\sigma', n')$ when $n' = n + M(p)$ and $\mathsf{exit}_0\ p\ \sigma\ \sigma'$.

The idea of this instrumentation is to add a "stack counter" that keeps track of the *stack space available* at all times. The unit for this counter can be arbitrary but, in practice, using bytes is the most natural choice. When a procedure is entered, the amount of stack available is decreased by the size of its stack frame. When the procedure is left, this same amount of stack is made available by adding it to the counter. I used the *resource metric M* to map a procedure name p to the frame size $M(p)$ of the corresponding procedure. In practice M is not available at reasoning time, so all the theorems proved are universally quantified over this function. The resource-aware compiler that I describe in Section 3.4 is responsible for providing the resource metric.

3.2 Resource Safety

3.2.1 Resource Bound Inference as a Safety Problem

Let me start with an inspirational quote from *The Art of Designing Embedded Systems* of Jack Ganssle [Gan00].

> "With experience, one learns the standard, scientific way to compute the proper size for a stack: Pick a size at random and hope."

While the advocated technique might appear unscientific at first, there is a way to exploit it fruitfully by defining the concept of resource safety. Using the stack-tracking operational semantics of the language defined in the previous section, we can "run" the program and see if it does not run out of stack. Running out of stack is indicated by a stack counter becoming negative. Thus, following Ganssle I define a valid resource bound B as one such that the program running with an initial stack of size B will not crash. In other words, we can find resource bounds by proving a safety property. The stack usage of a program is then the smallest stack size that will allow it to run safely.

Consequently, define the *resource-safety invariant* $I \subseteq \Sigma$ as

$$I := \{(\sigma, n) \mid n \geqslant 0\}.$$

Define a resource-safe configuration as one satisfying the resource-safety invariant. And write that a command c is resource-safe for an initial state σ and a stack size S, when all configurations reachable from $(\sigma, S, c, \mathsf{kstop})$ are resource-safe configurations. Ganssle's ironical statement is now more formally stated as:

$$\boxed{\begin{array}{l}\text{The number } S \in \mathbb{N} \text{ is a proper stack size for the command } c \text{ if the configuration} \\ (\sigma, S, c, \mathsf{kstop}) \text{ is resource safe for any initial state } \sigma \in \Sigma_0.\end{array}}$$

And thus, using the completeness Corollary 7 from Chapter 2, S is a proper stack size for c if the triple $\vdash_I \{(\sigma, n) \mid n = S\}\ c\ \{\top, \top\}$ is derivable. In practice, it is preferable to have a parametric bound B that maps initial states in Σ_0 to bounds in $\mathbb{N}$. In that case, $B(\sigma)$ is the stack bound of the command c started in the initial state σ.

Dually, by the soundness of invariant logics, if the triple $\vdash_I \{P\}\ c\ \{\_, \_\}$ is derivable, then B is a proper stack size for the command c started in all memory states σ such that $(\sigma, B) \in P$.

A note about regular safety. Note that the resource-safety of a program does not imply that the program is free of all crashes. It is only free of "resource crashes." For example, resource-safety can be proved for programs that make invalid memory accesses or divide by zero. This means that, when proving the complete correctness of a program, one must prove regular safety *and* resource safety independently. One way to do this is to have base statements of the language transition to an error state $\lightning$ when they fail and use an invariant logic with an invariant I that excludes this error state. In the sequel, I only focus on pure resource safety.

3.2.2 Stack Bounds using an Invariant Logic

In this section, I explain how to use the resource-safety invariant in combination with a logic of Chapter 2 to prove stack-usage bounds on some simple commands.

Preconditions and stack bounds. When deriving a stack-usage bounds for the command c, what should the precondition of the triple derived with the invariant logic look like? Since we are interested in stack bounds, it seems reasonable to require it to be $\{(\sigma, n) \mid n \geqslant B\}$, for some bound B; that is, to permit all initial configurations where the stack size is at least B. Remember that the n component of the state is the amount of stack available. In practice, though, some recursive procedures might need a stack size that depends on the initial memory state, so we should use the more general version $\{(\sigma, n) \mid n \geqslant B(\sigma)\}$.

Base statements. Let us try to derive a triple for an arbitrary base statement b. The base statement b does not consume nor releases any stack, so what triple should we derive for b? Because of the remarks in the previous paragraph, it seems that the triple $T := \vdash_I \{n \geqslant 0\}\ b\ \{n \geqslant 0, \bot\}$ must be a valid specification for b. I used the notation $\{n \geqslant 0\}$ as a shorthand for the set $\{(\sigma, n) \mid n \geqslant 0\}$; and in general, I will now drop the first part of the set comprehension notation when it is clear what the intended meaning is. Remember the rule L:Base from Figure 2.2 (p. 22)

$$\frac{}{\vdash_I \{(\sigma, n) \mid \forall \sigma'\, n'.\, (\sigma, n) \in I \wedge (\sigma, n)[\![b]\!](\sigma', n') \Longrightarrow (\sigma', n') \in I \cap Q\}\ b\ \{Q, \bot\}}.$$

And recall that $(\sigma, n)[\![b]\!](\sigma', n')$ is defined to be "$n = n'$ and $\sigma[\![b]\!]_0\sigma'$". Since Q is $\{n \geqslant 0\}$, and I is $\{(\sigma, n) \mid n \geqslant 0\}$, the precondition is always true. And we can derive the triple T with an application of the consequence rule.

In general, however, the triple T is not precise enough for at least two reasons. First, it loses track of the action of the base statement b on the memory state. If further computations depend on this result, we might lack information to prove a stack bound. Second, it does not ensure that the stack size before and after the statement are the same. To fix these two issues, we can change the specification of b to be the following family of triples:

$$\vdash_I \{n \geqslant z \wedge \forall \sigma', \sigma[\![b]\!]_0\sigma' \Longrightarrow \sigma' \in Q\}\ b\ \{n \geqslant z \wedge \sigma \in Q, \bot\}.$$

The family is indexed by Q (all the possible postconditions) and z (all the possible stack sizes). I use z as a meta-variable to relate the stack size in the pre and postcondition. This letter choice is not innocent: I used z for auxiliary state in procedure specifications in Section 2.6, and as we will see below, the auxiliary state is handily used to generalize over the caller's "context", and its stack size in particular. Note that z can be negative in this specification, it is however useless in practice as the soundness Theorem 3 will only ever apply to initial states with a non-negative stack size (i.e., matching the resource-safety invariant).

Chaining procedure calls. Assume that a program contains two procedures p and q that do not change the memory state but have non-null stack needs. They could be used to output some text for instance. If we call them in sequence, we would like to be able to show that the stack consumption does not exceed $B := \max(M(p), M(q))$.

To derive a triple about a command containing calls, we need to setup a context Γ of hypotheses about the various procedures called. In the example, the procedures p and q need specifications in the context Γ. We will use the same specification for both procedures:

$$[\forall z \in \mathbb{Z}_0^+, (n \geqslant z, n \geqslant z)].$$

This specification simply says that, across a call, the stack space available does not decrease. As we will see shortly, the stack accounting is done in the caller, thus this specification does not have to mention the stack sizes of p and q.

Using this specification for p, we can show that the triple $T_p := \vdash_I \{n \geqslant B\}$ $\mathsf{call}\ p$ $\{n \geqslant B, \bot\}$ holds. To do so we apply the L:Call rule with z set to $B - M(p)$. This yields

$$\begin{aligned}\vdash_I\ \ & \{\forall \sigma'\, n'.\, \mathsf{entry}_0\ p\ \sigma\ \sigma' \wedge n' = n - M(p) \Longrightarrow n' \geqslant 0 \wedge n' \geqslant B - M(p)\}\\ & \mathsf{call}\ p\\ & \{\exists \sigma'\, n'.\, \mathsf{exit}_0\ p\ \sigma'\ \sigma \wedge n = n' + M(p) \wedge n' \geqslant B - M(p), \bot\}.\end{aligned}$$

Regardless of how the entry calling convention relation acts on the state, if $n \geqslant B$, then $n' = n - M(p) \geqslant 0$ (because $B = \max(M(p), M(q)) \geqslant M(p)$), and $n' \geqslant B - M(p)$. This proves that the assertion $\{n \geqslant B\}$ is included in the precondition. By another simple reasoning, the postcondition of the above triple is included in $\{n \geqslant B\}$. Thus, by the consequence rule, the triple T_p is derivable.

The same argument shows that $T_q := \vdash_I \{n \geqslant B\}$ $\mathsf{call}\ q$ $\{n \geqslant B, \bot\}$ is derivable, and, by the sequence rule L:Seq, we can compose the two triples to prove the goal $\vdash_I \{n \geqslant \max(M(p), M(q))\}$ $\mathsf{call}\ p;$ $\mathsf{call}\ q$ $\{n \geqslant \max(M(p), M(q)), \bot\}$. Note that this triple not only bounds the stack usage of the sequenced calls but also proves the preservation of the original stack size via the postcondition.

3.2.3 Monotonicity of Resource-Safety and Bound Algebra

Here is an essential property of resources: having more resources will never degrade the behavior of a program. Formally, we can prove by induction on the small-step semantics that if (σ, n, c, k) is a resource-safe configuration, then so is (σ, n', c, k) with $n' \geqslant n$. Resource-safety is a *monotonic* property. This validates formally the obvious fact that if B is a valid stack-bound, so is $B' \geqslant B$. This observation can be put in correspondance with the consequence rule of the invariant logic, if P a precondition making a command safe, then so is $P' \subseteq P$. Similarly, we can link other logical operators on assertions with numerical operators on bounds.

Assertions	$\subseteq$	$\cap$	$\cup$	$\perp$	$\top$
Bounds	$\geqslant$	max	min	∞	0

Let me make the correspondance more formal by introducing two mappings. To one assertion $P \subseteq \Sigma_0 \times \mathbb{Z}$, I associate the resource bound $B_P(\sigma) := \min\{n \mid (\sigma, n) \in P \wedge n \geqslant 0\}$; and to one resource bound B, I associate the assertion $P_B := \{(\sigma, n) \mid n \geqslant B(\sigma)\}$. Those two correspondances map one construct from the "Assertions" column above to the corresponding one in the "Bounds" column, and vice versa. I now give two lemmas stating this result precisely for the first column of the table.

Lemma 3. *If $P \subseteq Q \subseteq \Sigma_0 \times \mathbb{Z}$, then for any state $\sigma \in \Sigma_0$, $B_P(\sigma) \geqslant B_Q(\sigma)$.*

Proof. Any n such that $(\sigma, n) \in P$ also satisfies $(\sigma, n) \in Q$, thus $B_P(\sigma) = \min\{n \mid (\sigma, n) \in P \wedge n \geqslant 0\} \geqslant \min\{n \mid (\sigma, n) \in Q \wedge n \geqslant 0\} = B_Q(\sigma)$. □

Lemma 4. *If for all $\sigma \in \Sigma_0$, $B(\sigma) \geqslant C(\sigma)$, then $P_B \subseteq P_C$.*

Proof. Assume (σ, n) is such that $n \geqslant B(\sigma)$, then $n \geqslant C(\sigma)$, thus $(\sigma, n) \in P_C$. □

The proofs of the two lemmas have a visual explanation in Figure 3.1 (p. 40).

The remarks of this section are suggestive that bounds themselves can be seen as logical objects. As such, one can wonder if they respect disciplined logical rules like program assertions do. In the next section, I answer positively to this question by introducing a *quantitative logic*: a logic dealing directly with resource bounds.

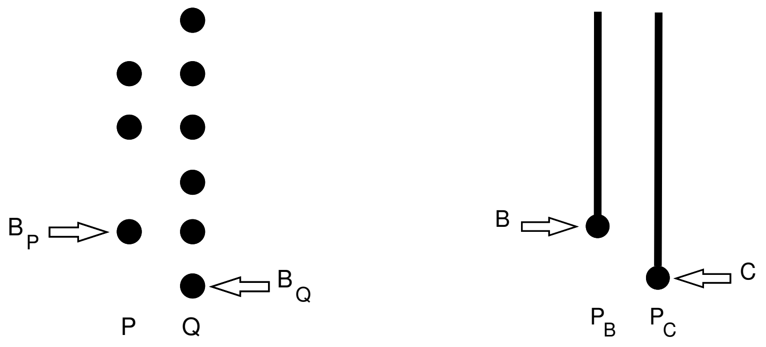

Figure 3.1: Illustrations of the two lemmas with σ component of the state fixed fixed. On the left, the black dots represent the numbers n such that $(\sigma, n) \in P$ or $(\sigma, n) \in Q$. The minimum such numbers are pointed by an arrow. On the right, the dots stand for B and C and the lines represent the sets of numbers n such that $(\sigma, n) \in P_B$ and $(\sigma, n) \in P_C$. On both figures, n increases upwards.

3.3 A Quantitative Logic for Stack Usage

In this section, I present the first quantitative logic of this dissertation. It derives stack-usage information about programs by manipulating the stack-space bounds directly. The quantitative logic is proved sound and complete using a translation technique. And finally, the bounds inferred in Section 3.2.2 are recovered using the quantitative logic.

3.3.1 Inference Rules

Assertions. Instead of the usual assertions as sets of memory states, the logic uses parametric resource bounds

$$A \in (\Sigma_0 \to \mathbb{N} \cup \{\infty\}).$$

As suggested at the end of the previous section, bounds can also be understood as a refinement of the typical assertions where $\bot$ is interpreted as $(\_ \mapsto \infty)$, $\top$ is interpreted as $(\_ \mapsto 0)$, etc.

Like for the invariant logic, the quantitative logic has two postconditions, one for the fallthrough termination of the code enclosed in the triple, and another for the break termination. To handle procedure calls, I also follow the invariant logic and use a context Γ that maps each procedure name to a set of specifications. Procedure specifications are triples

$$[\forall z \in Z, \{P_z\}\{Q_z\}],$$

where Z is an arbitrary set of auxiliary states and P_z and Q_z are bounds that can depend on some auxiliary state $z \in Z$.

In summary, a quantitative triple has the form

$$\Gamma \vdash \{P\}\ c\ \{Q, B\},$$

where Γ is the procedure context, and P, Q, and B are resource bounds in $\Sigma_0 \to \mathbb{N} \cup \{\infty\}$. Intuitively, an assertion is a *potential function* that maps a memory state to a non-negative potential. The potential of the precondition P must be sufficient to cover the resource cost of the execution of the command c, and the potential Q or B after the execution of c.

Rules. In the rules of the quantitative logic, I use the usual extention of the operations $+$ and $\geqslant$ on $\mathbb{N} \cup \{\infty\}$. That is, $\infty + n = n + \infty = \infty$ and $\infty \geqslant n$ for all $n \in \mathbb{N} \cup \{\infty\}$. I also use the least upper bound and greatest lower bound operations max and min on the lattice $\mathbb{N} \cup \{\infty\}$. Finally, I define $\bot(\sigma) := \infty$ and $\top(\sigma) := 0$ for all σ. The full set of rules is given in Figure 3.2 (p. 42). In the following, I explain informally the intuition behind some of the rules.

In the rule Q:Skip, we do not have to account for any stack consumption. Thus, the constraint on the precondition potential is that it is sufficient to pay for the fallthrough postcondition. In the logic, to keep rules orthogonal, I actually constrain it to be exactly equal to the postcondition; excess potential can however be added to the precondition using the consequence rule. The break postcondition is set to $\bot$ since a skip command always falls through.

The rule Q:Break is similar to the rule Q:Skip. The only difference is that the role of the fallthrough and the break postconditions are swapped.

The rule Q:Base accounts for all state-changing commands of the language. It uses the classic "backwards" style of Hoare logic: The precondition bound is enforced to be as large as the postcondition bound on all the memory states reachable after executing the base statement b. It is the essence of a worst-case reasoning.

The rules Q:Alt, Q:Seq, and Q:Loop are taken verbatim out of the invariant logic. These rules (like Q:Skip and Q:Break) are purely about the control flow of the program; thus many, if not all, Hoare-based program logics will have them identical. This can be illustrated by translating the program to a control-flow graph. What these rules ensure is that program "points" connected by pure flow edges get the same logical specification.

$\boxed{\vdash \Gamma}$

$$\vdash \Gamma \quad := \quad [\forall z \in Z.\, \{P_z\}\{Q_z\}] \in \Gamma(p) \quad \Longrightarrow \quad \forall z \in Z. \quad \Gamma \vdash \{P_z\}\ \Delta(p)\ \{Q_z, \top\}$$

$\boxed{\Gamma \vdash \{P\}\ c\ \{Q, B\}}$

$$\frac{}{\vdash \{Q\}\ \mathsf{skip}\ \{Q, \bot\}}\ (\text{Q:}\textsc{Skip}) \qquad \frac{}{\vdash \{\sigma \mapsto \max\{Q(\sigma') \mid \sigma[\![b]\!]\sigma'\}\}\ b\ \{Q, \bot\}}\ (\text{Q:}\textsc{Base})$$

$$\frac{\vdash \{P\}\ c_1\ \{Q, B\} \qquad \vdash \{P\}\ c_2\ \{Q, B\}}{\vdash \{P\}\ c_1 {+} c_2\ \{Q, B\}}\ (\text{Q:}\textsc{Alt}) \qquad \frac{}{\vdash \{Q\}\ \mathsf{break}\ \{\bot, Q\}}\ (\text{Q:}\textsc{Break})$$

$$\frac{\vdash \{P\}\ c_1\ \{Q, B\} \qquad \vdash \{Q\}\ c_2\ \{R, B\}}{\vdash \{P\}\ c_1; c_2\ \{R, B\}}\ (\text{Q:}\textsc{Seq}) \qquad \frac{\vdash \{P\}\ c\ \{P, B\}}{\vdash \{P\}\ \mathsf{loop}\ c\ \{B, \bot\}}\ (\text{Q:}\textsc{Loop})$$

$$\frac{\begin{array}{c}[\forall z \in Z.\, \{P_z\}\{Q_z\}] \in \Gamma(p) \qquad z \in Z \\ P(\sigma) := \max\{P_z(\sigma') \mid \mathsf{entry}_0\ p\ \sigma\ \sigma'\} \qquad Q(\sigma) := \min\{Q_z(\sigma') \mid \mathsf{exit}_0\ p\ \sigma'\ \sigma\}\end{array}}{\vdash \{P + M(p)\}\ \mathsf{call}\ p\ \{Q + M(p), \bot\}}\ (\text{Q:}\textsc{Call})$$

$$\frac{P' \geqslant P \qquad \vdash \{P\}\ c\ \{Q, B\} \qquad Q \geqslant Q' \qquad B \geqslant B'}{\vdash \{P'\}\ c\ \{Q', B'\}}\ (\text{Q:}\textsc{Conseq})$$

Figure 3.2: Quantitative logic derivation rules in a context Γ (elided) with a resource metric M. An assertion maps one memory state to an integer potential. The potential of the precondition "pays" for the stack consumption of the command and for the potential of the postcondition, enabling compositional reasoning.

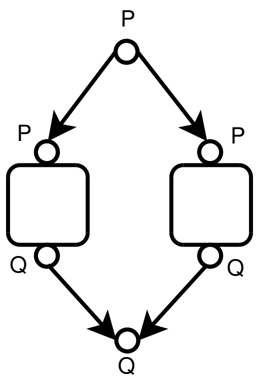

Figure 3.3: Control flow for an alternative command $c_1{+}c_2$ with bound annotations.

Let me illustrate this last remark by considering the control flow graph representation of an alternative command $c_1{+}c_2$. This graph is depicted in the Figure 3.3. The action of the Q:Alt rule on this figure is simply to enforce that all the points connected by arrows get the same potential function. A similar argument can be given for the sequence rule Q:Seq. The rule Q:Loop would require a slightly more complex compilation to flow graphs to handle `break` commands, but the essence of this observation would be preserved. This discussion will be enlightened and formalized in the Chapter 5 where I treat directly the case of low-level programs.

The Q:Call rule accounts for the actual stack-space usage of programs. It enforces that enough stack space is available to call the procedure p by adding $M(p)$ to the precondition and the postcondition. The pre and postcondition are taken from the context Γ and instantiated with an arbitrary auxiliary state $z \in Z$. Note the symmetry of the rule Q:Call: $M(p)$ is added on both sides to account for the stack space that is first consumed and then becomes available after the call.

Finally, the rule Q:Conseq is a quantitative adaptation of the typical consequence rule of Hoare logic. It uses the correspondance I described in Section 3.2.3. It can be explained simply by considering the nature of a "bound." If for a state σ, $P(\sigma)$ is a bound on the stack usage and $P' \geqslant P$, then clearly $P'(\sigma)$ is also a bound on the stack usage. The inequality is the opposite for the postconditions because they are not "requirements," like the precondition, but "guarantees" instead. (A guarantee on the amount of stack available after the execution of the command.)

Framing. In the conference version of this work a framing rule was also part of the logic.

$$\frac{\vdash \{P\}\ c\ \{Q, B\} \qquad x \geqslant 0}{\vdash_I \{P + x\}\ c\ \{Q + x, B + x\}}\ \text{Q:Frame}$$

The frame rule is, like the consequence rule, non-structural: it can be applied to any command. It is a simple version of the frame rule of separation logic and says, intuitively, that resource-safe program executions can always be changed to have more resource available. Additionally, the extra resource we put in can be retrieved at the end of the execution. The frame rule is sound because it is admissible (any derivation using the frame rule can be rewritten without it). However, I do not think that it is derivable from other rules. In particular, attempting to prove it using the consequence rule would fail, indeed $P + x \geqslant P$ is always true when $x \geqslant 0$, but not $Q \geqslant Q + x$.

I chose to leave the frame rule out of the quantitative logic. Indeed, in the next section we will see that the quantitative logic is already complete without it. In some cases, using the frame rule can be useful to prove resource-safety; since it is sound, we are free to use it, and we can even add it as a first-class rule in the quantitative logic but it would not make any new triple provable.

3.3.2 Metatheory

By mere superficial observation of the quantitative logic, the similarities with invariant logics are already striking. In the Section 3.2.3 I gave mappings to go from a traditional assertion to a resource bound and back. In this section, I show that these correspondances allow to go from one logic to the other. These correspondances are then used to piggyback the proof burden of soundness and completeness on work already done.

Remember that we can go from one assertion $P \subseteq (\Sigma_0 \times \mathbb{Z}) = \Sigma$ to a bound $\lfloor P \rfloor \in (\Sigma_0 \to \mathbb{N} \cup \{\infty\})$ by defining

$$\lfloor P \rfloor(\sigma) := \min\{n \mid (\sigma, n) \in P \wedge n \geqslant 0\}.$$

And conversely, we can go from a bound B to an assertion $\lceil B \rceil$ using

$$\lceil B \rceil := \{(\sigma, n) \mid n \geqslant B(\sigma)\}.$$

These transformations are naturally extended to contexts Γ by applying them to the two assertions of all the procedure specifications in the context.

Now I state the theorems mapping a quantitative logic deriviation to an invariant logic derivation and vice versa.

Theorem 6. *If the triple $\Gamma \vdash \{P\}\ c\ \{Q, B\}$ is derivable in the quantitative logic, then $\lceil \Gamma \rceil \vdash_I \{\lceil P \rceil\}\ c\ \{\lceil Q \rceil, \lceil B \rceil\}$ is derivable in the invariant logic for resource safety.*

Proof sketch. By induction on the derivation of the triple in the quantitative logic. All the control-flow rules are simple to handle as they are identical in the two logics. The Lemma 4 justifies the consequence rule. The rule Q:CALL is the most interesting case. The essence of its proof is that the addition of $M(p)$ in the pre and postcondition of the quantitative logic is matched with the entry and exit calling conventions in the invariant logic. The addition of $M(p)$ in the precondition ensures that the entry calling convention leads to a state in the resource-safety invariant. □

Theorem 7. *If the triple $\Gamma \vdash_I \{P\}\ c\ \{Q, B\}$ is derivable in the invariant logic for resource safety, then $\lfloor\Gamma\rfloor \vdash \{\lfloor P\rfloor\}\ c\ \{\lfloor Q\rfloor, \lfloor B\rfloor\}$ is derivable in the quantitative logic.*

Proof sketch. By induction on the derivation of the triple in the invariant logic. All the control-flow rules are simple to handle as they are identical in the two logics. The Lemma 3 justifies the consequence rule. In general, we simply invert the arguments in the proof of Theorem 6. □

Using these two theorems, we can prove that the transformation of contexts also preserves validity.

Theorem 8. *If $\vdash \Gamma_q$ is a valid context in the quantitative logic and $\vdash_I \Gamma_i$ is a valid context in the invariant logic, then $\vdash \lfloor\Gamma_i\rfloor$ and $\vdash_I \lceil\Gamma_q\rceil$.*

Proof. Apply the two previous theorems. To prove $\vdash_I \lceil\Gamma_q\rceil$, we have to show that the postconditions translated from the quantitative logic are sufficient to ensure that the resource-safety invariant is restored after the function exit. This is true because the function exit will *increase* the stack counter of a state that is already resource-safe. □

By chaining Theorem 6 and the soundness theorem of the invariant logic, we obtain the soundness of the quantitative logic.

Theorem 9 (Soundess of the stack quantitative logic)**.** *If $\vdash \Gamma$ and $\Gamma \vdash \{P\}\ c\ \{Q, B\}$ in the quantitative logic, if σ is a memory state and n is an integer such that $n \geqslant P(\sigma)$, then for any n' such that $(\sigma, n, c, \mathsf{kstop}) \to^* (\_, n', \_, \_)$, we have $n' \geqslant 0$. That is, P is a valid stack bound for c.*

Proof. By Theorems 6 and 8, we have $\vdash_I \lceil\Gamma\rceil$ and $\lceil\Gamma\rceil \vdash_I \{\lceil P\rceil\}\ c\ \{\lceil Q\rceil, \lceil B\rceil\}$; then by soundness of the invariant logic, we know that for any $(\sigma, n) \in \lceil P\rceil$ and n' such that $(\sigma, n, c, \mathsf{kstop}) \to^* (\_, n', \_, \_)$, we have $n' \geqslant 0$. But by unfolding the $\lceil P\rceil$ we have, $(\sigma, n) \in \lceil P\rceil$ if and only if $n \geqslant P(\sigma)$. □

Dually, using the other transformation we conclude the completeness of the quantitative logic.

Theorem 10 (Completeness of the stack quantitative logic)**.** *If P is a valid bound for c, then there is a valid context $\vdash \Gamma$ such that the triple $\Gamma \vdash \{P\}\ c\ \{\top, \top\}$ is derivable.*

Proof. By completeness of the invariant logic, there is a valid context $\vdash_I \Gamma$ such that $\Gamma \vdash_I \{\lceil P \rceil\}\ c\ \{\top, \top\}$ is derivable in the invariant logic. We can then apply Theorems 7 and 8 to obtain $\vdash \lfloor \Gamma \rfloor$ and a derivable triple $\lfloor \Gamma \rfloor \vdash_I \{\lfloor \lceil P \rceil \rfloor\}\ c\ \{\top, \top\}$. But by definition of the two transformations, $\lfloor \lceil P \rceil \rfloor = P$. (Note that applying the tranformations in the opposite order would not necessarily preserve assertions.) $\square$

To sum up the results about the quantitative logic, I defined in Section 3.2.3 two natural transformations going from stack bounds to assertions, and back. Then I observed that these two transformations naturally transform triples and contexts from the quantitative logic to the invariant logic, and back. Not only triples are translated, but the Theorems 6 and 7 prove that the derivability is also preserved. Finally, I used this correspondance to get soundness and completeness of the quantitative logic "for free."

3.3.3 Why Bother with a Quantitative Logic?

Since invariant logics are already able to prove stack bounds, one might wonder what the use of a quantitative logic is. In this section, I give two reason why quantitative logics are worth presenting.

Structure versus expressivity. When designing a logic—and similarly, a type system—one always aims to strike a balance between expressivity and ease of use. The more structured a logic is, the better it guides its user through reasoning and the easier it is to use. Consequently, structured logics are also more suitable to automation. On the other hand, a logic that is too constrained can loose expressivity by forbidding certain facts to be derived.

The quantitative logic is interesting because it does better on the structure/expressivity tradeoff than an invariant logic. Indeed, with Theorem 10, I showed that no expressivity is lost compared to an invariant logic. And in addition, the quantitative logic provides a more structured mean to reason.

— Instead of considering arbitrary assertions on the memory state and the resource counter, we constrain them to be of the form $B(\sigma) \geqslant n$, where B is a bound and n is the resource counter. So the interactions between the resource counter and the memory state are more limited in the quantitative logic.

— In the quantitative logic for stack, we can explicitely encode the stack discipline in the Q:Call rule. An invariant logic would not give an account as clear as the quantitative logic does since the stack-counter changes would be hidden in the entry and exit calling conventions.

Potential reasoning. Quantitative logics are the proper "Programming Language" formalization of the concept of proofs using the potential method. The "bounds" manipulated by the quantitative logic are the same as potential functions defined in potential-based reasoning. They are both storing an amount of stashed resource units that can be used to pay for the cost of operations in the program. Thus, quantitative logics are one well-defined formal setting in which potential-based proofs can be expressed. This insight should become clearer as I give more examples in further chapters.

3.3.4 Stack Bounds using the Quantitative Logic

In this section I revisit examples from Section 3.2.2 and express them in the quantitative logic. Since we have seen an explicit translation to the quantitative logic, it should be no surprise that the same bounds are derivable.

Base statements. In Section 3.2.2, using the invariant logic, we derived the following family of resource specification for a base statement b.

$$\vdash_I \{n \geqslant z \wedge \forall \sigma', \sigma[\![b]\!]_0\sigma' \implies \sigma' \in Q\}\ b\ \{n \geqslant z \wedge \sigma \in Q, \bot\}. \tag{3.1}$$

The postcondition $\{n \geqslant z \wedge \sigma \in Q\}$ is not of the form $\{n \geqslant B(\sigma)\}$ that would be easier to translate. But the table in Section 3.2.3 shows how to embed regular assertations into the language of resource bounds: define $B_Q(\sigma)$ to be 0 when $\sigma \in Q$ and ∞ otherwise. Then for any z, the rule Q:Base proves

$$\vdash \{\sigma \mapsto \max\{\max\{z, B_Q(\sigma')\} \mid \sigma[\![b]\!]\sigma'\}\}\ b\ \{\max\{z, B_Q\}, \bot\}.$$

Since B_Q is always 0 or ∞, we have $\max\{z, B_Q(\sigma)\} = z + B_Q(\sigma)$, and we in fact have

$$\vdash \{\sigma \mapsto z + \max\{B_Q(\sigma') \mid \sigma[\![b]\!]\sigma'\}\}\ b\ \{z + B_Q, \bot\}.$$

This triple is equivalent to the triple (3.1) derived using the invariant logic.

Chaining procedure calls. We are here interested in deriving a bound for the command "$\mathsf{call}\ p; \mathsf{call}\ q$" for p and q two arbitrary leaf procedures (thus with constant stack usage). The desired bound is $B := \max(M(p), M(q))$. Like when using the invariant logic, the first thing to do is to give specifications for the two procedures. We give them both the following specification

$$[\forall z \in \mathbb{Z}_0^+ . \{z\}\{z\}].$$

Like in the invariant logic, this specification encodes that the stack available is preserved through the procedure call. Note that these specifications do not mention the stack necessary to execute p and q themselves. This is because the accounting for those is done in the Q:Call rule, that is, in the caller.

Using these procedure specifications, to prove the triple $\vdash \{B\}\ \mathsf{call}\ p\ \{B, \bot\}$ we only have to apply the Q:Call rule with z set to $B - M(p)$. Indeed, regardless of the calling convention entry and exit, we always have $z \geqslant \max\{z \mid \mathsf{entry}_0\ p\ \sigma\ \sigma'\}$ and $\min\{z \mid \mathsf{exit}_0\ p\ \sigma'\ \sigma\} \geqslant z$. Note that these inequalities hold even when the sets are empty. The same triple is derived for $\mathsf{call}\ q$, and the two triples are composed with the sequence rule Q:Seq to get the final bound.

3.4 Certified Stack-Aware Compilation

In this section, I introduce our new technique for verifying *rewource-aware compiler correctness* and its implementation in Quantitative CompCert. The credit for this implementation work goes mostly to Tahina Ramananandro.

3.4.1 Verification Overview

Our development is highly influenced by the design of CompCert [Ler09], a verified compiler for the C language. CompCert C accepts most of the ISO-C-90 language and produces machine code for the x86 architecture (among others). CompCert uses 11 intermediate languages and 20 passes to compile a C AST to an x86 assembly AST.

The soundness proof of CompCert is based on trace-based operational semantics for the source, target, and intermediate languages. These semantics generate traces of events during the execution of programs. Events include input/output and external function calls. The soundness theorem of CompCert states that every event trace that can be generated by the compiled program can also be generated by the source program provided that the source program does not go wrong. In other words, the compiled program is a refinement of the source program with respect to the observable events.

In the following, I show how to extend trace-based compiler-correctness proofs to also cover stack-space consumption. In short, the technique works as follows.

1. We generate events for semantic actions that are relevant for stack-space usage, that is, function calls and returns.

2. We define a *weight* function for event traces that describes the stack-space consumption of program executions that produce that trace. The weight of an event trace is parameterized by a resource metric that describes the cost of each event.

3. We formally verify that for all resource metrics and for all event traces produced by a target program, the source program either goes wrong or produces an equivalent (see the following definition) event trace with a greater or equal weight.

4. During compilation, we produce a cost metric that accurately describes the memory consumption of target programs: If an execution of a target program produces an event trace of weight n under the produced metric then this execution can be performed on a system with stack size n.

I now formalize and elaborate on these points.

3.4.2 Event Traces

In CompCert, the observable events are external function calls (e.g., I/O events) that are represented by function identifiers together with a list of input values and an output value. In the following definition of these events, n is an integer literal and q is a floating point number. The intention is that the function identifier f specifies an external function such as printf, malloc, and free.

Event values	$v := \mathsf{int}(n) \mid \mathsf{float}(q)$
I/O events	$\nu := f(\vec{v} \mapsto v)$

To track stack usage, we add memory events for internal function calls and returns. Memory events do not have to be preserved during compilation. Here, f can be an arbitrary function identifer picked in the set $\mathcal{F}$.

$$\text{Memory events} \qquad \mu := \mathsf{call}(f) \mid \mathsf{return}(f)$$

Event traces are defined similar as in CompCert. We distinguish finite (inductive) traces t and possibly infinite (coinductive) traces T. A program behavior is either a converging computation $\mathsf{conv}(t, n)$ producing a finite event trace t and a return code n, a diverging computation $\mathsf{div}(T)$ producing a finite or infinite trace T, or a computation $\mathsf{fail}(t)$ that goes wrong and produces the finite trace t.

$$\begin{aligned}
\text{Events} &\qquad \iota : := \nu \mid \mu \\
\text{Finite event traces} &\qquad t := \epsilon \mid \iota \cdot t \\
\text{Coinductive event traces} &\qquad T := \epsilon \mid \iota \cdot T \\
\text{Behaviors} &\qquad B := \mathsf{conv}(t, n) \mid \mathsf{div}(T) \mid \mathsf{fail}(t)
\end{aligned}$$

I write $\mathcal{E}$ for the set of memory and I/O events, $\mathcal{B}$ for the set of behaviors, and $\mathcal{T}$ for the set of traces.

3.4.3 Weights of Behaviors

For a behavior B, we define the set of finite prefix traces *prefs*(B) of B as follows.

$$\begin{aligned}
\mathit{prefs}(\mathsf{conv}(t, n)) &:= \{t_1 \mid t = t_1 \cdot t_2\} \\
\mathit{prefs}(\mathsf{div}(T)) &:= \{t \mid T = t \cdot T'\} \\
\mathit{prefs}(\mathsf{fail}(t)) &:= \{t_1 \mid t = t_1 \cdot t_2\}
\end{aligned}$$

The *weight* $W_M(B) \in \mathbb{N} \cup \{\infty\}$ of a behavior B describes the number of bytes that are needed in an execution that produces B. It is parameterized by a stack metric $M \in (\mathcal{F} \to \mathbb{N})$ that maps function identifiers to natrual numbers (bytes). In the following, I will implicitely lift the metric M to work on events too. For an event $e \in \mathcal{E}$, $M(e)$ is defined by

$$M(\mathsf{call}(f)) := M(f) \qquad M(\mathsf{return}(f)) := -M(f) \qquad M(g(\vec{v} \mapsto v)) := 0.$$

The purpose of the metric is to relate memory events to the sizes of the stack frames of functions in the target code. In the Coq implementation of our compiler, we can also deal with nonzero stack consumption for external functions as long as the stack consumption of each call is bounded by a constant.

To define the weight of a trace, we first define the valuation $V_M(t)$ of a finite trace t. The valuation of a trace is essentially its net stack consumption; that is, the amount of stack in use at the end of the trace.

$$V_M(\epsilon) := 0 \qquad\qquad V_M(\alpha \cdot t) := M(\alpha) + V_M(t)$$

We now define the weight $W_M(B)$ of the behavior B under the metric M as follows.

$$W_M(B) := \sup\{V_M(t) \mid t \in \mathit{prefs}(B)\}$$

It is handy to use overloading to define the weight $W_M(T)$ of a (possibly infinite) trace T in the same way

$$W_M(T) := \sup\{V_M(t) \mid T = t \cdot T'\}$$

The following lemma follows directly from the definition of a valuation.

Lemma 5. *Let M be a metric and let t_1, t_2 be finite traces. Then $V_M(t_1 \cdot t_2) = V_M(t_1) + V_M(t_2)$.*

Lemma 6 shows how to decompose the weight of a trace into the weights and valuations of a prefix and a suffix of the trace. As usual, I use $n + \infty = \infty$.

Lemma 6. *Let M be a metric, t a finite trace, and T a possibly infinite trace. Then $W_M(t \cdot T) = \max\{W_M(t), V_M(t) + W_M(T)\}$.*

Proof. Since t is finite, we have $W_M(t) = \max\{V_M(t_1) \mid t = t_1 \cdot t_2\} \in \{V_M(t') \mid t \cdot T = t' \cdot T'\}$. Thus $W_M(t \cdot T) \geqslant W_M(t)$. $\square$

3.4.4 Examples

Consider the following trace t that is generated by a call to a recursive function f that does not call any other functions.

$$t = \mathsf{call}(f), \mathsf{call}(f), \mathsf{call}(f), \mathsf{call}(f), \mathsf{return}(f), \mathsf{return}(f), \mathsf{return}(f), \mathsf{return}(f)$$

Under a stack metric M the weight of t is $W_M(t) = 4{\cdot}M(f)$, but the valuation of t is $V_M(t) = 0$.

In the next example, I assume a function g that first calls a function h_1, then recursively calls g, and finally calls h_2. The following event trace t' is generated by a call to g.

$$\begin{aligned} t' = {} & \mathsf{call}(g), \mathsf{call}(h_1), \mathsf{return}(h_1), \mathsf{call}(g), \mathsf{call}(h_1), \mathsf{return}(h_1), \mathsf{call}(h_2), \\ & \mathsf{return}(h_2), \mathsf{return}(g), \mathsf{call}(h_2), \mathsf{return}(h_2), \mathsf{return}(g) \end{aligned}$$

Under a stack metric M the weight of the trace t' is $W_M(t') = 2 \cdot M(g) + \max\{M(h_1), M(h_2)\}$, but the valuation of t' is null again.

3.4.5 Quantitative Refinement

For the description of quantitative refinements I leave the definition of programs abstract. A program $s \in \mathcal{P}$ is simply an object that is associated, through a function $[\![\cdot]\!] : \mathcal{P} \to \mathcal{B}$, with a set of behaviors $[\![s]\!] \in \mathcal{B}$. An execution of a program can produce different traces, either due to non-determinism in the semantics or due to user inputs that are recorded in the event traces.

For a behavior B, define the pruned behavior as the behavior $\overline{B}$ that results from deleting all memory events ($\mathsf{call}(f)$ or $\mathsf{return}(f)$) from B. First pruned finite trances are defined inductively as follows. As always, ν denotes an I/O event and μ denotes a memory event.

$$\overline{\epsilon} := \epsilon \qquad \overline{\nu \cdot t} := \nu \cdot \overline{t} \qquad \overline{\mu \cdot t} := \overline{t}$$

Similarly, pruning for possibily infinite traces is defined as

$$\overline{\mu_1 \cdot \cdots \cdot \mu_n \cdot \epsilon} := \epsilon \qquad \overline{\mu_1 \cdot \cdots \cdot \mu_n \cdot \nu \cdot T} := \nu \cdot \overline{T} \qquad \overline{\mu_1 \cdot \cdots \cdot \mu_n \cdot \cdots} := \epsilon \;.$$

Finally, pruned behaviors are defined as

$$\overline{\mathsf{conv}(t,n)} := \mathsf{conv}(\bar{t},n) \qquad \overline{\mathsf{div}(T)} := \mathsf{div}(\overline{T}) \qquad \overline{\mathsf{fail}(t)} := \mathsf{fail}(\bar{t}) \ .$$

In CompCert, compiler correctness is formalized through the notion of *refinement*. A (target) program s' is a refinement of a (source) program s, written $s' \sqsubseteq s$, if for every behavior $B' \in [\![s']\!]$ there is $B \in [\![s]\!]$ such that $\overline{B} = \overline{B'}$ or $\mathsf{fail}(t) \in [\![P]\!]$ for a trace $t \in \mathit{prefs}(\overline{B})$. Note that memory events are not taken into account in CompCert's classic definition of refinement.

To also relate the memory events in the behaviors of two programs, we define a novel *quantitative refinement*. A (target) program s' is a quantitative refinement of a (source) program s, written $s' \sqsubseteq_Q s$ if the following holds. For every behavior $B' \in [\![s']\!]$ there exists $B \in [\![s]\!]$ such that $\overline{B} = \overline{B'}$ and $W_M(B') \leqslant W_M(B)$ for all stack metrics M, or $\mathsf{fail}(t) \in [\![P']\!]$ for a trace t with $\bar{t} \in \mathit{prefs}(\overline{B})$.

In Quantitative CompCert, our modified CompCert compiler, we prove for each compiler pass C that $C(s) \sqsubseteq_Q s$ for every program s.

3.4.6 Weights and Stack-Tracking Semantics

The way traces are generated by programs in the intermediate languages of CompCert is via an instrumentation of the semantics. In particular, Clight—the language used in our implementation—is defined using small-step semantics that define transitions $c \to_t c'$ where c and c' are configurations much like the ones of Section 2.2 and t is a finite list of events. The event trace of a (possibly infinite) execution is simply the concatenation of all the traces generated along it. Then, the stack consumption is defined using a weight function as explained in Section 3.4.3.

This treatment, convenient for the proofs of quantitative refinements, is relatively inconvenient for reasoning on the stack consumption of a program. Additionally, it differs from the resource-safety approach described in Section 3.2. The idea of the solution we implemented is to prove a theorem that relates our stack-tracking language L to CompCert's Clight. A configuration C of Clight is related to a family of configurations $(\sigma_C, \_, c_C, k_C)$ of our stack-tracking language L, and a theorem similar to the followin one is proved.

Theorem 11. *If C is a Clight configuration such that (σ_C, n, c_C, k_C) is a resource-safe configuration (in L), then for any trace t such that $C \to_t C'$ with Clight's semantics, we have $W_M(t) \leqslant n$.*

In the Coq implementation I use a slightly stronger theorem because it handles external calls and all the other events that Clight can generate. Using this theorem, we essentially leave the handling

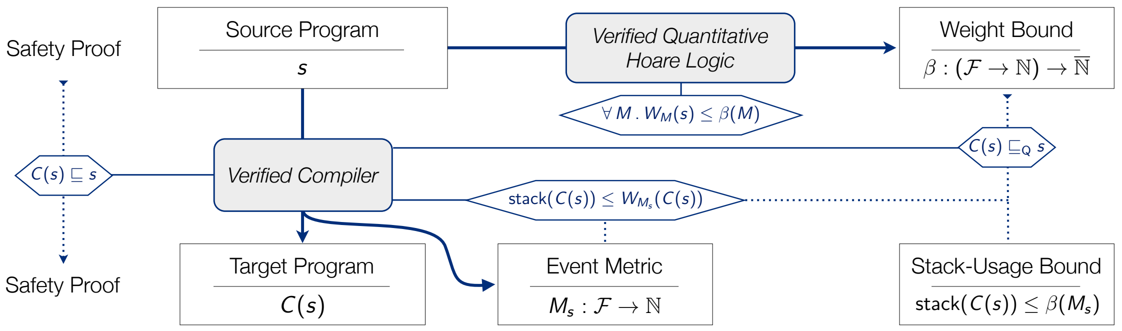
Safety Proof
Source Program
s
Verified Quantitative Hoare Logic
Weight Bound
$\beta : (\mathcal{F} \to \mathbb{N}) \to \overline{\mathbb{N}}$
$\forall M . W_M(s) \leq \beta(M)$
$C(s) \sqsubseteq_Q s$
$C(s) \sqsubseteq s$
Verified Compiler
$\mathsf{stack}(C(s)) \leq W_{M_s}(C(s))$
Safety Proof
Target Program
$C(s)$
Event Metric
$M_s : \mathcal{F} \to \mathbb{N}$
Stack-Usage Bound
$\mathsf{stack}(C(s)) \leq \beta(M_s)$

Figure 3.4: Overview of the stack-usage verification framework. We write $W_M(s) = \sup\{W_M(B) \mid B \in [\![s]\!]\}$ for the weight of the program s under the metric M. Furthermore, we write $\mathsf{stack}(s)$ for the smallest number n so that s runs without stack overflow if executed with a stack of size n. The notation $\overline{\mathbb{N}}$ is used to denote $\mathbb{N} \cup \{\infty\}$.

of traces to CompCert and prove stack-bounds using the resource-safety approach discussed in Section 3.2.

3.4.7 Verifying Stack Usage

Figure 3.4 shows how we verify the stack-space usage of a program in our framework. First, we prove a bound $\beta \in ((\mathcal{E} \to \mathbb{Z}) \to \mathbb{N} \cup \{\infty\})$ on the weights of the event traces that a program can produce. We do this using the quantitative logic and the theorem discussed in the previous section. The bound derived is parameterized by a stack metric $M \in (\mathcal{F} \to \mathbb{N})$. Second, our verified compiler—thanks to quantitative refinement—ensures that the computed bound also holds for the weights of the traces of the compiled program.

Third, we have to relate the computed bound to the actual stack usage of the compiled code. Therefore, our compiler computes not only a target program $C(s)$ but also a metric M_s such that $C(s)$ can be safely executed with a stack-memory size of $\sup\{W_{M_s}(B) \mid B \in [\![C(s)]\!]\}$ bytes. As a result, the initially derived bound for the source code can be instantiated with the metric M_s to obtain the wanted stack-space bound $M_s(\beta)$ for the target program.

In this overview picture, we assume that the semantics of the target and source languages are both formulated with an unbounded stack. The final step of the soundness proof (not illustrated in Figure 3.4) is to relate the trace-based semantics of the target language to a realistic assembly semantics in which the program is executed with a fixed stack size. To this end, we prove that an execution of $C(s)$ with bounded stack space $\sup\{W_{M_s}(B) \mid B \in [\![C(s)]\!]\}$ is a refinement of the execution of $C(s)$ in the semantics with unbounded stack.

3.5 Coq Implementation

In this section we give a brief description of our Coq implementation of the quantitative logic and its metatheory. We also quickly describe a simple automatic procedure to derive stack bounds automatically for non-recursive programs. Coupled with Quantitative CompCert and the soundness proof of the quantitative logic, this automatic procedure allows us to build a stack-aware certified compiler that automatically infers an upper bounds on the minimum stack size required to run the compiled program.

3.5.1 Logics and Quantitative CompCert

Quantitative logic. The Theorems 6, 7, and 8 relating the quantitative and the invariant logics were proved in Coq. I found that they are quite delicate results and that using a proof assistant was of great use to spot many bugs in the initial presentation of the quantitative logic.

In the development, assertions of the invariant logic are defined as follows.

```
Definition assn: Type := (memory × Z) → Prop.
```

This is the simplest way to represent a set using the Coq proof assistant. Using `Prop` instead of `bool` lets us leverage built-in tactics to automate the reasoning. The `Z` component of the state is the stack counter, while the `memory` component is the normal program memory. The quantitative assertions are defined as follows.

```
Definition qassn: Type := memory → option Z.
```

I used `option Z` to represent the set $\mathbb{Z} \cup \{\infty\}$. Note that I used $\mathbb{Z}$ instead of $\mathbb{N}$, this saves a lot of coercions and other technical annoyances because the stack counter itself is of type `Z`—indeed, it has to be able to become negative to detect stack overflows. In practice this required me to add non-negativity guards in many places in the quantitative logic. They are implicit in the version presented in this dissertation where I chose to use $\mathbb{N} \cup \{\infty\}$ to have a lighter presentation.

In various places in the quantitative logic, I use max and min operators on sets. If sets are represented as maps to `Prop`, such operators are not constructive, and hence could not be expressed in Coq. I side-stepped this problem by axiomatizing the two operators, with the incovenient that some proofs become non-constructive.

Quantitative CompCert. The adaptations required to make the proof of the CompCert compiler stack aware was non-trivial. To explain them, I first informally describe the CompCert memory model [LB08]. In CompCert, memory is not one contiguous array of bytes, but a sequence of finite arrays of bytes, called *memory blocks*. The address of a memory location is of the form (b, o) where b is the sequence number of the memory block, and o is a machine integer representing the offset of the byte within this block. The most important thing to know about this memory model is that memory blocks are independent of each other: pointer arithmetics can be done only within a given block, so that, for instance, shifting an address (b, o) by some offset δ yields $(b, o + \delta)$: CompCert guarantees that such pointer arithmetics will never cross block boundaries.

In Clight, CompCert allocates one block for each addressable local variable. By contrast, Cminor allocates only one block per function call, into which all those local variables are merged by a set of compilation passes. This single memory block per function call actually corresponds to one stack frame, whose size stays the same from Cminor down to Linear, and gets increased only in Mach, where it also receives the spilling locations and function arguments, without allocating any new blocks for these additional data.

In the original CompCert x86 assembly language, the notion of stack frame is still kept, so that this language has two *pseudo-instructions* Pallocframe and Pfreeframe responsible of allocating and freeing the corresponding memory block. Those pseudo-instructions are then turned into real x86 assembly instructions performing additions and substractions on stack pointer register. This latter transformation cannot be proved correct in CompCert, because pointer arithmetics cannot cross block boundaries in the CompCert memory model. It is therefore done in an unverified "pretty-printing" stage, after CompCert has generated the x86 assembly code of the source program.

Our new assembly semantics overcomes this limitation. Now, instead of allocating different memory blocks, we preallocate one single fixed-size block at the beginning of the program to hold the whole stack, and our assembly generation pass ensures that the value of the stack pointer always points within this block. Therefore the pseudo-instructions are no longer necessary, and the pointer arithmetics needed at function entry and exit can be performed within our formalized stack-aware assembly language.

3.5.2 A Simple Proof-Producing Automation

In larger C programs a manual interactive verification with a program logic is too tedious and time-consuming to be practical. Therefore I have developed an automatic stack analysis tool that operates at the Clight level to enable the analysis of real system code. This automatic tool is the first proof of concept that demonstrates the value of the logic for formal verification of static analysis tools. Later in this dissertation, I will present more complex analyses that also leverage quantitative logics as a formal "backend."

The basic idea of the automatic stack analyzer is to compute a call graph from the Clight code and to derive a stack bound for each function in topological order. In Coq, the derivation of a function bound is implemented by a recursive function auto_bound on the abstract syntax tree (AST) of a Clight program. The function auto_bound does not only compute a stack bound but also a derivation in the quantitative logic. This ensures the correctness of the generated bound and enables the composition of stack bounds that have been derived interactively or with other static analysis tools. In addition to the AST, auto_bound takes a context of known function bounds together with their derivations in the logic as an argument.

Using the verified quantitative logic, the implementation of auto_bound is straightforward. For trivial commands like assignments or skip, auto_bound simply generates the bound 0 and a derivation like $\vdash \{0\}$ skip $\{0, 0\}$. For a sequential composition $c_1; c_2$ it inductively applies auto_bound to c_1 and c_2, and derives the bounds $\vdash \{P_i\}\ c_i\ \{Q_i, B_i\}$ for $i{=}1, 2$. Then, auto_bound returns the precondition $\{\max(P_1, P_2)\}$ and the postcondition $\{\max(Q_1, Q_2), \max(B_1, B_2)\}$ for $c_1; c_2$. The derivation of this bound is similar to the example derivation sketched in Section 3.3.4. The computation of the bound for the conditional works similarly. For loops we can use the bound derived for the loop body to obtain a bound for the loop; indeed, because of the stack discipline the stack needs for a loop is at most the stack-usage of its body (at most, because the loop might never be entered). In the derivation we just apply the rule Q:Loop. Function calls are handled with the context of known function bounds (recursion is not allowed here) and the rule Q:Call.

For a given C program, auto_bound is applied to every function definition in the well-founded topological order provided by the call graph. Inductively, we use the resulting bounds to generate the context of known function bounds for the following calls of auto_bound.

We have combined our automatic stack analyzer with the Quantitative CompCert compiler. The result is a verified C compiler that translates a program without function pointers and recursive calls to x86 assembly and automatically derives a stack bound for each function in the program, including main. The soundness theorem we have proved states the following. If a given program is memory-safe and the verified compiler successfully produces an assembly program A then A refines the source program and runs safely on an x86 machine with the stack size that has been

Function Name	Verified Stack Bound
`recid()`	$8a$ bytes
`bsearch(x,lo,hi)`	$40(1 + \log_2(hi - lo))$ bytes
`fib(n)`	$24n$ bytes
`qsort(a,lo,hi)`	$48(hi - lo)$ bytes
`filter_pos(a,sz,lo,hi)`	$48(hi - lo)$ bytes
`sum(a,lo,hi)`	$32(hi - lo)$ bytes
`fact_sq(n)`	$40 + 24n^2$ bytes
`filter_find(a,sz,lo,hi)`	$128 + 48(hi - lo) + 40 \log_2(BL)$ bytes

Table 3.1: Manually verified stack bounds for C functions.

computed by the automatic stack analysis for `main`. During compilation, the stack-aware CompCert compiler prints the computed stack bound for every function and the overall stack requirement for the program.

3.6 Experimental Evaluation

To validate the practicality of the framework for stack-bound verification, I have performed an evaluation with more than 3000 lines of C code from different sources. The C programs used for the evaluation include handwritten code, programs from the CompCert test suite, programs from the MiBench [GRE+01] embedded software benchmarks, and modules from the simplified development version of the CertiKOS operating system kernel [GKR+15]. The size of the analyzed example files varies from a recursive Fibonacci function of 8 lines of code to a CertKOS module of 819 lines of code.

Testing the automatic analysis. The programs used to test our automatic analysis came from three different sources. The main test case for the automatic stack-analyzer is the CertiKOS operating system kernel [GKR+15]. Since CertiKOS does not make use of recursion, the automatic analysis can be used to derive precise stack bounds. Using our quantitative compiler, we were, for instance, able to compile and compute bounds for both the virtual memory management module and the process management modules of the kernel. Second, the MiBench [GRE+01] benchmark

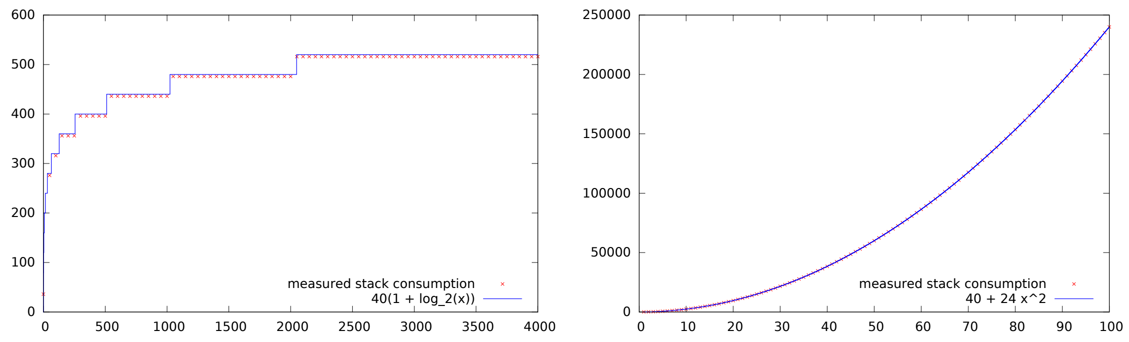

Figure 3.5: Hand-derived (blue line) v.s. measured (red crosses) stack usage. The left plot is for the `bsearch` program and the right one is for `fact_sq`.

suite was used to test the automation on safety critical software. Finally, because our compiler is based on CompCert, we found it natural to use the CompCert test suite to test the quantitative compiler and the automated analysis. Files with automatically derived bounds for non-recursive functions from the CompCert test suite include `mandelbrot.c` which computes an approximation of the Mandelbrot set and `nbody.c` which computes an n-body simulation of a part of our solar system. In total roughly 3000 lines of system C code were processed without any human interaction.

Testing manual bound inference. Table 3.1 (p. 58) contains some recursive functions that were analyzed manually using the quantitative logic. The function `bsearch` is a recursive binary search with logarithmic recursion depth. The function `fib`, from the CompCert test suite, computes the Fibonacci sequence using an exponential algorithm. The function `qsort` is also from the CompCert test suite and implements a recursive version of the quicksort algorithm. In both cases the asymptotically tight linear bounds were proved. The verification of the function `fact_sq` shows the compositionality of the logic: I first verify a linear bound for the factorial function and then use this bound to verify `fact_sq(n)`, which contains the call `fact(n*n)`. The function `filter_pos` takes an array and computes a new array that contains all positive elements of the input array. Similarly, `filter_find` uses the binary search `bsearch` to filter out all elements of an input array that are contained in another array of size `BL`.

Accuracy of the bounds derived. I have evaluated the precision of the derived bounds by comparing them with the actual stack-space consumption during the execution of the compiled C programs. The experiments show that the derived bounds are very precise: both the manually and automatically derived bounds differs from the measured stack usage by exactly four bytes (because of the unaccounted space for the return address of `main`).

Figure 3.5 (p. 59) shows the results of two experiments I made with hand-derived stack bounds using the quantitative logic. The plots show the derived bounds for the functions `bsearch` and `fact_sq` (blue lines) and the measured stack usage for different inputs (red crosses). The x-axis shows the size of the input; either the value of the integer argument (for `fact_sq`) or the length of the input array (for `bsearch`). The y-axis shows the stack usage in bytes.

I also evaluated the automatic tool on complete programs. This includes part of the CompCert benchmarks and some programs from the MiBench benchmark suite. The derived bounds are all off by exactly four bytes. Unfortunately, the precision of bounds derived on the CertiKOS operating system kernel could not be experimentally verified since it cannot be executed as a regular program.

The actual stack usage of programs was measured using a tool I implemented for that purpose. It uses the Linux system call `ptrace` to single-step the monitored process and keeps track of its maximum stack-space usage.

Chapter 4

The Interval System

In this chapter, I introduce a technique to automate the inference of resource bounds. By design, the soundness of this automation is a direct consequence of the soundness of quantitative logics. Remember that quantitative logics use resource bounds where typical logics use assertions. The first key to automation is to fix the shape of these resource bounds to match a well-chosen template parameterized by rational coefficients. The second key to automation is to remark that constraints between resource bounds in this template form can be enforced with linear constraints on the coefficients. Because the constraints are linear, an off-the-shelf linear-programming solver can solve them, and automate the inference of resource bounds.

4.1 Language Definition

The language we are going to use in this chapter is again an instance of the abstract structured language of Chapter 2. In this language, the resource of interest will be the number of "ticks" inserted by the programmer or by a preprocessing transformation.

Ticks. One way to handle arbitrary resources is to attach a custom cost to each step that the operational semantics can take. The usual name of the tuple of all these costs is a *cost metric*. This presentation has practical benefits: for example, it gives a good framework to infer precise bounds on the running time and memory usage of programs [DH17]. On the other hand, it tangles the usual semantics with the resource behavior of programs. These two concepts are orthogonal, and I prefer to present them in an orthogonal fashion. To this end, I introduce a $\mathsf{tick}\ k$ statement that subtracts the integer k from the resource counter. If k is positive, resource is consumed; if k is negative, resource is released. Note that `tick` must take a constant as argument, not an expression.

Language definition. The language I am going to use in the following development is an instance of the generic language with procedure calls from Section 2.6. To describe the automation procedure in a concrete setting, we will be using concrete base statements and define memory states that support global and local variables. The set of global variables is fixed beforehand to be $\mathcal{G} \subseteq \mathcal{V}$.

Memory states $m \in \Sigma_0 := \mathcal{V} \to \mathbb{Z}$ map variables names to integers. Because we wish to support local variables in procedures, *program states* in Σ must contain a *memory stack* which is used to restore the variables of the caller when returning from a call. Because we also want to keep track of the resource consumption of the program, states in Σ also have to embed a resource counter in $\mathbb{Z}$. Therefore, we define

$$\Sigma = [\Sigma_0]_1 \times \mathbb{Z}.$$

The notation $[\Sigma_0]_1$ is used to denote the set of all memory stacks: lists of memory states that contain at least one element. I will use the standard list notation $m{::}\mu$ to denote the list resulting from appending the element m to the list μ.

As a convention, any variable used in a procedure and not in the set $\mathcal{G}$ of global variables is considered local (preserved across calls). To define the calling convention relations entry and exit, I will use the binary operator $\cup_{\mathcal{G}}$ on memory states that is defined by

$$m \cup_{\mathcal{G}} m' := \begin{cases} m'(v) & \text{if } v \in \mathcal{G} \\ m(v) & \text{otherwise} \end{cases}.$$

It fuses two memory states taking the values of global variables from its right-hand operand and the values of local variables from its left-hand operand. When entering a procedure, we keep the value of global variables and initialize the local variables of the called procedure to 0. The topmost memory state on the stack can thus be taken to be $0 \cup_{\mathcal{G}} m$, where m was the old topmost memory state and 0 is the memory state that associates 0 to all variables. Dually, the exit predicate simply pops the topmost state off the stack and fetches the value of the global variables out of it. The precise definitions of the two relations is given below.

$$\mathsf{entry}\ p\ \sigma_0\ \sigma_1 \iff \sigma_0 = (m{::}\mu, n) \wedge \sigma_1 = ((0 \cup_{\mathcal{G}} m){::}m{::}\mu, n)$$

$$\mathsf{exit}\ p\ \sigma_0\ \sigma_1 \iff \sigma_0 = (m_p{::}m{::}\mu, n) \wedge \sigma_1 = ((m \cup_{\mathcal{G}} m_p){::}\mu, n)$$

Note that contrary to the semantics we used to track the stack usage in Section 3.3.1, the resource counter is not changed by entering and leaving procedures.

The base statements available are:

$$\begin{aligned} &\text{(Base statements)} && b := v \leftarrow v + \theta \mid v \leftarrow \theta \mid v \leftarrow \star \mid \mathsf{tick}\ k \mid \mathsf{guard}\ G \\ &\text{(Operands)} && \theta := k \mid v' \end{aligned}$$

Where $v, v' \in \mathcal{V}$, $k \in \mathbb{Z}$, and G is a set of memory states. Base statements can be—in the order they are listed—a variable increment (either by a constant or by another variable), a deterministic variable assignment, a non-deterministic variable assignment, a tick instruction (as explained in the previous paragraph), or a guard.

The semantics of each base statement is naturally defined by

$$\begin{aligned} (m{::}\mu, n)[\![v \leftarrow v + \theta]\!](m'{::}\mu', n') &\iff m' = m[[\![v + \theta]\!]_m / v], \\ (m{::}\mu, n)[\![v \leftarrow \theta]\!](m'{::}\mu', n') &\iff m' = m[[\![\theta]\!]_m / v], \\ (m{::}\mu, n)[\![v \leftarrow \star]\!](m'{::}\mu', n') &\iff \forall v' \neq v,\ m'(v') = m(v'), \\ (m{::}\mu, n)[\![\mathsf{tick}\ k]\!](m'{::}\mu', n') &\iff n' = n - k, \\ (m{::}\mu, n)[\![\mathsf{guard}\ G]\!](m'{::}\mu', n') &\iff m \in G. \end{aligned}$$

Each relation above changes at most one component of the full state $(m{::}\mu, n)$, when a component and its primed version are not explicitely related, they are implicitely constrained to be the same. I used the notation $[\![\theta]\!]_m$ to denote $m(v')$ when $\theta = v'$, and k when $\theta = k$. And similarly, $[\![v + \theta]\!]_m := m(v) + [\![\theta]\!]_m$. I also used the standard notation $m[x/v]$ to denote the map that results from updating the binding of v in m to x. Note that the only way to change the resource counter n is to use the tick statement, and that all the base statements change only the topmost memory state on the stack or the resource counter.

4.2 A Quantitative Logic for Tick Usage

In Section 3.3.1, I presented a logic to infer bounds on the stack consumption of a program. In this section, I adapt the stack logic to infer bounds on the tick usage of programs. I will use this quantitative logic as a backend for the automated analysis presented later in this chapter.

4.2.1 Inference Rules

The tick quantitative logic, similarly to the one for stack-space usage of Section 3.3.1, uses functions instead of classic assertions. These functions are *potential functions* that map memory stacks in $[\Sigma_0]_1$ to non-negative rational numbers. For the most part, the rules of the tick quantitative logic are similar to the ones of the stack logic. The complete set of rules is presented in Figure 4.1 (p. 65). The rule for the base statement got specialized to the base statements described in the previous paragraph as G:Assign, G:NonDet, G:Guard, and G:Tick.

The G:Call rule provides a new *framing mechanism* that we have not encountered yet. I am going to use this mechanism later in this chapter to prove the soundness of the interval system. In the rule, the *local frame* L is a function that maps memory stacks to non-negative rational numbers. It is required to depend only on local variables (and possibly on the contents of the memory states higher in the memory stack). The frame is used to carry potential associated with local variables across the procedure call.

4.2.2 Metatheory

Because the primary motivation for this logic is to prove the soundness of the automated interval system, I will simply prove the soundness of the logic. Following the development in Section 3.3.2, I will show a translation that turns a derivation of the tick quantitative logic into a derivation of the invariant logic with the resource-safety invariant $I := \{(m{::}\mu, n) \mid n \geqslant 0\}$.

Because of the new framing mechanism of the G:Call rule, the translation turning a potential function P into the assertion $\overline{P} := \{(m{::}\mu, n) \mid n \geqslant P(m{::}\mu)\}$ cannot be used as is. Indeed, when trying to prove that the translated triple is derivable in the invariant logic, the case of the G:Call rule does not go through. (Even though all the others do!) The key to this proof is to instead consider the translation of a triple $\vdash \{P\}\ c\ \{Q, B\}$ (and all the assertions composing it) in a given frame $X \in ([\Sigma_0] \to \mathbb{Q}_0^+)$. The translation of a potential function P is taken to be $\overline{P + X} := \{(m{::}\mu, n) \mid n \geqslant P(m{::}\mu) + X(\mu)\}$. And then, a function specification $[\forall z \in Z.\, \{P_z\}\{Q_z\}]$ is translated to

$$[\forall (z, X) \in Z \times ([\Sigma_0] \to \mathbb{Q}_0^+).\, \{\overline{P_z + X}\}\{\overline{P_z + X}\}]. \tag{4.1}$$

The essential detail making this translation suitable to the soundness proof is that each procedure specification can be used in an arbitrary frame X.

This modified translation scheme enables to prove that, if a triple $T := \vdash \{P\}\ c\ \{Q, B\}$ is derivable

$\boxed{\vdash \Gamma}$

$$\vdash \Gamma \quad := \quad [\forall z \in Z.\, \{P_z\}\{Q_z\}] \in \Gamma(p) \quad \Longrightarrow \quad \forall z \in Z. \quad \Gamma \vdash \{P_z\}\ \Delta(p)\ \{Q_z, \top\}$$

$\boxed{\Gamma \vdash \{P\}\ c\ \{Q, B\}}$

$$\frac{e = v + \theta \text{ or } e = \theta}{\vdash \{(m{::}\mu) \mapsto Q(m[\llbracket e \rrbracket_m / v]{::}\mu)\}\ v \leftarrow e\ \{Q, \bot\}}\ (\text{G:Assign})$$

$$\frac{}{\vdash \{(m{::}\mu) \mapsto \max\{Q(m[k/v]{::}\mu) \mid k \in \mathbb{Z}\}\}\ v \leftarrow \star\ \{Q, \bot\}}\ (\text{G:NonDet})$$

$$\frac{}{\vdash \{(m{::}\mu) \mapsto \mathsf{if}\ m \in G\ \mathsf{then}\ Q(m{::}\mu)\ \mathsf{else}\ 0\}\ \mathsf{guard}\ G\ \{Q, \bot\}}\ (\text{G:Guard})$$

$$\frac{}{\vdash \{Q\}\ \mathsf{skip}\ \{Q, \bot\}}\ (\text{G:Skip}) \qquad \frac{}{\vdash \{Q + k\}\ \mathsf{tick}\ k\ \{Q, \bot\}}\ (\text{G:Tick})$$

$$\frac{}{\vdash \{Q\}\ \mathsf{break}\ \{\bot, Q\}}\ (\text{G:Break}) \qquad \frac{\vdash \{P\}\ c_1\ \{Q, B\} \qquad \vdash \{Q\}\ c_2\ \{R, B\}}{\vdash \{P\}\ c_1; c_2\ \{R, B\}}\ (\text{G:Seq})$$

$$\frac{\vdash \{P\}\ c_1\ \{Q, B\} \qquad \vdash \{P\}\ c_2\ \{Q, B\}}{\vdash \{P\}\ c_1{+}c_2\ \{Q, B\}}\ (\text{G:Alt}) \qquad \frac{\vdash \{P\}\ c\ \{P, B\}}{\vdash \{P\}\ \mathsf{loop}\ c\ \{B, \bot\}}\ (\text{G:Loop})$$

$$\frac{\begin{array}{c} [\forall z \in Z.\, \{P_z\}\{Q_z\}] \in \Gamma(p) \qquad z \in Z \\ L \in ([\Sigma_0]_1 \to \mathbb{Q}_0^+) \qquad \forall m, m', \mu.\, L(m{::}\mu) = L((m \cup_{\mathcal{G}} m'){::}\mu) \qquad \text{(i.e., } L \text{ only depends on locals)} \\ P(m{::}\mu) = P_z((0 \cup_{\mathcal{G}} m){::}m{::}\mu) \qquad Q(m{::}\mu) = \min\{Q_z(m'{::}m''{::}\mu) \mid m'' \cup_{\mathcal{G}} m' = m\} \end{array}}{\vdash \{P + L\}\ \mathsf{call}\ p\ \{Q + L, \bot\}}\ (\text{G:Call})$$

$$\frac{P' \geqslant P \qquad \vdash \{P\}\ c\ \{Q, B\} \qquad Q \geqslant Q' \qquad B \geqslant B'}{\vdash \{P'\}\ c\ \{Q', B'\}}\ (\text{G:Conseq})$$

Figure 4.1: Tick quantitative logic derivation rules in a context Γ (elided).

in the tick quantitative logic, then the triple $\vdash_I \{\overline{P + X}\}\ c\ \{\overline{Q + X}, \overline{B + X}\}$ is derivable in the invariant logic. The proof goes by induction on the derivation of the triple T. Most cases are similar to their equivalent in the soundness proof of the stack quantitative logic. The G:Call case is the only one that differs and makes use of the translation of procedure specifications showed in Equation (4.1). The core of the argument in that case is that, if the translation is made with the frame X, the frame used with the specification of the called procedure is essentially $X'(\mu) := X(\mathsf{tl}(\mu)) + L(\mu)$, where L is the frame used in the application of the G:Call rule, and tl is the function returning the memory stack resulting from dropping the topmost memory state off its argument. The fact that L depends only on local variables is used when proving that the postcondition of the translated triple is correct. All the details of this translation and the proof of the following soundness theorem can be read in the accompanying Coq development.

Theorem 12 (Soundess of the tick quantitative logic)**.** *If* $\vdash \Gamma$ *and* $\Gamma \vdash \{P\}\ c\ \{Q, \top\}$*, and the memory stack* $m{::}\mu$ *and integer* n *are such that* $n \geqslant P(m{::}\mu)$*, then any* n' *such that* $(m{::}\mu, n, c, \mathit{kstop}) \to^* (\_, n', \_, \_)$ *satisfies* $n' \geqslant 0$*.*

Proof sketch. Use the translation described above with the constant frame $X := (\mu \mapsto 0)$, then apply the soundness result of the invariant logic to conclude that any state reachable is resource safe, and thus, n' must be non-negative. □

4.2.3 Digression: Cost-Free Judgements and Framing

The framing mechanism across calls described in the previous section is only one specific instance of framing. In this section, I will informally describe a more general framing principle based on cost-free programs—or equivalently, cost-free derivations. This more general framing principle is useful to prove the soundness of more complicated automated system. We will encounter it again in Chapter 5, this time in a more formal setting.

Cost-free programs and size changes. A cost-free program is a program where all tick statements have 0 as argument. Obviously, such a program does not consume nor releases any resource. Any program can be turned into a cost-free program by simply changing the argument of all its tick statements to 0. Cost-free programs are trivial from a resource-analysis point-of-view, however, we will see that they give a new reading of the triples of the quantitative logic.

Since a command c of a cost-free program has no resource consumption, the triple $\vdash \{0\}\ c\ \{0, 0\}$ is derivable; this triple says exactly that c has no resource cost. Unsurprisingly, the resource analysis of cost-free programs is not very interesting. Nonetheless, the quantitative logic is able to derive non-trival facts about cost-free programs: it can infer size-change information. A correct reading of the triple $T := \vdash \{P\}\ c\ \{Q, \top\}$ is that if the resource counter is initially larger than the precondition potential $P(m{::}\mu)$, then the command c does not run out of resource, *and in addition*, at least $Q(m'{::}\mu')$ resource units are available at the end of the execution. Theorem 12 does not provide this reading, but it is readily justified using the translation of the previous section and the Hoare-style soundness result of the invariant logic. When the command c is cost-free, the resource counter n at the beginning of the execution is never changed, so the final resource counter is also n, and by the new reading of the triple T, we have

$$\forall n, n \geqslant P(m{::}\mu) \implies n \geqslant Q(m'{::}\mu').$$

This formula is equivalent to the simpler $P(m{::}\mu) \geqslant Q(m'{::}\mu')$ (in one direction, apply the formula to $n = P(m{::}\mu)$; in the other, use the transitivity of $\geqslant$). Thus, a triple relates the initial and final memory stacks using the initial and final potential. For example, if c is cost-free and we can derive the triple $\vdash \{g_1 + g_2\}\ c\ \{g_1, \top\}$, we proved that the value of the variable g_1 in the final state is no more than the sum of the values of g_1 and g_2 in the initial state: *we bounded the size change of* g_1. In summary, a quantitative logic is not only a mean to deriving resource bounds: it can also derive cross-program invariant such as size-change information—which is, reciprocally, useful to derive resource bounds! This is a powerful feature of quantitative logics.

Framing. The framing mechanism present in the G:Call rule allows to transfer potential associated to local variables across a procedure call. The soundness of this rule relies critically on the following two informal observations.

1. The local variables are unchanged through a call, so the value of the frame function $L(m)$ before the call is identical to the value of the frame $L(m')$ after the call.

2. A constant $X \geqslant 0$ can always be added to each potential function in a triple $\vdash \{P\}\ c\ \{Q, B\}$ to yield $\vdash \{P + X\}\ c\ \{Q + X, B + X\}$. The reason for this is that additional resource can always be framed to any resource-safe execution and will be retrieved at the end of it.

The first requirement can in fact be relaxed: if instead of $L(m) = L(m')$, we have $L(m) \geqslant L(m')$, L can also be soundly framed. If the inequality is strict, some potential was simply lost along the way. The constraint $L(m) \geqslant L(m')$ is a size-change property. As it so happens, the previous paragraph precisely explains that the quantitative logic is able to prove size-change properties when applied to cost-free programs.

To infer size-change properties on programs that are not cost-free, one can simply do as if all the tick statements had 0 as argument. To that end, we use a new judgement $\vdash_{\mathsf{cf}} \{P\}\ c\ \{Q, B\}$ (and $\vdash_{\mathsf{cf}} \Gamma$ for contexts). This judgement is defined by the exact same rules we presented in Figure 4.1 (p. 65) except for the G:Tick rule that now becomes

$$\frac{}{\vdash_{\mathsf{cf}} \{Q\}\ \mathsf{tick}\ k\ \{Q, \bot\}}\ (\text{CF:Tick}),$$

essentially treating the tick statement as if its argument were 0. When the cost-free judgement $\vdash_{\mathsf{cf}} \{P\}\ c\ \{Q, B\}$ can be derived, the precondition evaluated on the initial state $P(m)$ bounds the postcondition evaluated on the state at the end of the execution of c, $Q(m')$. Chapter 5 elaborates more on this idea and explains how to create a complete size-tracking analysis.

Using cost-free judgements, we can define a new frame rule which generalizes all the framing arguments we have seen so far:

$$\boxed{\frac{\vdash \{P\}\ c\ \{Q, B\} \qquad \vdash_{\mathsf{cf}} \{P'\}\ c\ \{Q', B'\}}{\vdash \{P + P'\}\ c\ \{Q + Q', B + B'\}}\ (\text{G:Frame}).}$$

The G:Frame rule is an admissible rule; that is, assuming the derivability of the two premisses, the derivability of the conclusion is ensured. I will only sketch a proof here.

1. First, if P makes the command c resource-safe, then $P + P'$ also does because it provides more resource to execute with. (Remember that potential functions are always non-negative in this chapter.)

2. Second, assume an execution $\epsilon := (m{::}\mu, n, c, \mathsf{kstop}) \to^* (m'{::}\mu', n', \mathsf{skip}, \mathsf{kstop})$, we show that $n' \geqslant (Q + Q')(m'{::}\mu')$ assuming that $n \geqslant (P + P')(m{::}\mu)$—this justifies that the fallthrough postcondition is valid. Let us define another execution ϵ' where all the resource counters have $P'(m{::}\mu)$ subtracted. The initial counter is then $n - P'(m{::}\mu) \geqslant P(m{::}\mu)$, thus by soundness of the first premiss, we have that $n' - P'(m{::}\mu) \geqslant Q(m'{::}\mu')$ (i). Using the fact that cost-free derivations prove size-change results, we have $P'(m{::}\mu) \geqslant Q'(m'{::}\mu')$ (ii). Combining equations (i) and (ii), we conclude $n' \geqslant Q(m'{::}\mu') + P'(m{::}\mu) \geqslant Q(m'{::}\mu') + Q'(m'{::}\mu') = (Q + Q')(m'{::}\mu')$.

3. A similar argument justifies that the break postcondition is also valid.

The three items above form the semantic meaning of the triple $T := \vdash \{P + P'\}\ c\ \{Q + Q', B + B'\}$. Invoking completeness of the quantitative logic (which I did not prove here), one could conclude the derivability of the triple T.

The reasoning we just did is valid to justify the admissibility of the G:Frame rule when there are no procedure calls in the command c. Otherwise, the context Γ implicitely used in the triple $\vdash \{P\}\ c\ \{Q, B\}$ causes technical difficulties. In that case, the admissiblilty of the rule can be proved using a translation technique to the invariant logic similar to what we saw in Section 4.2.1.

Concluding remarks on framing. Framing information through calls is tremendously useful in any system that reasons on programs with procedure calls. The framing I presented in this section combines one reasoning about the resource behavior of a command ($\vdash$) and one reasoning about the size change of some program data ($\vdash_{\mathsf{cf}}$) into a single triple. I believe this specific instance of framing is one of the great strengths of potential-based techniques, as well as a feature making the quantitative logic framework very versatile: an analysis based on this logic does not run in two different phases (size change and resource analysis), but combines the two reasonings elegantly.

The argumentation in this section was intentionally a bird's-eye view of the topic, but it presented what I think is the essence of the issue. The cost-free annotations and the "additivity" of potential annoatation as embodied in the rule G:Frame are recurrent in potential-based systems. In RAML, for instance, they are used to implement the so-called *resource-polymorphic recursion.*

4.3 Automated Inference System with Intervals

The quantitative logic presented in the previous section provides a sound mean to establish resource bounds. In this section, we show a simple way to automatically find derivations in the quantitative logic and prove linear bounds. Surprisingly, even though the interval system is basic, it can infer bounds for a wide range of programs with linear complexity. I will present multiple challenging examples from previous works and some original ones, and explain how the bounds are discovered. The interval system was implemented in the tool C4B.

4.3.1 Potential Functions

To find resource bounds automatically, we first need to restrict the search space. Potential functions in the automation are split in two parts: a logical context and an interval potential function. The logical context is used to encode the invariants that hold at a given program point; it is inferred using well-known abstract-interpretation techniques. The interval potential functions form the core of the automation, they are linear combinations of smaller base interval functions. Their inference is automated using linear programming.

Interval potential functions. An *interval potential function* maps a memory state to a non-negative rational number, it is restricted to be of the following shape

$$\Phi(m) = q_0 + \sum_{\substack{x,y \in \mathcal{V} \\ x \neq y}} q_{(x,y)} \cdot |[m(x), m(y)]| \,.$$

Here, we used the interval notation $|[a, b]|$ to mean $\max(0, b - a)$, and $q_i \in \mathbb{Q}_0^+$. Note that an interval potential function takes a memory state $m \in \Sigma_0$ as argument, not a complete memory stack $m{::}\mu \in [\Sigma_0]_1$. The convention I will take from now on is that a function acting on memory states also acts on memory stacks as $\Phi(m{::}\mu) := \Phi(m)$. Similarly, a set of memory states S can be seen as the set of memory stacks that have their topmost memory state in S.

To simplify the references to the linear coefficients q_i, we introduce an *index set* I. This set is defined to be $\{0\} \cup \{xy \mid x, y \in \mathcal{V} \wedge x \neq y\}$. Each index i corresponds to a *base function* f_i in the potential function: 0 corresponds to the constant function $m \mapsto 1$, and xy corresponds to $m \mapsto |[m(x), m(y)]|$. Using these notations we can rewrite the above equality as

$$\Phi(m) = \sum_{i \in I} q_i \cdot f_i(m).$$

The family $(f_i)_{i \in I}$ is a generating family (in the linear-algebra sense) for interval potential functions. The index set allows us to represent any interval potential function Φ as a *quantitative annotation* $Q = (q_i)_{i \in I}$, that is, a family of rational numbers where only a finite number of elements are not zero. In the following, we will use the annotation Q in place of the interval potential Φ it denotes.

To simplify the presentation in the rest of this chapter, I will treat constants as global variables that cannot be assigned to. For example, if the program contains the constant 8128 then there is a variable c_{8128} and $m(c_{8128}) = 8128$ for any memory state m.

Logical contexts. In addition to the quantitative annotations, the automatic amortized analysis needs to maintain a minimal abstract state to justify certain operations on quantitative annotations. For example when analyzing the command $x \leftarrow x + y$, it is helpful to know the sign of y to determine which intervals in a potential function will increase or decrease.

In the interval system, an abstract program state is represented as a *logical context* P. The implementation finds these logical contexts using abstract interpretation with the domain of linear inequalities. We observed that the rules of the analysis often require only minimal local knowledge. This means that it is not necessary for us to compute precise loop invariants and only a rough fixpoint (e.g., keeping only inequalities on variables unchanged by the loop) is sufficient to obtain good bounds in many instances.

Potential functions. The potential functions used in the analysis are a combination of the two previous elements $(P; Q)$. The logical context P is a set of memory states valid at the point where the potential function is used; and the interval potential Q represents the "quantitative" part of the potential, it returns the potential associated to a given state. Formally, the meaning of the potential function $(P; Q)$ is

$$(P; Q)(m{::}\mu) := \mathsf{if}\ m \in P\ \mathsf{then}\ Q(m)\ \mathsf{else}\ \infty, \tag{4.2}$$

where, as explained above, we used $Q(m)$ for value of the potential function $\Phi_Q = \sum_{i \in I} q_i \cdot f_i$ associated to Q on the memory state m.

4.3.2 Inference Rules for the Interval System

Overview. The derivation rules for the interval system are defined in Figure 4.2 (p. 73). The essence of this set of rules is to specialize the tick quantitative logic with the potential functions of the previous section. Doing so, we lose completeness of the system, but simplify greatly the search for a valid derivation.

Like with a normal quantitative logic, the interval system enforces constraints between potential functions. However, because the potential functions now match a certain pattern, the constraints can be expressed much more precisely. In particular, the constraints on the coefficients of interval potential functions are simple linear arithmetic constraints that can be satisfied by a linear-programming solver.

On the other hand, the constraints on logical contexts are left quite general. This is because the inference of abstract program states is a well-known problem out of the scope of this dissertation. In a practical implementation of the automation, even the simplest abstract interpretation procedure will give satisfactory results on many examples, as we will see later.

Judgements of the interval system. The derivation rules of the interval system infer judgements of the form

$$\vdash \{P; Q\}\ c\ \{P_s; Q_s, P_b; Q_b\}. \tag{4.3}$$

These judgements, with the potential functions $(P; Q)$, $(P_s; Q_s)$, and $(P_b; Q_b)$ interpreted as described in Equation (4.2), must be read as triples of the tick quantitative logic of Figure 4.1 (p. 65). With the specific potential functions used by the automation, the triple of Equation 4.3 means: If c is executed with starting state $m{::}\mu$ where $m \in P$ (the call stack μ does not matter here because we chose potential functions that are independent of it), and at least $Q(m)$ resources units are available, then the evaluation does not run out of resources and, if the execution of c terminates normally in a state $m'{::}\mu$, there are at least $Q_s(m')$ resources units left and moreover $m' \in P_s$. The judgement also has a break postcondition in case the c command terminates by executing `break`. If that is the case, $Q_b(m')$ resource units are left and $m' \in P_b$.

Procedure specifications. Like in triples, the potential functions of procedure specifications are pairs of a logical context and an interval potential function. The auxiliary state $z \in Z$ provided by the tick quantitative logic is not used: Z instantiated with a trivial singleton set $\{*\}$. Thus, a procedure specification takes the form $\{P_e; Q_e\}\{P_x; Q_x\}$, where we omitted the unused auxiliary state.

The analysis critically relies on being able to give multiple procedure annotations for a single procedure. Indeed, it is often desirable to use different annotations for the different call sites of a procedure. For instance, a procedure incrementing two variables x and y might be called in two different contexts where only x, or only y matter for the reasoning. In that case, two different annotations are desirable because the language we used for potential functions is not rich enough to encode the two different behaviors in a single annotation.

Derivation rules. The complete set of rules for the interval system is given in the Figure 4.2 (p. 73). The following notations and conventions are used throughout the figure. When Q and Q' are quantitative annotations we assume that $Q = (q_i)_{i \in I}$ and $Q' = (q'_i)_{i \in I}$. The notation $Q \pm n$, used

$\boxed{\vdash \Gamma}$

$$\vdash \Gamma \quad := \quad \{P^e; Q^e\}\{P^x; Q^x\} \in \Gamma(p) \quad \Longrightarrow$$

$$\Gamma \vdash \{P^e; Q^e\}\ \Delta(p)\ \{P^x; Q^x, \top\} \quad \wedge \bigwedge_{x \notin \mathcal{G} \vee y \notin \mathcal{G}} q^e_{xy} = q^x_{xy} = 0$$

where $Q^e = (q^e_i)_i$ and $Q^x = (q^x_i)_i$.

$\boxed{\Gamma \vdash \{P; Q\}\ c\ \{P_s; Q_s, P_b; Q_b\}}$

$$\frac{\bigwedge_{u \neq v, v'} q_{v'u} = q'_{v'u} + q'_{vu} \wedge q_{uv'} = q'_{uv'} + q'_{uv} \qquad \bigwedge_{u \neq v'} q_{uv'} = q_{v'u} = 0 \qquad q'_{vv'}, q'_{v'v} \in \mathbb{Q}^+_0}{\vdash \{P[\llbracket v' \rrbracket_m / v]; Q\}\ v \leftarrow v'\ \{P; Q', \bot\}} \ \text{(A:Set)}$$

$$\frac{\begin{array}{c} P = P'[\llbracket v + \delta \rrbracket_m / v] \qquad \delta \neq v \\ P \models \delta > 0 \qquad S = \{u \mid P \models v + \delta \in [v, u]\} \qquad q'_{0\delta} = q_{0\delta} + \sum_{u \in S} q_{vu} - \sum_{u \notin S} q_{uv} \end{array}}{\vdash \{P; Q\}\ v \leftarrow v + \delta\ \{P'; Q', \bot\}} \ \text{(A:IncPos)}$$

$$\frac{\begin{array}{c} P = P'[\llbracket v + \delta \rrbracket_m / v] \qquad \delta \neq v \\ P \models \delta < 0 \qquad S = \{u \mid P \models v + \delta \in [u, v]\} \qquad q'_{\delta 0} = q_{\delta 0} + \sum_{u \in S} q_{uv} - \sum_{u \notin S} q_{vu} \end{array}}{\vdash \{P; Q\}\ v \leftarrow v + \delta\ \{P'; Q', \bot\}} \ \text{(A:IncNeg)}$$

$$\frac{\forall x_1, x_2 \in \mathbb{Z}.\, P[x_1/v] = P[x_2/v] \qquad \bigwedge_{u \in \mathcal{V}} q_{vu} = 0 \wedge q_{uv} = 0}{\vdash \{P; Q\}\ v \leftarrow \star\ \{P; Q, \bot\}} \ \text{(A:NonDet)}$$

$$\frac{}{\vdash \{P; Q\}\ \mathsf{guard}\ G\ \{P \cap G; Q, \bot\}} \ \text{(A:Guard)} \qquad \frac{}{\vdash \{P; Q\}\ \mathsf{skip}\ \{P; Q, \bot\}} \ \text{(A:Skip)}$$

$$\frac{}{\vdash \{P; Q + k\}\ \mathsf{tick}\ k\ \{P; Q, \bot\}} \ \text{(A:Tick)} \qquad \frac{}{\vdash \{P; Q\}\ \mathsf{break}\ \{\bot, P; Q\}} \ \text{(A:Break)}$$

Figure 4.2: Rules for the automated analysis. All the rules enforce the constraints of the tick quantitative logic by constraining the individual coefficients of the quantitative annotations. (Continued.)

$$\boxed{\Gamma \vdash \{P; Q\}\ c\ \{P_s; Q_s, P_b; Q_b\}}$$

$$\frac{\vdash \{P_l; Q_l\}\ c\ \{P_l; Q_l, P_b; Q_b\}}{\vdash \{P_l; Q_l\}\ \textsf{loop}\ c\ \{P_b; Q_b, \bot\}}\ (\text{A:}\textsc{Loop})$$

$$\frac{\vdash \{P; Q\}\ c_1\ \{P_s; Q_s, P_b; Q_b\} \qquad \vdash \{P_s; Q_s\}\ c_2\ \{P'_s; Q'_s, P_b; Q_b\}}{\vdash \{P; Q\}\ c_1; c_2\ \{P'_s; Q'_s, P_b; Q_b\}}\ (\text{A:}\textsc{Seq})$$

$$\frac{\vdash \{P; Q\}\ c_1\ \{P_s; Q_s, P_b; Q_b\} \qquad \vdash \{P; Q\}\ c_2\ \{P_s; Q_s, P_b; Q_b\}}{\vdash \{P; Q\}\ c_1 + c_2\ \{P_s; Q_s, P_b; Q_b\}}\ (\text{A:}\textsc{Alt})$$

$$\frac{\begin{array}{c}\{P^e; Q^e\}\{P^x; Q^x\} \in \Gamma(p) \qquad G = \{xy \mid x, y \in \mathcal{G}\} \qquad L = \{xy \mid x, y \notin \mathcal{G}\} \\ c \in \mathbb{Q}_0^+ \qquad \bigwedge_{i \in G} q_i = q_i^e \wedge q_i^x = q'_i \qquad \bigwedge_{i \in L} q_i = q'_i \qquad \bigwedge_{i \notin G \cup L} q_i = 0 \wedge q'_i = 0\end{array}}{\vdash \{P^e; Q + c\}\ \textsf{call}\ p\ \{P^x; Q' + c, \bot\}}\ (\text{A:}\textsc{Call})$$

$$\frac{\begin{array}{c}\vdash \{P; Q\}\ c\ \{P_s; Q_s, P_b; Q_b\} \\ P' \subseteq P \qquad Q' \succeq_{P'} Q \qquad P_s \subseteq P'_s \qquad Q_s \succeq_{P_s} Q'_s \qquad P_b \subseteq P'_b \qquad Q_b \succeq_{P_b} Q'_b\end{array}}{\vdash \{P'; Q'\}\ c\ \{P'_s; Q'_s, P'_b; Q'_b\}}\ (\text{A:}\textsc{Weak})$$

$$\frac{\begin{array}{c}L = \{xy \mid \exists l_{xy} \in \mathbb{N}.\, P \models l_{xy} \leqslant |[x, y]|\} \qquad U = \{xy \mid \exists u_{xy} \in \mathbb{N}.\, P \models |[x, y]| \leqslant u_{xy}\} \\ \forall i.\, p_i, r_i \in \mathbb{Q}_0^+ \qquad \bigwedge_{i \in U \setminus L} q'_i \geqslant q_i - r_i \qquad \bigwedge_{i \in L \setminus U} q'_i \geqslant q_i + p_i \qquad \bigwedge_{i \in L \cap U} q'_i \geqslant q_i + p_i - r_i \\ \bigwedge_{i \notin U \cup \mathcal{L} \cup \{0\}} q'_i \geqslant q_i \qquad q'_0 \geqslant q_0 + \sum_{i \in U} u_i r_i - \sum_{i \in L} l_i p_i\end{array}}{Q' \succeq_P Q}\ (\textsc{Relax})$$

Figure 4.2: Rules for the automated analysis. (End.)

in the rules A:Tick and A:Call, defines a new context Q' such that $q'_0 = q_0 \pm n$ and $\bigwedge_{i \neq 0} q'_i = q_i$. In all the rules, there is an implicit side condition that all rational coefficients are non-negative. If a rule mentions Q and Q' and leaves the latter undefined at some index i we assume that $q'_i = q_i$. Finally, since logical contexts are sets of memory states, they are manipulated with set-theoretic operations. The operation $P[e/v]$ is a shorthand notation for the set $\{m \mid m[[\![e]\!]_m/v] \in P\}$.

In the rest of this section, I will give an intuitive rationale for some selected rules of the system. In the next section, these intuitions are formalized into a concrete soundness proof. The model to keep in mind when reading a rule is that the precondition potential in an initial state must be enough to account for the tick consumption of the command in the triple and for the postcondition potential in the final state.

The A:Skip rule, for instance, applies to skip statements which do not consume any resource (i.e., do not make any changes to the tick counter) and do not change the memory state. Thus, the fallthrough postcondition can be reused as is. It would also be sound to have the rule derive $\vdash \{P; Q\}$ skip $\{P; Q', \bot\}$ with the side condition $Q \geqslant Q'$ because some potential is always soundly lost. For the sake of orthogonality, it is not how we present the rule, and this form of weakening is in fact expressed as the stand-alone rule A:Weak described later. Finally, note that the A:Skip rule is exactly the G:Skip rule of the tick quantitative logic only with specialized potential functions. More generally, all the commands that are purely control-related, like the break, the sequence, the loop, and the alternation commands all are direct specializations of their counterpart in the tick quantitative logic.

The tick k command has no effect on the memory state but it decrements the tick counter by k units. Because k tick units are consumed, the precondition potential $(P; Q + k)$ pays for the cost of the tick command. Because the memory state is left unchanged by the tick command, the interval potential function Q will evaluate to the same value before and after executing the tick k command, and the coefficients of program variable intervals $|[u, v]|$ do not have to be changed.

The rules Q:IncPos and Q:IncNeg describe how the potential is redistributed after a size change of a variable. The rule Q:IncPos is for incements $v \leftarrow v + \delta$ when $\delta > 0$ and the symmetric rule Q:IncNeg is for decrements when $\delta < 0$. These two rules are essentially the same, so we only focus on the explanation of Q:IncPos. In this case, the program updates a variable v with $v + \delta$. Since v is changed, the interval potential function must be updated to reflect the change of the program state. We write v' for the value of v after the assignment. Since v is the only variable changed, only intervals of the form $[u, v]$ and $[v, u]$ will be resized. Note that for any u,

$|[v,u]|$ will get smaller with the update, and if $v' \in [v,u]$, we have $|[v,u]| = |[v,v']| + |[v',u]|$. But $|[v,v']| = |[0,\delta]|$, which means that the potential $q'_{0\delta}$ of $|[0,\delta]|$ in the postcondition can be increased by q_{vu} under the guard that $v' \in [v,u]$ (i.e., $v \in S$ in the rule). Dually, the interval $[u,v]$ can only get bigger with the update. We know that $|[u,v']| \leqslant |[u,v]| + \delta$, so we decrease the potential of $|[0,\delta]|$ by q_{uv} to pay for this change. This decrease is only enforced by the rule when $u \notin S$, because otherwise $|[u,v]| = |[u,v']| = 0$.

The A:Call rule also enforces many constraints. However, it follows quite closely the pattern given by the tick quantitative logic. We can sum up the main ideas of the rule as follows.

— The purely global part of the pre- and postcondition potential is taken from the procedure specification of the callee.

— The potential of intervals $|[x,y]|$ is preserved across a function call when x and y are local variables.

— The unknown intervals after the call (e.g., $|[x,g]|$ with x local and g global) have their potential set to zero in the pre- and postcondition.

— Finally, a constant frame $c \geqslant 0$ can be added to the pre- and postcondition.

As I will show in the soundness proof, each of these components of the pre- and postcondition potentials is naturally matched with a piece of the quantitative logic G:Call rule.

The rule Q:Weak is the only one that is not syntax directed. In the implementation we apply Q:Weak before loops and between the two statements of a sequential composition. We could integrate weakenings into every syntax directed rule but this simple heuristic proved sufficient and generates less constraints. The high-level idea of Q:Weak is the following: if we have a sound judgement, then it is sound to add more potential to the precondition and remove potential from the postconditions. The concept of *more potential* is formalized by the relation $Q' \succeq_\Gamma Q$ that is defined in the rule Relax. This rule also deals with the important task of transferring constant potential (represented by q_0) to interval sizes and vice versa. If we can deduce from the logical context that the interval size $|[v,w]|$ is larger than a constant l then we can turn the potential $q_{vw} \cdot |[v,w]|$ form the interval into the constant potential $l \cdot q_{vw}$ and guarantee that we do not gain potential. Conversely, if $|[v,w]| \leqslant u$ for a constant u then we can transfer constant potential $u \cdot q_{vw}$ to the interval potential $q_{vw} \cdot |[v,w]|$ without gaining potential.

4.3.3 Soundness of the Analysis

The soundness of the analysis is justified using the soundness of the quantitative logic. Formally, we will see that a derivation in the interval system can be mapped to a derivation in the quantitative logic with the same pre- and postcondition. The mapping is straightforward: each rule is mapped to its corresponding rule in the quantitative logic. To define the translation between the two systems we proceed inductively.

Remember that Q represents the interval potential function $q_0 + \sum_{x,y \in \mathcal{V}} q_{xy} |[x, y]|$, and that for any two variables x and y, the function $|[x, y]|$ applies to a memory stack $m{::}\mu$ like so: $|[x, y]|(m{::}\mu) = |[m(x), m(y)]|$. Since the potential functions of the interval system do not inspect the call stack, we will omit it.

Procedure specifications. The procedure specification of the interval system $\{P_e; Q_e\}\{P_x; Q_x\}$ is mapped to the corresponding $[\forall z \in \{*\}. \{P_e; Q_e\}\{P_x; Q_x\}]$ in the quantitative logic. Essentially making no use of the auxiliary state available in the quantitative logic. Because we will see that any triple of the automation is valid in the quantitative logic, the validity of contexts is also preserved by translation.

• **Case** A:Set. To prove that the triple derived by A:Set is valid in the quantitative logic, we use the rule G:Set. Doing so requires to prove that

$$(P[[\![v']\!]_m/v]; Q)(m) = (P; Q')(m[[\![v']\!]_m/v]).$$

By definition of $P[[\![v']\!]_m/v]$, we can already conclude that the first components of the potential functions are equal. Remains to check that $Q(m) = Q'(m[[\![v']\!]_m/v]$. We now write m' for $m[[\![v']\!]_m/v]$. Let us see how each interval length $|[x, y]|$ is changed by the state update. If both x and y are different from v, then the interval is unchanged. If one of them, say x, is v, then $|[x, y]|(m') = |[v, y]|(m') = |[v', y]|(m)$. Remark that if, in addition, y is v', then $|[x, y]|(m') = 0$. Thus, putting it together we obtain

$$\begin{aligned} Q'(m') = q'_0 &+ \sum_{x,y \neq v,v'} q'_{xy} |[x, y]|(m') + \sum_{x \neq v,v'} q'_{vx} |[v, x]|(m') + \sum_{x \neq v,v'} q'_{xv} |[x, v]|(m') \\ &+ \sum_{x \neq v,v} q'_{v'x} |[v', x]|(m') + \sum_{x \neq v,v} q'_{xv'} |[x, v']|(m') \end{aligned} \tag{4.4}$$

$$= q_0 + \sum_{x,y \neq v,v'} q_{xy}|[x,y]|(m) + \sum_{x \neq v,v'} q'_{vx}|[v',x]|(m) + \sum_{x \neq v,v'} q'_{xv}|[x,v']|(m)$$
$$+ \sum_{x \neq v,v} q'_{v'x}|[v',x]|(m) + \sum_{x \neq v,v} q'_{xv'}|[x,v']|(m) \tag{4.5}$$
$$= q_0 + \sum_{x,y \neq v,v'} q_{xy}|[x,y]|(m') + \sum_{x \neq v,v'} q_{v'x}|[v',x]|(m) + \sum_{x \neq v,v'} q_{xv'}|[x,v']|(m) \tag{4.6}$$
$$= Q(m) \tag{4.7}$$

The step (4.4) is justified by definition of Q' and because $|[v,v']|(m') = |[v',v]|(m') = 0$; in step (4.5) we use the observations of the previous paragraph; in step (4.6) we used the constraints $q_{xv'} = q'_{xv} + q'_{xv'}$ and $q_{v'x} = q'_{vx} + q'_{v'x}$; finally, in step (4.7), we used that $q_{xv'} = q_{v'x} = 0$ for any $x \neq v'$. This concludes the proof of this case.

• **Case** A:IncPos **and** A:IncNeg. We will only treat the A:IncPos case in details; the rule IncNeg is handled by swapping the signs and the bounds of the intervals.

In the A:IncPos rule, we handle the case where a variable is incremented by a positive amount. Like in the previous paragraph, the logical part of the assertion does not cause any problems and we focus on the relation between Q and Q'. In this case, we will show that $Q(m) \geqslant Q'(m')$ where m' is $m[\llbracket v + \delta \rrbracket / v]$. That is, we apply the quantitative logic rule G:Set then, directly after, one consequence rule G:Conseq. The key to this proof is again to look at how each interval size is changed with the assignment. An interval size $|[x,v]|$, after the assignment, will grow by at most δ units. At most because, for example, if v is 8, x is 10, and δ is 3, the interval size $|[x,v]|$ grows by only 1. Similarly, an interval size $|[v,x]|$ will decrease, after the assignment, by at most δ. More precisely, if originally $v + \delta \in [v,x]$, then $|[v,x]|$ will decrease by exactly δ (i.e., $|[v,x]|(m') = |[v,x]|(m) - \delta$) and $|[x,v]|$ will remain 0 (i.e., $|[x,v]|(m') = |[x,v]|(m)$). Like in the rule, we write S the set of variables x such that, in the memory state m, $v + \delta \in [v,x]$. Then

$$Q'(m') = q'_0 + \sum_{x,y \neq v} q'_{xy}|[x,y]|(m') + \sum_{x \neq v} q'_{vx}|[v,x]|(m') + \sum_{x \neq v} q'_{xv}|[x,v]|(m') \tag{4.8}$$
$$\leqslant q'_0 + \sum_{x,y \neq v} q'_{xy}|[x,y]|(m) + \sum_{x \in S} q'_{vx}(|[v,x]|(m) - \delta) + q'_{xv}|[x,v]|(m)$$
$$+ \sum_{x \notin S} q'_{vx}|[v,x]|(m) + q'_{xv}(|[x,v]|(m) + \delta) \tag{4.9}$$
$$= q_0 + \sum_{(x,y) \neq (0,\delta)} q_{xy}|[x,y]|(m) + q'_{0\delta}\delta - \sum_{x \in S} q_{vx}\delta + \sum_{x \notin S} q_{xv}\delta \tag{4.10}$$

$$= q_0 + \sum_{(x,y)\neq(0,\delta)} q_{xy}|[x,y]|(m) + q_{0\delta}\delta = Q(m) \tag{4.11}$$

In step (4.8) we reassociated the definition of Q'; in step (4.9) we noted δ the integer $m'(\delta)$ (that is equal to $|[0,\delta]|(m')$ and $|[0,\delta]|(m)$ since δ is positive at that point), and we used the reasoning about interval sizes of the previous paragraph; in step (4.10) we reassociated the sum and used the constraints (i.e., $q_{xy} = q'_{xy}$ for any pair of variable $(x,y) \neq (0,\delta)$); and finally in step (4.11) we used the constraint on $q'_{0\delta}$. This concludes the proof of this case.

• **Case** A:NonDet. This rule uses $P; Q$ as both pre- and postcondition. The condition imposed on P makes sure that it is independent of the value of the variable v assigned. Similarly, all the base functions that depend on v in Q: the interval lengths $|[x,v]|$ and $|[v,x]|$ for any variable x have their coefficient constrained to be 0. Thus

$$\begin{aligned}\max\{(P;Q)(m[k/v]) \mid k \in \mathbb{Z}\} &= (P;Q)(m[[\![v]\!]_m/v]) \\ &= (P;Q)(m).\end{aligned}$$

And the triple $\vdash \{P;Q\}\ v \leftarrow \star\ \{P;Q,\bot\}$ can be derived by applying the G:NonDet rule.

• **Case** A:Guard. We apply the G:Guard rule and a chain it with a consequence rule G:Conseq to weaken the precondition by observing that

$$\begin{aligned}& \textsf{if } m \in G \textsf{ then } (P \cap G; Q)(m) \textsf{ else } 0 \\ = \;& \textsf{if } m \in G \textsf{ then } (\textsf{if } m \in P \cap G \textsf{ then } Q(m) \textsf{ else } \infty) \textsf{ else } 0 && (4.12) \\ = \;& \textsf{if } m \in G \textsf{ then } (\textsf{if } m \in P \textsf{ then } Q(m) \textsf{ else } \infty) \textsf{ else } 0 && (4.13) \\ \leqslant \;& \textsf{if } m \in P \textsf{ then } Q(m) \textsf{ else } \infty = (P;Q)(m). && (4.14)\end{aligned}$$

The step (4.12) expands the definition of $(P \cap G; Q)$; the step (4.13) is justified by the fact that in the then branch, $m \in G$; and the step (4.14) is justified by remarking that the then branch of (4.13) is an upper-bound on the else branch. This concludes the proof of this case.

• **Case** A:Weak. Observe that to prove $(P';Q') \geqslant (P;Q)$ it is sufficient to show that $P' \subseteq P$, and that for any memory state $m \in P'$, $Q'(m) \geqslant Q(m)$. This is clear when unfolding the definition of $(P;Q)$. We will use this reasoning principle to show that the rule A:Weak corresponds directly to an application of G:Conseq.

The constraints on the logical parts of the potential functions are readily met by hypothesis of the rule A:Weak; thus, we focus on the quantitative part. To show the required inequalities, we prove that the Relax rule encodes an order constraint:

$$\text{If } Q' \succeq_P Q, \text{ then for all } m \in P,\ Q'(m) \geqslant Q(m).$$

Let Q, Q', P, and m satisfying $Q' \succeq_P Q$ and $m \in P$. By hypothesis of the rule Relax, we have two sets $L = \{xy \mid \exists l_{xy} \in \mathbb{N}.\, l_{xy} \leqslant |[x,y]|(m)\}$ and $U = \{xy \mid \exists u_{xy} \in \mathbb{N}.\, |[x,y]|(m) \leqslant u_{xy}\}$. Then

$$\begin{aligned}
Q'(m) &= q'_0 + \sum_{xy \notin U \cup L} q'_{xy}|[x,y]|(m) + \sum_{xy \in U \cup L} q'_{xy}|[x,y]|(m) && (4.15)\\
&\geqslant q_0 + \sum_{i \in U} u_i r_i - \sum_{i \in L} l_i p_i + \sum_{xy \notin U \cup L} q_{xy}|[x,y]|(m) + \sum_{xy \in U \cup L} q'_{xy}|[x,y]|(m) && (4.16)\\
&= q_0 + \sum_{xy \in L \setminus U} (q'_{xy}|[x,y]|(m) - l_{xy}p_{xy}) + \sum_{xy \in U \setminus L} (q'_{xy}|[x,y]|(m) + u_{xy}r_{xy})\\
&\quad + \sum_{xy \in L \cap U} (q'_{xy}|[x,y]|(m) + u_{xy}r_{xy} - l_{xy}p_{xy}) + \sum_{xy \notin U \cup L} q_{xy}|[x,y]|(m) && (4.17)\\
&\geqslant q_0 + \sum_{xy \in L \setminus U} (q'_{xy} - p_{xy})|[x,y]|(m) + \sum_{xy \in U \setminus L} (q'_{xy} + r_{xy})|[x,y]|(m)\\
&\quad + \sum_{xy \in L \cap U} (q'_{xy} + r_{xy} - p_{xy})|[x,y]|(m) + \sum_{xy \notin U \cup L} q_{xy}|[x,y]|(m) && (4.18)\\
&\geqslant q_0 + \sum_{xy \in L \setminus U} q_{xy}|[x,y]|(m) + \sum_{xy \in U \setminus L} q_{xy}|[x,y]|(m)\\
&\quad + \sum_{xy \in L \cap U} q_{xy}|[x,y]|(m) + \sum_{xy \notin U \cup L} q_{xy}|[x,y]|(m) && (4.19)\\
&= Q(m). && (4.20)
\end{aligned}$$

Even though the notations are heavy, the reasoning is elementary: In step (4.15) we simply unfold the definition and split the sum over intervals in two; in step (4.16) we use the constraints on q_0 and q_{xy} when $xy \notin U \cup L$; in step (4.17) we reorder the summations conveniently; in step (4.18) we use the definitions of the sets U and L; in step (4.19) we use the rest of the constraints from the Relax rule; finally in step (4.20) we fold the definition of Q. This complete the proof of the lemma about the relax rule. To conclude the validity of the G:Conseq application, we apply this lemma to the precondition and to the two postconditions.

• **Case** A:Call. Applications of A:Call are translated directly to an application of G:Call. Since the auxiliary state Z for translated specifications is the trivial singleton $\{*\}$, there is no choice to make for the variable $z \in Z$ in G:Call. The frame L used in G:Call will be noted L' from now on, as it conflicts the set of indices L from the rule A:Call. Remember the rule A:Call:

$$\frac{\begin{array}{llll}\{P^e;Q^e\}\{P^x;Q^x\} \in \Gamma(p) & G = \{xy \mid x,y \in \mathcal{G}\} & & L = \{xy \mid x,y \notin \mathcal{G}\} \\ c \in \mathbb{Q}_0^+ \quad \bigwedge_{i\in G} q_i = q_i^e \wedge q_i^x = q_i' & \bigwedge_{i\in L} q_i = q_i' & & \bigwedge_{i\notin G\cup L} q_i = 0 \wedge q_i' = 0\end{array}}{\vdash \{P^e;Q+c\} \text{ call } p \{P^x;Q'+c,\bot\}}$$

Essentially, the potential before and after the call is constrained to be, respectively, $(P^e;Q^e+X+c)$ and $(P^x;Q^x+X+c)$, where $X = \sum_{x,y\notin\mathcal{G}} q_{xy} \cdot |[x,y]|$ only depends on local variables. This is because, assuming the procedure specification chosen is in a valid context, all the coefficients of Q^e and Q^x for the indices that are not in G are zero—i.e., valid procedure specifications can only depend on global variables. The frame L' we are going to use is thus $L' = X + c$. As required, it only depends on local variables and maps to non-negative rational numbers. Then, observe that

$$\begin{aligned} Q^e(m{::}\mu) &= Q^e(m) = Q^e(0 \cup_{\mathcal{G}} m) = Q^e((0 \cup_{\mathcal{G}} m){::}m{::}\mu), \text{ and} \\ Q^x(m{::}\mu) &= Q^x(m) = \min\{Q^x(m') \mid m'' \cup_{\mathcal{G}} m' = m\} \\ &= \min\{Q^x(m'{::}m''{::}\mu) \mid m'' \cup_{\mathcal{G}} m' = m\}. \end{aligned}$$

The core of the two reasoning above is that interval potential functions only depend on the topmost memory state in their memory stack argument, and that Q^e and Q^x only depend on global variables.

These observations prove that the triple $\vdash \{Q^e + L'\}$ call p $\{Q^x + L'\}$ is derivable by application of G:Call. Finally, conclude by observing that $Q^e + L' = Q^e + X + c = Q + c$ and similarly $Q^x + L' = Q' + c$.

• **Case** A:Skip**,** A:Break**,** A:Tick**,** A:Loop**,** A:Seq**, and** A:Alt**.** By direct application of the corresponding rule in the quantitative logic and, when applicable, using the hypothesis that the premiss triples are valid in the quantitative logic.

Conclusion. We just saw that, inductively, any derivation of the interval system can be translated as a derivation of the quantitative logic with the same conclusion. Similarly, valid contexts of the automation are valid contexts of the quantitative logic. This shows that any inference made by the automation provides with a correct resource bound for ticks. The interval derivation system is not complete, but as I explain in the next section, its major advantage is that it can be automated with linear programming.

4.3.4 Automation using Linear Programming

The search of a derivation is split in two steps. First, a derivation template is created by inductively applying syntax directed rules as we go. The coefficients of the linear interval potential functions are left as symbolic names and the constraints they must satisfy for each rule application to be valid are collected along the way. Second, we pass the collected constraints to an off-the-shelf linear-programming solver. If the solver successfully finds a solution, we know that a derivation exists and extract the coefficients of the initial potential Q from the solution to get a resource bound for the program. To get a full derivation, we extract the complete solution from the solver and apply it to the symbolic names of the coefficients in the derivation. If the linear-programming solver fails to find a solution, an error is reported.

If the command to analyze makes calls to procedures—and, transitively, these procedures also make calls—the analysis is recursively called to generates derivations for the procedure bodies transitively called. This technique is essentially equivalent to inlining the body of the called procedures. When recursion is involved, this inlining strategy could lead to non-termination of the analysis. Thus, as the analysis procedure progresses, a "context" of active procedure specifications is maintained. When a call to a procedure that already has a specification in the active context is encountered, the annotation of the context is used.

Figure 4.3 (p. 83) displays a partial pseudocode implementation of the constraint-generating phase of C4B. It defines a function `cgen` that recursively inspects its argument `cmd` and generates two symbolic interval potential functions. Informally, if the returned value is `(pre, pst)` and all the constraints generated along the way can be satisfied, the triple $\vdash \{\mathsf{pre}\}\ \mathsf{cmd}\ \{\mathsf{pst}\}$ can be derived in the interval system. The actual implementation also has to handle logical contexts and break postconditions. Additionally, it is optimized to limit the number of symbolic names generated.

Detailed example. Figure 4.4 (p. 84) contains an example derivation as produced by C4B. The upper case letters (with optional superscript) such as Q^{de} are families of variables that are later part of the constraint system that is passed to the linear-programming solver. For example Q^{de} stands for the potential function $q^{\text{de}}_0 + q^{\text{de}}_{x\,0}|[x,0]| + q^{\text{de}}_{0\,x}|[0,x]| + q^{\text{de}}_{x\,10}|[x,10]| + q^{\text{de}}_{10\,x}|[10,x]| + q^{\text{de}}_{0\,10}|[0,10]|$,

```
let rec cgen active cmd =
 match cmd with
 | CSkip -> (* When the command is skip *)
   let (pre, pst) = new_intp () in
   constrain (pre = pst); (* To apply A:SKIP *)
   (pre, pst)
 | CSeq (c1, c2) -> (* When the command is $c_1; c_2$ *)
   let (pre1, pst1) = cgen context c1 in
   let (pre2, pst2) = cgen context c2 in
   constrain (pst1 = pre2); (* To apply A:SEQ *)
   (pre1, pst2)
 | CCall p -> (* When the command is call p *)
   if ∃ (pre, pst). { p ↦ (pre, pst) } ∈ active then
     (pre, pst)
   else
     let (pre, pst) = new_quantitative_annots () in
     let (pre’, pst’) =
       (* Add the specification to the active context in case of recursion *)
       cgen ({ p ↦ (pre, pst) } :: active) (body p) in
     constrain (pre = pre’ ∧ pst = pst’);
     (pre, pst)
 | ... (* Other cases. *)
```

Figure 4.3: OCaml-like pseudocode for the constraint-generating algorithm in the implementation C4B. The function cgen generates constraints to build a valid derivation for its argument command cmd; it returns the tentative pre- and postcondition of the argument command as quantitative annotations with indeterminate coefficients.

$$\dfrac{}{\vdash \{x{\geqslant}10; Q^{\mathrm{de}}\}\ x \leftarrow x - 10\ \{\cdot; Q_s^{\mathrm{de}}, \bot\}}\ (\text{A:IncNeg})$$

$$\dfrac{\vdash \{x{\geqslant}10; Q^{\mathrm{de}}\}\ x \leftarrow x - 10\ \{\cdot; Q_s^{\mathrm{de}}, \bot\}}{\vdash \{x{\geqslant}10; Q^{\mathrm{w3}}\}\ x \leftarrow x - 10\ \{\cdot; Q_s^{\mathrm{w3}}, \cdot; Q_b^{\mathrm{w3}}\}}\ (\text{A:Weak}_3) \qquad \dfrac{}{\vdash \{\cdot; Q^{\mathrm{ti}}\}\ \mathsf{tick}\ 5\ \{\cdot; Q_s^{\mathrm{ti}}, \bot\}}\ (\text{A:Tick})$$

$$\dfrac{\vdash \{x{\geqslant}10; Q^{\mathrm{w3}}\}\ x \leftarrow x - 10\ \{\cdot; Q_s^{\mathrm{w3}}, \cdot; Q_b^{\mathrm{w3}}\} \qquad \vdash \{\cdot; Q^{\mathrm{ti}}\}\ \mathsf{tick}\ 5\ \{\cdot; Q_s^{\mathrm{ti}}, \bot\}}{\vdash \{x{\geqslant}10; Q^{\mathrm{sq}}\}\ x \leftarrow x - 10;\ \mathsf{tick}\ 5\ \{\cdot; Q_s^{\mathrm{sq}}, \cdot; Q_b^{\mathrm{sq}}\}}\ (\text{A:Seq}_2)$$

$$\dfrac{\vdash \{x{\geqslant}10; Q^{\mathrm{sq}}\}\ x \leftarrow x - 10;\ \mathsf{tick}\ 5\ \{\cdot; Q_s^{\mathrm{sq}}, \cdot; Q_b^{\mathrm{sq}}\}}{\vdash \{x{\geqslant}10; Q^{\mathrm{w2}}\}\ x \leftarrow x - 10;\ \mathsf{tick}\ 5\ \{\cdot; Q_s^{\mathrm{w2}}, \cdot; Q_b^{\mathrm{w2}}\}}\ (\text{A:Weak}_2)$$

$$\dfrac{}{\vdash \{\cdot; Q^{\mathrm{br}}\}\ \mathsf{break}\ \{\bot, \cdot; Q_b^{\mathrm{br}}\}}\ (\text{A:Break})$$

$$\dfrac{\vdash \{\cdot; Q^{\mathrm{br}}\}\ \mathsf{break}\ \{\bot, \cdot; Q_b^{\mathrm{br}}\}}{\vdash \{\cdot; Q^{\mathrm{ar}}\}\ \mathsf{break}\ \{\cdot; Q_s^{\mathrm{ar}}, \cdot; Q_b^{\mathrm{ar}}\}}\ (\text{A:Weak}_4)$$

$$\dfrac{}{\vdash \{.; Q^{\mathrm{gd}}\}\ \mathsf{guard}\ (x \geqslant 10)\ \{x{\geqslant}10; Q_s^{\mathrm{gd}}, \bot\}}\ (\text{A:Guard})$$

$$\dfrac{\vdash \{.; Q^{\mathrm{gd}}\}\ \mathsf{guard}\ (x \geqslant 10)\ \{x{\geqslant}10; Q_s^{\mathrm{gd}}, \bot\}}{\vdash \{.; Q^{\mathrm{w1}}\}\ \mathsf{guard}\ (x \geqslant 10)\ \{x{\geqslant}10; Q_s^{\mathrm{w1}}, \cdot; Q_b^{\mathrm{w1}}\}}\ (\text{A:Weak}_1)$$

$$\dfrac{\vdash \{.; Q^{\mathrm{w1}}\}\ \mathsf{guard}\ (x \geqslant 10)\ \{x{\geqslant}10; Q_s^{\mathrm{w1}}, \cdot; Q_b^{\mathrm{w1}}\} \qquad \vdots}{\vdash \{\cdot; Q^{\mathrm{al}}\}\ \mathsf{guard}\ (x \geqslant 10); x \leftarrow x - 10;\ \mathsf{tick}\ 5\ \{\cdot; Q_s^{\mathrm{al}}, \cdot; Q_b^{\mathrm{al}}\}}\ (\text{A:Seq}_1)$$

$$\dfrac{\vdash \{\cdot; Q^{\mathrm{al}}\}\ \mathsf{guard}\ (x \geqslant 10); x \leftarrow x - 10;\ \mathsf{tick}\ 5\ \{\cdot; Q_s^{\mathrm{al}}, \cdot; Q_b^{\mathrm{al}}\} \qquad \vdots}{\vdash \{\cdot; Q^{\mathrm{lo}}\}\ (\mathsf{guard}\ (x \geqslant 10);\ x \leftarrow x - 10;\ \mathsf{tick}\ 5) + \mathsf{break}\ \{\cdot; Q_s^{\mathrm{lo}}, \cdot; Q_b^{\mathrm{lo}}\}}\ (\text{A:Alt})$$

$$\dfrac{\vdash \{\cdot; Q^{\mathrm{lo}}\}\ (\mathsf{guard}\ (x \geqslant 10);\ x \leftarrow x - 10;\ \mathsf{tick}\ 5) + \mathsf{break}\ \{\cdot; Q_s^{\mathrm{lo}}, \cdot; Q_b^{\mathrm{lo}}\}}{\vdash \{\cdot; Q\}\ \mathsf{loop}\ \{(\mathsf{guard}\ (x \geqslant 10);\ x \leftarrow x - 10;\ \mathsf{tick}\ 5) + \mathsf{break}\}\ \{\cdot; Q_s, \bot\}}\ (\text{A:Loop})$$

Constraints:

$$\begin{array}{lr}
Q = Q^{\mathrm{lo}} = Q_s^{\mathrm{lo}} \wedge Q_s = Q_b^{\mathrm{lo}} & (\text{A:Loop}) \\
Q^{\mathrm{ar}} = Q^{\mathrm{al}} = Q^{\mathrm{lo}} \wedge Q_s^{\mathrm{ar}} = Q_s^{\mathrm{al}} = Q_s^{\mathrm{lo}} \wedge Q_b^{\mathrm{ar}} = Q_b^{\mathrm{al}} = Q_b^{\mathrm{lo}} & (\text{A:Alt}) \\
Q^{\mathrm{al}} = Q^{\mathrm{w1}} \wedge Q_s^{\mathrm{w1}} = Q^{\mathrm{w2}} \wedge Q_s^{\mathrm{al}} = Q_s^{\mathrm{w2}} \wedge Q_b^{\mathrm{al}} = Q_b^{\mathrm{w1}} = Q_b^{\mathrm{w2}} & (\text{A:Seq}_1) \\
Q^{\mathrm{w1}} \succeq_{(\cdot)} Q^{\mathrm{gd}} \wedge Q_s^{\mathrm{gd}} \succeq_{(x\geqslant 10)} Q_s^{\mathrm{w1}} & (\text{A:Weak}_1) \\
Q^{\mathrm{gd}} = Q_s^{\mathrm{gd}} & (\text{A:Guard})
\end{array}$$

$$\begin{array}{lr}
Q^{\mathrm{w2}} \succeq_{(x\geqslant 10)} Q^{\mathrm{sq}} \wedge Q_s^{\mathrm{sq}} \succeq_{(\cdot)} Q_s^{\mathrm{w2}} \wedge Q_b^{\mathrm{sq}} \succeq_{(\cdot)} Q_b^{\mathrm{w2}} & (\text{A:Weak}_2) \\
Q^{\mathrm{sq}} = Q^{\mathrm{w3}} \wedge Q_s^{\mathrm{w3}} = Q^{\mathrm{ti}} \wedge Q_s^{\mathrm{ti}} = Q_s^{\mathrm{sq}} \wedge Q_b^{\mathrm{sq}} = Q_b^{\mathrm{w3}} & (\text{A:Seq}_2) \\
Q^{\mathrm{ti}} = Q_s^{\mathrm{ti}} + 5 & (\text{A:Tick}) \\
Q^{\mathrm{w3}} \succeq_{(x\geqslant 10)} Q^{\mathrm{de}} \wedge Q_s^{\mathrm{de}} \succeq_{(\cdot)} Q_s^{\mathrm{w3}} & (\text{A:Weak}_3) \\
q^{\mathrm{de}}_{s\,0\,10} = q^{\mathrm{de}}_{0\,10} + q^{\mathrm{de}}_{0\,x} \wedge q^{\mathrm{de}}_{s\,0} = q^{\mathrm{de}}_{0} \wedge \bigwedge_{\alpha\beta \neq 0\,10} q^{\mathrm{de}}_{s\,\alpha\beta} = q^{\mathrm{de}}_{\alpha\beta} & (\text{A:IncNeg})
\end{array}$$

$$\begin{array}{lr}
Q^{\mathrm{ar}} \succeq_{(\cdot)} Q^{\mathrm{br}} \wedge Q_b^{\mathrm{br}} \succeq_{(\cdot)} Q_b^{\mathrm{ar}} & (\text{A:Weak}_4) \\
Q^{\mathrm{br}} = Q_b^{\mathrm{br}} & (\text{A:Break})
\end{array}$$

Linear Objective Function: $1 \cdot q_{x\,0} + 10000 \cdot q_{0\,x} + 11 \cdot q_{x\,10} + 9990 \cdot q_{10\,x}$

Constant Objective Function: $1 \cdot q_0 + 11 \cdot q_{0\,10}$

Figure 4.4: An example symbolic derivation as produced by the constraint generation algorithm of C4B. The constraints are resolved by an off-the-shelf linear-programming solver.

where the variables such as $q^{\mathrm{de}}_{x\,10}$ are yet unknown and later instantiated by the linear-programming solver.

In general, the weakening rule can be applied after every syntax directed rule. However, it can be left out in practice at some places to decrease the number of constraints generated. The weakening operation $\succeq_\Gamma$ is defined by the rule Relax. It is parameterized by a logical context that is used to gather information on interval sizes. For example,

$$\begin{aligned} Q^{\mathrm{w1}} \succeq_{(\cdot)} Q^{\mathrm{gd}} \equiv\ & q^{\mathrm{gd}}_{0\,10} \leqslant q^{\mathrm{w1}}_{0\,10} + u_{0\,10} - v_{0\,10} \\ \wedge\ & q^{\mathrm{gd}}_{0} \leqslant q^{\mathrm{w1}}_{0} - 10 \cdot u_{0\,10} + 10 \cdot v_{0\,10} \\ \wedge\ & \bigwedge_{\alpha\beta \neq 0\,10} q^{\mathrm{gd}}_{\alpha\beta} \leqslant q^{\mathrm{w1}}_{\alpha\beta}\ . \end{aligned}$$

The other rules are syntax directed and applied inductively. For example, the outermost expression is a loop, so we use the rule A:Loop at the root of the derivation tree. At this point, we do not know yet whether a loop invariant exists or not. But we produce the constraints $Q^{\mathrm{lo}} = Q^{\mathrm{lo}}_s$, which is short for the following constraint set.

$$\begin{array}{lll} q^{\mathrm{lo}}_{0} = q^{\mathrm{lo}/s}_{0} & q^{\mathrm{lo}}_{x\,0} = q^{\mathrm{lo}/s}_{x\,0} & q^{\mathrm{lo}}_{0\,x} = q^{\mathrm{lo}/s}_{0\,x} \\ q^{\mathrm{lo}}_{x\,10} = q^{\mathrm{lo}/s}_{x\,10} & q^{\mathrm{lo}}_{10\,x} = q^{\mathrm{lo}/s}_{10\,x} & q^{\mathrm{lo}}_{0\,10} = q^{\mathrm{lo}/s}_{0\,10} \end{array}$$

These constraints express the fact that the potential functions before and after the loop body are equal and thus constitute an invariant.

After the constraint generation, the solver is provided with an objective function to minimize. We wish to minimize the initial potential, which is a resource bound on the whole program. Here it is given by Q. Moreover, we would like to express that minimization of linear potential such as $q_{10\,x}|[10, x]|$ takes priority over minimization of constant potential such as $q_{0\,10}|[0, 10]|$.

To get a tight bound, we use modern linear-programming solvers that allow constraint solving and minimization at the same time. First we consider our initial constraint set as given in Figure 4.4 (p. 84) and ask the solver to find a solution that satisfies the constraints and minimizes the linear expression

$$1 \cdot q_{x\,0} + 10000 \cdot q_{0\,x} + 11 \cdot q_{x\,10} + 9990 \cdot q_{10\,x}\ .$$

The penalties given to certain factors are used to prioritize certain intervals. For example, a bound with $|[10, x]|$ will be preferred to another with $|[0, x]|$ because $|[10, x]| \leqslant |[0, x]|$. The solver now returns a solution of the constraint set and an objective value, that is, a mapping from variables to floating-point numbers and the value of the objective function with this instantiation. The solver also memorizes the optimization path that led to the optimal solution. In this case, the objective value would be 5000 since the solver assigns $q_{0\,x} = 0.5$ and $q_* = 0$ otherwise. We now add the constraint

$$1 \cdot q_{x\,0} + 10000 \cdot q_{0\,x} + 11 \cdot q_{x\,10} + 9990 \cdot q_{10\,x} \leqslant 5000$$

to our constraint set and ask the solver to optimize the objective function

$$q_0 + 11 \cdot q_{0\,10} \ .$$

This happens in almost no time in practice. The final solution is $q_{0\,x} = 0.5$ and $q_* = 0$ otherwise. Thus the derived bound is $0.5|[0, x]|$.

A notable advantage of the linear-programming-based approach compared to SMT-solver-based techniques is that a satisfying assignment is a proof certificate instead of a counter example. In Chapter 6, I will show how to us the assignment returned by the linear-programming solver to create Coq certificates that validate the resource bounds found by the analysis.

4.4 Examples Programs with their Resource Bounds

In this section, I show that despite being quite simple, the interval system can infer bounds on multiple non-trivial programs. These bounds are often tight and enjoy a very simple representation using the interval notation $|[a, b]|$.

Challenging loops. One might think that our set of potential functions is too restricted to be able to express and prove bounds for realistic programs. Nevertheless, it can handle challenging example programs without special tricks or techniques. Examples *speed_1* and *speed_2* in Figure 4.5 (p. 87), which are taken from previous work [GMC09], demonstrate that the interval system can handle tricky iteration patterns. The SPEED tool [GMC09] derives the same bounds but requires undocumented heuristics for its counter instrumentation. These loops can also be handled with inference of disjunctive invariants but, in the abstract interpretation community, these invariants are known to be notoriously difficult to generate. In Example *speed_1* we have one loop that first increments variable y up to m and then increments variable x up to n. The automation derives the

```
while n > x:
    $\{n>x;\ |[x,n]|+|[y,m]|\}$
    if m > y:
        $\{m>y;\ |[x,n]|+|[y,m]|\}$
        y = y + 1
        $\{\cdot;\ 1+|[x,n]|+|[y,m]|\}$
    else:
        $\{n>x;\ |[x,n]|+|[y,m]|\}$
        x = x + 1
        $\{\cdot;\ 1+|[x,n]|+|[y,m]|\}$
    $\{\cdot;\ 1+|[x,n]|+|[y,m]|\}$
    tick 1
$\{\cdot;\ |[x,n]|+|[y,m]|\}$
```

$|[x,n]| + |[y,m]|$

speed_1

```
while x < n:
    $\{x<n;\ |[x,n]|+|[z,n]|\}$
    if z > x:
        $\{x<n;\ |[x,n]|+|[z,n]|\}$
        x = x + 1
        $\{\cdot;\ 1+|[x,n]|+|[z,n]|\}$
    else:
        $\{z\leqslant x, x<n;\ |[x,n]|+|[z,n]|\}$
        z = z + 1
        $\{\cdot;\ 1+|[x,n]|+|[z,n]|\}$
    $\{\cdot;\ 1+|[x,n]|+|[z,n]|\}$
    tick 1
$\{\cdot;\ |[x,n]|+|[z,n]|\}$
```

$|[x,n]| + |[z,n]|$

speed_2

```
while z - y > 0:
    $\{y<z;\ 3.1|[y,z]|+0.1|[0,y]|\}$
    y = y + 1
    $\{\cdot;\ 3+3.1|[y,z]|+0.1|[0,y]|\}$
    tick 3
    $\{\cdot;\ 3.1|[y,z]|+0.1|[0,y]|\}$
$\{\cdot;\ 3.1|[y,z]|+0.1|[0,y]|\}$
while y > 9:
    $\{y>9;\ 3.1|[y,z]|+0.1|[0,y]|\}$
    y = y - 10
    $\{\cdot;\ 1+3.1|[y,z]|+0.1|[0,y]|\}$
    tick 1
$\{\cdot;\ 3.1|[y,z]|+0.1|[0,y]|\}$
```

$3.1|[y,z]| + 0.1|[0,y]|$

t08a

```
while n < 0:
    $\{n<0;\ 59|[n,0]|+0.05|[0,y]|\}$
    n = n + 1
    $\{\cdot;\ 59+59|[n,0]|+0.05|[0,y]|\}$
    y = y + 1000
    $\{\cdot;\ 9+59|[n,0]|+0.05|[0,y]|\}$
    while y >= 100 and random:
        $\{y>99;\ 9+59|[n,0]|+0.05|[0,y]|\}$
        y = y - 100
        $\{\cdot;\ 14+59|[n,0]|+0.05|[0,y]|\}$
        tick 5
    $\{\cdot;\ 9+59|[n,0]|+0.05|[0,y]|\}$
    tick 9
$\{\cdot;\ P(n,y)\}$
```

$59|[n,0]|+0.05|[0,y]|$

t27

Figure 4.5: Derivations of bounds on the number of ticks for challenging examples. (Part 1.)

```
def c_down():
   if x > y:
      tick 1
      x = x - 1
      c_up()

def c_up():
   if y + 1 < x:
      tick 1
      y = y + 2
      c_down()
```

$0.33 + 0.67\lvert[y, x]\rvert$	`c_down()`
$0.67\lvert[y, x]\rvert$	`c_up()`

t39

```
while n >= 8:
   # process one block
   tick N
   n = n - 8
while n > 0:
   # save leftovers
   tick 1
   n = n - 1
```

$\frac{N}{8}\lvert[0, n]\rvert$	if $N \geqslant 8$
$7\frac{8-N}{8} + \frac{N}{8}\lvert[0, n]\rvert$	if $N < 8$

t61

```
while true:
   repeat:
      l = l + 1
      tick 1
   while l < h and random
   repeat:
      h = h - 1
      tick 1
   while h > l and random
   if h <= l:
      break
   tick 1
   # swap the two elements
```

$2 + 3\lvert[l, h]\rvert$

t62

Figure 4.6: Derivations of bounds on the number of ticks for challenging examples. (Part 2.)

tight bound $|[x, n]| + |[y, m]|$. Example *speed_2* is even trickier, and finding a bound manually is not trivial. However, using potential transfer reasoning as in amortized analysis, it is easy to prove the tight bound $|[x, n]| + |[z, n]|$.

Nested and sequenced loops. Example *t08a* in Figure 4.5 (p. 87) shows the ability of the analysis to discover interaction between *sequenced loops* through size change of variables. It accurately tracks the size change of y in the first loop by transferring the potential 0.1 from $|[y, z]|$ to $|[0, y]|$. Furthermore, *t08a* shows again that we do not handle the constants 1 or 0 in any special way. In all examples we could replace 0 and 1 with other constants like in the second loop and still derive a tight bound. The only information, that the analyzer needs is $y \geqslant c$ before assigning `y = y - c`. Example *t27* in Figure 4.5 (p. 87) shows how amortization can be used to handle *interacting nested loops*. In the outer loop we increment the variable n until $n = 0$. In each of the $|[n, 0]|$ iterations, we increment the variable y by 1000. Then we non-deterministically execute an inner loop that decrements y by 100 until $y < 100$. The analysis discovers that only the first execution of the inner loop depends on the initial value of y. It again derives tight constant factors.

Mutually recursive functions. Because it is based on a quantitative logic with support for procedure calls, the analysis handles mutual recursion. Example *t39* in Figure 4.6 (p. 88) contains two mutually-recursive functions with their automatically derived resource bounds. The function `c_down` decrements its first argument x until it reaches the second argument y. It then recursively calls the function `c_up`, which is dual to `c_down`: `c_up` increments y by 2 and calls `c_down`. C4B is the only available system that computes a tight bound. The analysis amounts to computing the meeting point of two vehicles that approach each other with different speeds.

Compositionality. With two concrete examples from open-source projects I demonstrate the compositionality of the potential-based analysis. The compositionality provides a uniform framework for the analysis of complete programs: loops are always analyzed in the context of their program, and the analysis naturally performs iteration bounding and size change analysis concurrently.

Example *t61* in Figure 4.6 (p. 88) is a typical pattern in implementations of block-based cryptographic primitives: data of arbitrary length is consumed in blocks and the leftover is stored in a buffer for future use when more data is available. This pattern is present in all the block encryption routines of PGP and is also used in performance critical code to unroll loops. For example, it is used in a bit manipulating function of the libtiff library and a CRC computation routine of MAD, an MPEG decoder. This looping pattern is handled precisely by the analysis. If $N \geqslant 8$, C4B infers the bound $\frac{N}{8}|[0,n]|$; but if $N < 8$, it infers $7\frac{8-N}{8} + \frac{N}{8}|[0,n]|$. The selection of the block size (8) and the cost in the second loop (`tick 1`) are arbitrary choices and C4B would also derive tight bounds for other values.

To understand the resource bound for the case $N < 8$, first note that the cost of the second loop is $|[0,n]|$. After the first loop, we still have $\frac{N}{8}|[0,n]|$ potential available from the invariant. So we have to raise the potential of $|[0,n]|$ from $\frac{N}{8}$ to 1, that is, we must pay $\frac{8-N}{8}|[0,n]|$. But since we got out of the first loop, we know that $n < 8$, so it is sound to only pay $7\frac{8-N}{8}$ potential units instead. This level of precision and compositionality is possible thanks to potential-based reasoning and only achieved by C4B. No other available tool derives the aforementioned tight bounds.

Example *t62* in Figure 4.6 (p. 88) is the inner loop of a Quicksort implementation in cBench. More precisely, it is the partitioning part of the algorithm. This partition loop has linear complexity, and C4B gives a worst-case bound of $2 + 3|[l,h]|$ ticks for it. This bound is not optimal but it can be refined by rewriting the program. To understand the bound, we can reason as follows. If $h \geqslant l$ initially, the cost of the loop is 2. Otherwise, the cost of each round (at most 3) can be payed using the potential of $[l,h]$ by the first increment to l because we know that $l < h$. The two inner loops can also use $[l,h]$ to pay for their inner costs. KoAT fails to find a bound and Loopus'14 derives the quadratic bound $(h-l-1)^2$. Following the classical technique, these tools try to find one ranking function for each loop and combine them multiplicatively or additively.

4.5 Experimental Evaluation

To evaluate the implementation of the interval system in the tool C4B, I compared it to competing analyses on a benchmark set of 33 challenging loop and recursion patterns from open-source code and the literature [GMC09, GJK09, GZ10].

Benchmark set. The benchmark set used for the evaluation focusses on loop bounds where the resource of interest is the number loop iterations during the program execution. There are also three examples with recursive procedures, one where only one loop is assigned a cost, and one where resources are freed. The last two examples can only be handled by C4B since no other tools provide fine-grained control of the cost model at the source level and no other tool can handle non-monotonic resources.

t09	**t19**	**t30**	**t15**	**t13**
`i = 1; j = 0`			`assume y>=0`	`while x>0:`
`while j<x:`	`while i>100:`	`while x>0:`	`while x>y:`	`  x = x-1`
`  j = j+1`	`  i = i-1; tick 1`	`  t = x`	`  x = x-(y+1)`	`  if random: y=y+1`
`  if i>=4:`	`i = i+k+50;`	`  x = y`	`  z=y`	`  else: while y>0:`
`    i=1; tick 40`	`while i>=0:`	`  y = t-1`	`  while z>0:`	`          y = y-1`
`  else: i = i+1`	`  i = i-1; tick 1`	`  tick 1`	`    z = z-1; tick 1`	`          tick 1`
`  tick 1`			`  tick 1`	`  tick 1`
$11\lvert[0,x]\rvert$	$50{+}\lvert[-1,i]\rvert{+}\lvert[0,k]\rvert$	$\lvert[0,x]\rvert{+}\lvert[0,y]\rvert$	$\lvert[0,x]\rvert$	$2\lvert[0,x]\rvert{+}\lvert[0,y]\rvert$

Figure 4.7: Example loop patterns with linear complexity from the benchmark set. The bounds inferred by C4B are shown on the last line.

I now describe five representative loop patterns from the benchmark set which are displayed in Figure 4.7. Example *t09* is a loop that performs an expensive operation every 4 steps. C4B is the only tool able to amortize this cost over the input parameter x. Example *t19* demonstrates the compositionality of the analysis. The program consists of two loops that decrement a variable i. In the first loop, i is decremented down to 100 and in the second loop i is decremented further down to -1. However, the variable i is changed by the assignment `i=i+k+50` between the two loops. So in total the program performs $52 + |[-1, i]| + |[0, k]|$ ticks. C4B finds this tight bound because the potential method naturally takes into account the relation between the two loops. Example *t30* decrements both input variables x and y down to zero in an unconventional way: the action of the loop body is equivalent to decrementing x and exchanging x and y. C4B infers the bound $|[0, x]| + |[0, y]|$. Sometimes we need some assumptions on the inputs in order to derive a bound. Example *t15* is such a case. I assume here that the input variable y is non-negative and write **`assume`** `y>=0`. The assignment `x=x-(y+1)` in the loop is best seen as `x=x-1` followed by `x=x-y`. Inside the loop body we know that $x > 0$, so we can obtain one unit of potential potential from `x=x-1`. Then we know that $x \geqslant y \geqslant 0$, as a consequence we can share the potential of $|[0, x]|$ between $|[0, x]|$ and $|[0, y]|$ after `x=x-y`, thus providing potential for the inner loop. Example *t13* shows how amortization can be used to find linear bounds for nested loops. The outer loop is iterated $|[0, x]|$ times. In the conditional, we either (the branching condition is arbitrary) increment the variable y or we execute an inner loop in which y is counted back to 0. C4B computes a tight bound.

Results of the evaluation. The experimental evaluation compares the performance of C4B, KoAT [BEF$^+$14], Loopus'14 [SZV14], Loopus'15 [SZV15], and CoFloCo [FM16]. I also contacted

Tool	**C4B**	CoFloCo	Loopus'15	Loopus'14	KoAT	SPEED
#bounds	32	27	23	20	20	14
#linear	32	27	22	20	18	14
#tested	33	33	33	33	33	14

Table 4.1: Results of the experimental evaluation.

the authors of SPEED but have not been able to obtain this tool. For this reason, the bounds inferred by SPEED are assumed to be what was reported by its authors.

Table 4.1 summarizes the results of the experiments. It shows for each tool the number of derived bounds (#bounds), the number of asymptotically tight bounds (#linear), and the number of examples that I was able to test with the tool (#tested). Even though recursive functions are supported by both CoFloCo and KoAT, only C4B could give bounds on the three examples involving recursion.

The results show that C4B outperforms the existing tools on the benchmark set. However, this experimental evaluation has to be taken with a grain of salt. The other tools complement C4B since they can derive polynomial bounds and some have better support for program involving memory operations. The next chapter presents a more flexible potential-based analysis that closes this gap.

Chapter 5

A Generic Automated Framework

5.1 Introduction

In this chapter, I will describe a generic framework to automate the bound inference using linear programming. This framework is the result of a careful inspection of the previous automatic systems based on the potential method. Ultimately, in Section 5.7, the two essential requirements that allow automation with linear programming are made explicit. Then, the whole automated framework is buildt on them. To prove the applicability of the new framework, it is implemented in a new tool called Pastis. Pastis derives polynomial bounds for programs represented with LLVM bitcode; it is described in Section 5.8. The practicality of Pastis is justified by a thorough experimental evaluation.

5.1.1 Low-Level Programs

In contrast to the beginning of my dissertation, the generic framework of this chapter works with programs represented as low-level control-flow graphs with procedure calls. Their formal definition is given in Section 5.2. This change of presentation is justified by two points. First, the new presentation makes the operational semantics and the reasoning on programs very simple; this lets me put the emphasis on the automated analysis itself. Second, using this input format proves that potential-based reasoning is suitable even on very unstructured languages, opening the door to automatically verifying the resource consumption of compiled programs (e.g., programs expressed in bytecode or assembly).

5.1.2 Improvements on the Interval System

In addition to improving our understanding of automated potential-based tools, the framework of this chapter addresses concrete limitations of the interval system introduced in Chapter 4:

- *Lack of user interaction.* When the C4B fails to find a bound, the user has two options. One is to rewrite the input programs to please C4B. The other is to extend C4B to accomodate for the program (and rework the soundness proof). The new generic framework naturally provides a third option, hints—called rewrite functions—can be provided as annotations to guide the analsysis.

- *Limited set of base functions.* The interval system is limited to linear potential. Thus, programs with polynomial resource bounds cannot be analyzed. The generic framework of this chapter provides a natural setting for a system supporting polynomial bounds (and possibly more); as demonstrated by the new tool Pastis.

- *Limited potential rewrites.* When faced with intricate programs, the Relax rule sometimes does not offer enough freedom to "rewrite" the potential functions. For instance, if $b \in [a, c]$, one would like to be able to rewrite the potential $|[a, c]|$ as $|[a, b]| + |[b, c]|$. The Relax rule could be refined, but every such refinement requires to rework its soundness proof. Rewrite functions, presented in Section 5.6, offer a solution to this extensibility problem.

- *Lack of proof certificates in the implementation.* In theory, C4B could generate proof certificates to justify the bounds it inferred; however, I found it difficult to do so. With the new framework, I was able to implement with very little code a mechanism to extract proof certificates. The small number of rules in the framework and the simplicity of the semantics of control-flow graphs were two key factors of this success.

In addition to all these points, all the bounds derived in the interval system can be found with a suitable instantiation of this chapter's framework: the generic framework is a strict generalization of the interval system.

5.1.3 The Essence of Previous Work

In implementations of automated amortized analysis, the main idea enabling the automation is to fix the shape of potential functions. Potential functions are chosen to be linear combinations of more elementary *base potential functions.* The coefficients multiplying the base functions are then inferred using linear programming. To be useful in practice, base potential functions must account for quantities of interest with respect to the resource consumption of the program. For instance, C4B uses the size of program variable intervals, and RAML uses polynomials of the sizes of data structures. But this requirement alone does not guarantee that automation is possible. Now, I list three essential items that enable automation. These items are somewhat informal, but later in this chapter, I formalize them to present the generic automation framework.

(1) Linear stability. Potential functions need to be linearly closed under ubiquitous program constructs. For example, when a program construct changes the state, the potential of the new state should be expressible as a linear transformation of the one of the previous state. For instance in RAML, when the head constructor of a list with potential $\binom{|\ell|}{2}$ is popped, the new potential can be exactly expressed as $|\ell| + \binom{|\ell|}{2}$. In C4B, when a variable a is incremented by one and $a < b$, the potential $|[a, b]|$ becomes $1 + |[a, b]|$.

(2) Linear weakening. Potential functions need a linear weakening mechanism. There needs to be a set of linear constraints on the coefficients of the base potential functions that implies $P_1 \geqslant P_2$ for two general potential functions P_1 and P_2. The weakening is necessary to account for conditionals, for instance. If multiple branches have different costs, they are still required to have the same potential at their meeting points. The weakening mechanism is what allows to drop excessive potential in branches with low resource consumption. In RAML and C4B, base potential functions are globally non-negative, hence, a sufficient condition to realize an inequality between two potentials is to constrain the coefficients of their base potential functions pointwise.

(3) Linear rewriting. Potential functions need a linear rewrite mechanism. The need for potential rewriting occurs in systems where the data is loosely structured (e.g., integer programs). In that case, it is common that the same potential has two or more useful representations. For instance if $a \leqslant b \leqslant c$, the potential $|[a, c]|$ can also be represented as $|[a, b]| + |[b, c]|$. The freedom to transform one from to the other is critical to obtain precise bounds. In C4B, the Relax rule is responsible for this potential rewrite. For instance it is able to move potential from an interval with a known size to the constant potential. RAML does not have an explicit rewriting mechanism but its sharing operation is one form of potential rewrite. If RAML were to include good support for integers, one rewriting mechanism similar to the Relax rule of C4B would be needed.

A note about weakening and rewriting. The difference between rewriting and weakening is sometimes blurred, for instance, in C4B, when the size of an interval $[a, b]$ is known to be greater than or equal to a constant K, it is sound to change the potential $2 \cdot |[a, b]|$ into $2 \cdot K$ since $2 \cdot |[a, b]| \geqslant 2 \cdot K$. This transformation is both a rewrite—$|[a, b]|$ is changed into K—and a weakening—in the case $|[a, b]| > K$. In the generic framework, we will use rewrite functions; they allow to describe both rewriting and weakening with the same tool.

5.2 Interprocedural Control-Flow Graphs

Programs are now represented as standard interprocedural control-flow graphs. As explained in Section 5.1.1, this choice does not tie the presentation of the analysis to a specific set of control structures and allows the technique to be applied to unstructured input programs, including ones written in bytecode and other low-level languages.

5.2.1 Syntax

A program is represented as a directed graph where nodes are *program points* from the set $\mathcal{L}$ and edges bear program *actions* from the set *Act*. Intuitively the program counter jumps from node to node following edges and updates the memory state according to the actions it encounters. A procedure is represented by a pair (ℓ_e, ℓ_x) where $\ell_e, \ell_x \in \mathcal{L}$ are respectively the entry and exit points of the procedure. The different possible actions are defined below. Like in the presentation of Chapter 2 base actions are left abstract.

$$\mathit{Act} := \; b \mid \mathsf{call}\ p \mid \mathsf{guard}\ C \mid \mathsf{weaken}$$

An action can be a generic base action b. It can also be a call $\mathsf{call}\ p$ to the procedure p. The arguments and return values are passed using global variables. A guard action $\mathsf{guard}\ C$ is used to represent conditional statements and block the execution if the runtime memory state is not in the condition set C. Finally, an action can be an explicit weakening hint weaken. Such a hint can be inserted by the user or by a pre-processing heuristic; it does not have any semantic effect but is used by the analysis to perform potential rewrites in a syntax-directed way.

Now, a program is defined as a pair (E, Δ) where:

— $E \subseteq (\mathcal{L} \times \mathit{Act} \times \mathcal{L})$ is the set of all edges in the program;

$$\frac{(\ell, b, \ell') \in E}{(\ell, \sigma) \to (\ell', [\![b]\!](\sigma))} \qquad \frac{(\ell, \mathsf{weaken}, \ell') \in E}{(\ell, \sigma) \to (\ell', \sigma)} \qquad \frac{(\ell, \mathsf{guard}\ C, \ell') \in E \qquad \sigma \in C}{(\ell, \sigma) \to (\ell', \sigma)}$$

$$\frac{(\ell, \mathsf{call}\ P, \ell') \in E \qquad \Delta(P) = (\ell_e, \ell_x) \qquad (\ell_e, (\sigma_0, \sigma)) \to^* (\ell_x, (\_, \sigma'))}{(\ell, (\sigma_l, \sigma)) \to (\ell', (\sigma_l, \sigma'))}$$

Figure 5.1: Mixed-step semantics of a program (E, Δ, G). Non-call steps use a classic small-step transition while calls are atomically executed using $\to^*$.

— and Δ is a map from procedure names to procedures.

For simplicity, we also require that there is at most one procedure entry point from which any point $\ell \in \mathcal{L}$ is reachable in the graph formed by the edges E. This guarantees that procedures have disjoint sets of points.

5.2.2 Semantics

An *execution state* is a pair (ℓ, σ) where $\sigma \in \Sigma = (\Sigma_l \times \Sigma_g)$ is a memory state composed of a local and a global part, and $\ell \in \mathcal{L}$ is the program counter. The two sets Σ_l and Σ_g are left abstract, but we require the presence of $\sigma_0 \in \Sigma_l$ to initialize the local variables upon entry of a procedure. The complete operational semantics of programs is given in Figure 5.1. The single-step execution of a program is defined as a transition relation on execution states. When evaluating an edge of the program, the change made to the memory state is determined by the action labelling the edge. For each base statement b, we assume that we have a corresponding semantic function $[\![b]\!] \in (\Sigma \to \Sigma)$ to perform the action b on an input state. This requirement forces base statements to be deterministic, but note that it is possible to encode non-determinism by having multiple edges of the control-flow graph leaving the same program point. I used $\to^*$ to denote the reflexive transitive closure of the step relation $\to$. Note how, on return of a procedure call, the local state of the caller σ_l is restored from the memory state before the call.

5.3 Reasoning on Control-Flow Graphs

In the same way that Chapter 2 gave a general principle to reason on structured commands, I will now describe how to prove properties about control-flow graphs with procedure calls.

Procedure annotations for programs without calls. For a single procedure, a *procedure annotation* P is defined as a map from program points in $\mathcal{L}$ to sets of memory states in $\mathcal{P}(\Sigma)$. A procedure annotation is *sound* when for any execution $(\ell_1, \sigma_1) \to \ldots \to (\ell_n, \sigma_n)$ in the annotated procedure, if $\sigma_1 \in P(\ell_1)$ then $\sigma_n \in P(\ell_n)$.

When the procedure considered does not contain any calls, all annotations that are preserved by the edges of the procedure are sound. By "preserved by the edges of the procedure," I mean that for any edge (ℓ, a, ℓ') and any σ in $P(\ell)$, all the states that can result from the action of a on σ are in $P(\ell')$. The actions *Act* defined in the previous section are all deterministic, so there is at most one state in the image of an action—at most, because guard statements could block the execution. We can now build on this simple criterion for regular actions to reason on programs with procedure calls and recursion.

Programs with procedure calls. The notions of procedure annotation and sound procedure annotation defined in the previous paragraph still make sense in the context of programs with procedure calls. But when a control-flow graph contains procedure calls, what should be the condition respected by the call edges? Like in Chapter 2, the answer to this question requires to define what procedure specifications are. With procedure specifications, the source and destination annotations of any edge with action call p will have to be consequences of the procedure specifications of the callee p.

A *procedure specification* $[\forall z \in Z. P_z]$ is a couple of a set Z and a procedure annotation P_z that depends on a variable $z \in Z$. The set Z is, like in Chapter 2, a set of auxiliary state. In practice, it is used to relate the procedure annotation P_z at the entry and exit points of the procedure. Finally, an *interprocedural annotation* (IA) is defined as a mapping from procedure names to sets of procedure specifications. We are now ready to define what an admissible IA is.

Definition 2 (Admissible IA). *An IA Λ for a program (E, Δ) is admissible when for any procedure name p, any procedure specification $[\forall z \in Z. P_z] \in \Lambda(p)$, any auxiliary state $z_0 \in Z$, and any edge $(\ell, a, \ell') \in \Delta(p)$*

1. *if a is a base action b, then for any memory state σ, $\sigma \in P_{z_0}(\ell) \implies [\![b]\!](\sigma) \in P_{z_0}(\ell')$;*

2. *if a is* guard C*, then* $P_{z_0}(\ell) \cap C \subseteq P_{z_0}(\ell')$;

3. *if a is* weaken*, then* $P_{z_0}(\ell) \subseteq P_{z_0}(\ell')$*; and*

4. *if a is* call q *and* $\Delta(q) = (\ell_e, \ell_x)$*, then for any local memory state* σ_l*, and any two global memory states* σ *and* σ' *that are valid for the callee's specifications—that is,*

$$[\forall z \in Z.\, Q_z] \in \Lambda(q) \implies \big(\forall z \in Z.\, (\sigma_0, \sigma) \in Q_z(\ell_e) \Longrightarrow (\_, \sigma') \in Q_z(\ell_x)\big)$$

—we have $(\sigma_l, \sigma) \in P_{z_0}(\ell) \Longrightarrow (\sigma_l, \sigma') \in P_{z_0}(\ell')$.

The admissibility conditions give a practical reasoning principle for interprocedural control-flow graphs. The practicality lies in the fact that these conditions are local—they only constrain the two ends of one edge—but they imply global invariants. The following proposition formally states that any admissible annotation is also sound.

Proposition 1 (Admissible IAs are sound)**.** *Let* Λ *be an admissible IA for the program* (E, Δ)*, then for any procedure p, any procedure specification* $[\forall z \in Z.\, P_z] \in \Lambda(p)$*, any auxiliary state* $z_0 \in Z$*, and any execution trace* $(\ell_1, \sigma_1) \to \ldots \to (\ell_n, \sigma_n)$*, we have* $\sigma_1 \in P_{z_0}(\ell_1) \implies \sigma_n \in P_{z_0}(\ell_n)$.

Proof. (Checked in Coq.) By induction on the operational semantics. □

5.4 Size-Change Analysis

From now on I will make an important change to the problem we want solve: instead of focusing on proving resource safety, we will now aim to *bound the a the value of a goal function at end of a given procedure.* This new objective simplifies the presentation and, as I will show in Section 5.4.3, is equivalent to the resource-bound problem of the previous chapters.

5.4.1 Problem Statement

We assume that any program $P := (E, \Delta)$ has a distinguished "main" procedure m, so that a complete execution of P is an execution starting from the point ℓ_e and terminating at the point ℓ_x, if $\Delta(m) = (\ell_e, \ell_x)$. We call ℓ_e and ℓ_x the entry and exit points of the program P. My problem statement is then:

Let $g \in (\Sigma \to \mathbb{Q})$ be a *potential goal* that maps memory states to rational numbers. We desire to find a bound $\Phi \in (\Sigma \to \mathbb{Q})$ such that any complete execution of the program starting with memory state σ_e and terminating with memory state σ_x satisfies $\Phi(\sigma_e) \geqslant g(\sigma_x)$.

In the problem statement, Φ is an upper bound on the potential goal g. However, a solution to this problem also provides a way to obtain lower bounds. Indeed, if Φ is an upper bound on the goal $-g$, then $-\Phi$ is a lower bound on the goal g. Moreover, a "good" upper bound for $-g$ provides a "good" lower bound on g. Even though this observation is trivial, the work I present in this chapter is the first potential-based technique that can infer lower bounds as is.

5.4.2 Potential for Size Change

Logical contexts. Like in Chapter 4, we are going to split the bound inference from the abstract-interpretation reasoning. The inference of invariants on the memory states using abstract interpretation is beyond the scope of my dissertation, thus, throughout the rest of this chapter, I will assume that we are provided with a set of allowed memory states $S_\ell \subseteq \Sigma$ for each program point ℓ. These sets are called logical contexts.

To prove the soundness of the analysis, we additionally require that logical contexts (S_ℓ) for a program (E, Δ) satisfy the following conditions.

1. If the program contains an edge (ℓ, a, ℓ') where a is not a procedure call, then for any memory state σ in S_ℓ, any state σ' resulting from executing the action a on σ must be in $S_{\ell'}$.
2. If the program contains an edge $(\ell, \mathsf{call}\ p, \ell')$ and $\Delta(p) = (\ell_e, \ell_x)$, then there is a set of local states L such that $S_\ell = \{(\sigma_l, \sigma_g) \mid \sigma_l \in L \wedge (\sigma_0, \sigma_g) \in S_{\ell_e}\}$ and $S_{\ell'} = \{(\sigma_l, \sigma_g) \mid \sigma_l \in L \wedge \exists \sigma_l'.\, (\sigma_l', \sigma_g) \in S_{\ell_x}\}$.

Overview of the technique. The bound we are now looking for is on a goal $g \in (\Sigma \to \mathbb{Q})$ evaluated in the final state of the program—i.e., the exit point of the main procedure.

Now, I introduce the core reasoning principle on a control-flow graph without procedure calls. The entry and exit points of the program are, respectively, ℓ_e and ℓ_x. To bound the value of the goal g in the final state, we are going to associate a potential function $\Phi_\ell \in (\Sigma \to \mathbb{Q})$ with each point ℓ in the program. The potential of the exit point of the program Φ_{ℓ_x} is set to be the potential goal g, and the other potential functions also have to respect some constraints derived from the program structure. These constraints enforce that on any execution trace

$$\epsilon := (\ell_1, \sigma_1) \to (\ell_2, \sigma_2) \to \ldots \to (\ell_n, \sigma_n)$$

where σ_1 is valid—that is, $\sigma_1 \in S_{\ell_1}$—we have $\Phi_{\ell_1}(\sigma_1) \geqslant \Phi_{\ell_2}(\sigma_2) \geqslant \cdots \geqslant \Phi_{\ell_n}(\sigma_n)$. When the trace ϵ is a complete program execution—that is, $\ell_1 = \ell_e$ and $\ell_n = \ell_x$—by transitivity of $\geqslant$, we get that $\Phi_{\ell_e}(\sigma_1) \geqslant g(\sigma_n)$. So Φ_{ℓ_e} is a bound on the goal g evaluated in the final state.

With procedure calls, the essence of the idea is similar. In fact if there are no recursive calls, procedures can be inlined and the previous paragraph also applies. Recursive procedures are what complicates the constraints. In particular, how should the potential of the exit point of a recursive procedure be constrained? Section 5.5 gives a formal answer to this question by defining an admissiblity criterion for potential annotations on complete programs.

5.4.3 Size Change and Resource Bounds

In Section 4.2.3, I subverted a quantitative logic to not only prove resource bounds but also size-change properties. Reciprocally, this section is dedicated to explaining how a size-change analysis can be subverted to prove resource-bound properties.

Resources that cannot be freed. So far, I always split the program state into two parts, a memory state and a resource counter c. Resource-safe programs were then defined as programs respecting the invariant that the resource counter is always non-negative. From now on assume that the resource of interest cannot be freed. In that case, it is sufficient to check that the resource counter at the end of program is non-negative, because this is the point where it is maximal. Now, another way to view the resource-bound problem is to have a special program variable z accounting for the amount of resource that was consumed by the execution. With such a variable, the problem of finding the resource consumption is a simple matter of bounding the final value z_x of the variable z. And in fact, assuming z starts initialized to 0, the final value c_x of the resource counter c is related its initial value c_e by the equation

$$c_x = c_e - z_x.$$

Thus, any program starting with $c_e \geqslant z_x$ resource units is safe, because this implies that $c_x = c_e - z_x \geqslant 0$. In summary, we just observed that, unsurprisingly, the resource-safety of a program is strongly related to the size-change of a variable accounting for the amount of resource in use at a given time.

Resources that can be freed. The previous paragraph shows that the resource-bound problem can be solved by a mere size-change analysis when resources cannot be freed. But in general, resources can be freed; and resource safety is a safety property that must be checked at all steps taken by the program because it is related to the peak usage, not to the final usage. If a program allocates 3 integers, then frees them, a valid bound for the resource counter in the new size-change setting is 0, because no memory is in use at the end of the execution. However, the peak consumption of this program is 3.

Like in the previous paragraph, assume a variable z tracks the resource consumption. Allocation grows this counter variable and freeing lessens it. Additionally, assume the program of interest does not contain procedure calls. Like in Section 5.4.2, we associate a potential function Φ_ℓ to each program point. If in addition to the constraints the potential functions must satisfy, we require that for any location ℓ and any valid memory state at that point σ,

$$\Phi_\ell(\sigma) \geqslant \sigma(z). \tag{$\star$}$$

Then, the potential function of the initial program point Φ_{ℓ_e} is a bound on the peak value of the resource counter z.

To see this, consider an execution trace $\epsilon := (\ell_1, \sigma_1) \to (\ell_2, \sigma_2) \to \ldots \to (\ell_n, \sigma_n)$ where σ_1 is a valid memory state at ℓ_1, we are going to prove inductively on the suffixes of the trace that $\Phi_{\ell_1}(\sigma_1) \geqslant \max_{i \geqslant 1} \sigma_i(z)$.

— If the trace is a single execution state (ℓ_n, σ_n), then $\Phi_{\ell_n}(\sigma_n) \geqslant \sigma_n(z) = \max_{i \geqslant n} \sigma_i(z)$ by the condition $(\star)$.

— Otherwise the trace is $\epsilon = (\ell_k, \sigma_k) \to (\ell_{k+1}, \sigma_{k+1}) \to \ldots$ and by induction hypothesis $\Phi_{\ell_{k+1}}(\sigma_{k+1}) \geqslant \max_{i \geqslant k+1} \sigma_i(z)$. By the constraints of potential functions from Section 5.4.2, $\Phi_{\ell_k}(\sigma_k) \geqslant \Phi_{\ell_{k+1}}(\sigma_{k+1})$, and thus $\Phi_{\ell_k}(\sigma_k) \geqslant \max_{i \geqslant k+1} \sigma_i(z)$. But by the condition $(\star)$, we also have $\Phi_{\ell_k}(\sigma_k) \geqslant \sigma_k(z)$, and thus $\Phi_{\ell_k}(\sigma_k) \geqslant \max_{i \geqslant k} \sigma_i(z)$.

Note that the condition $(\star)$ is *local* but provides a *global* peak bound. In a practical implementation, the condition ($\star$) can be enforced naturally using the framework I describe in Section 5.6. The non-negativity requirement in the various quantitative logics of previous chapters is exactly the condition ($\star$), since the potential functions in these logics are essentially $\Phi - z$ in this chapter.

Another way to obtain a bound on the peak usage is to simply use a new program variable to track it. Interestingly, it is not clear to me how to relate the potential annotations bounding such a counter to the ones that satisfy the condition ($\star$). In practice, using the condition ($\star$) has the significant advantage of not complicating the control-flow of the input program: indeed, if one uses a peak counter, every increment to the real resource counter is followed by a check to see if the peak counter needs to be updated.

5.5 Interprocedural Potential Annotations

A *potential annotation* Φ associates a potential function Φ_ℓ with each program point ℓ in a procedure. An *interprocedural potential annotation* (IPA) Γ maps each procedure name p to a set of procedure annotations $\Gamma(p)$.

An IPA can be used to derive size-change bounds when it respects a set of admissibility conditions. These conditions are listed in the definition below.

Definition 3 (Admissible IPA)**.** *An IPA Γ for the program (E, Δ) is **admissible** when for every procedure p, edge $(\ell, a, \ell') \in E$, annotation $\Phi \in \Gamma(p)$, and valid state $\sigma \in S_\ell$:*

(A1) if a is a base action b, then $\Phi_\ell(\sigma) = \Phi_{\ell'}([\![b]\!](\sigma))$;

(A2) if a is guard C, then $\Phi_\ell(\sigma) = \Phi_{\ell'}(\sigma)$;

(A3) if a is weaken, then $\Phi_\ell(\sigma) \geqslant \Phi_{\ell'}(\sigma)$;

(A4) if a is call q and $\Delta(q) = (\ell_e, \ell_x)$, then there exists a potential function Ψ that depends only on the local memory state and a family of non-negative rational numbers (k_Φ) such that $\Phi_\ell = \Psi + \sum_{\Phi' \in \Gamma(q)} k_{\Phi'} \cdot \Phi'_{\ell_e}$ and $\Phi_{\ell'} = \Psi + \sum_{\Phi' \in \Gamma(q)} k_{\Phi'} \cdot \Phi'_{\ell_x}$.

Additionally, all potential functions attached to the entry and exit points of procedures are required to depend only on the global memory state.

The condition $\sigma \in S_\ell$ in Definition 3 ensures that the inequalities are required to hold only for states that can actually appear in an execution trace. The local nature of these conditions makes the generation of constraints described in Section 5.7 a local and compositional process.

As explained earlier, admissible potential annotations entail size-change invariants. The invariants obtained are described formally in the soundness Proposition 2.

Proposition 2. *For every valid program state σ at ℓ ($\sigma \in S_\ell$), if Γ is an admissible IPA and (ℓ', σ') is such that $(\ell, \sigma) \to^* (\ell', \sigma')$, then for all $\Phi \in \Gamma(p)$, $\Phi_\ell(\sigma) \geqslant \Phi_{\ell'}(\sigma')$.*

Using Proposition 2, if one wants to find a bound for the final value of a goal potential function g, it is enough to find an admissible IPA Γ with an annotation for the main procedure $\Phi \in \Gamma(m)$ such that $\Phi_{\ell_x} = g$ (where $\Delta(m) = (\ell_e, \ell_x)$). Then, for any complete program execution $(\ell_e, \sigma_e) \to^* (\ell_x, \sigma_x)$, we have $\Phi_{\ell_e}(\sigma_e) \geqslant \Phi_{\ell_x}(\sigma_x) = g(\sigma_x)$, and Φ_{ℓ_e} is a bound for the goal g. Thus, the rest of this chapter is dedicated to infering such an admissible IPA automatically.

Proof of Proposition 2. The idea of the proof is to create, from the admissible IPA Γ, an admissible IA Λ and leverage the soundness Proposition 1. In Λ, each potential annotation Φ_ℓ from Γ is interpreted as the predicate

$$P_{z\,\ell} := \{\sigma \mid \sigma \in S_\ell \wedge \Phi_\ell(\sigma) \leqslant z\}$$

where S_ℓ is the logical context at the program point ℓ, and $z \in \mathbb{Q}$ is an arbitrary auxiliary variable. The set of all the predicates P_z that results from interpreting all the potential functions Φ with the same auxiliary variable z yields the procedure annotation $[\forall z \in \mathbb{Q}.\, P_z]$. Finally, the collection of all the translated procedure annotations forms the IA Λ that I will now prove admissible according to Definition 2.

Let $[\forall z \in \mathbb{Q}.\, P_z] \in \Lambda(p)$ be a translated procedure annotation for p, $z_0 \in \mathbb{Q}$ an auxiliary state, and $(\ell, a, \ell') \in \Delta(p)$ an edge.

• If a is a base action b. Let $\sigma \in P_{z_0\,\ell}$, that is, $\sigma \in S_\ell$ and $\Phi_\ell(\sigma) \leqslant z_0$. We wish to prove that $[\![b]\!](\sigma) \in P_{z_0\,\ell'}$, that is (i) $[\![b]\!](\sigma) \in S_{\ell'}$ and (ii) $\Phi_{\ell'}([\![b]\!](\sigma)) \leqslant z_0$. The first condition we put on logical contexts in Section 5.4.2 proves (i). To prove (ii), we observe that, using the admissibility condition (A1), $\Phi_{\ell'}([\![b]\!](\sigma)) = \Phi_\ell(\sigma) \leqslant z_0$.

• If a is guard C. The admissibility condition is checked like in the previous case using the condition on logical contexts and the admissibility condition (A2) for potential annotations.

• If a is weaken. The reasoning is again similar to the two previous ones. To show $\Phi_{\ell'}(\sigma) \leqslant z_0$, we use the transitivity of $\leqslant$ and observe that condtion (A3) implies $\Phi_{\ell'}(\sigma) \leqslant \Phi_\ell(\sigma) \leqslant z_0$.

• If a is $\mathsf{call}\ q$. Let ℓ_e and ℓ_x be the entry and exit point of q, respectively. Assume that $\sigma = (\sigma_l, \sigma_g) \in P_{z_0\,\ell}$, we wish to prove that $\sigma' = (\sigma_l, \sigma'_g) \in P_{z_0\,\ell'}$ knowing that σ_g and σ'_g "match" the procedure specification of the callee q, that is,

$$[\forall z \in \mathbb{Q}.\, P_z] \in \Lambda(q) \implies \forall z \in \mathbb{Q}.\, \big((\sigma_0, \sigma_g) \in P_{z\,\ell_e} \implies (\_, \sigma'_g) \in P_{z\,\ell_x}\big). \tag{5.1}$$

Let us first see what this formula implies. Let $\Phi \in \Gamma(q)$ a potential annotation for q. Apply the formula (5.1) to the procedure annotation corresponding to Φ with $z = \Phi_{\ell_e}(\sigma_0, \sigma_g)$. After simplification, we obtain that

$$(\sigma_0, \sigma_g) \in S_{\ell_e} \implies (\_, \sigma'_g) \in S_{\ell_x} \wedge \Phi_{\ell_x}(\_, \sigma'_g) \leqslant \Phi_{\ell_e}(\sigma_0, \sigma_g).$$

But by the conditions we put on logical contexts around procedure calls in Section 5.4.2, we know that $(\sigma_0, \sigma_g) \in S_{\ell_e}$, so the conclusion of the above implication is true and *(i)* $(\_, \sigma'_g) \in S_{\ell_x}$ and *(ii)* $\Phi_{\ell_x}(\_, \sigma'_g) \leqslant \Phi_{\ell_e}(\sigma_0, \sigma_g)$.

We now prove that $\sigma' = (\sigma_l, \sigma'_g) \in S_{\ell'}$. By the condition we required on logical contexts around procedure calls in Section 5.4.2, we know that the exists a set L such that $\sigma_l \in L$ and that $\sigma' \in S_{\ell'}$ if $\sigma_l \in L$ and there is a local state σ'_l such that $(\sigma'_l, \sigma'_g) \in S_{\ell_x}$. This is exactly what *(i)* above proves. (To be precise, we need to know that there is at least one potential annotation in $\Gamma(q)$, but this can be satisfied by observing that the potential annotation associating $(\sigma \mapsto 0)$ with all locations can always be added to an admissible IPA.)

Finally, we show that $\Phi_{\ell'}(\sigma') \leqslant z_0$. Let Ψ and (k_Φ) be the potential function and non-negative coefficients from the admissibility condition (A4), then

$$\Phi_{\ell'}(\sigma') = \Psi(\sigma_l, \sigma'_g) + \sum_{\Phi' \in \Gamma(q)} k_{\Phi'} \cdot \Phi'_{\ell_x}(\sigma_l, \sigma'_g) \tag{5.2}$$

$$= \Psi(\sigma_l, \sigma_g) + \sum_{\Phi' \in \Gamma(q)} k_{\Phi'} \cdot \Phi'_{\ell_x}(\_, \sigma'_g) \tag{5.3}$$

$$\leqslant \Psi(\sigma_l, \sigma_g) + \sum_{\Phi' \in \Gamma(q)} k_{\Phi'} \cdot \Phi'_{\ell_e}(\sigma_0, \sigma_g) \tag{5.4}$$

$$\leqslant \Psi(\sigma_l, \sigma_g) + \sum_{\Phi' \in \Gamma(q)} k_{\Phi'} \cdot \Phi'_{\ell_e}(\sigma_l, \sigma_g) \tag{5.5}$$

$$= \Phi_\ell(\sigma) \leqslant z_0. \tag{5.6}$$

In step (5.2), we used the definition of σ' and the constraint on $\Phi_{\ell'}$ from the admissibility condition (A4). In step (5.3), we used the fact that annotations of exit points only depend on the global state and Ψ only depends on local state. In step (5.4), we used (*ii*) above and the fact that $k_{\Phi'} \geqslant 0$. In step (5.5), we used that annotations of entry points only depend on the global state. And finally in step (5.6) we used the constraint on Φ_ℓ and the hypothesis $\Phi_\ell(\sigma) \leqslant z_0$.

This concludes the proof that the IA Λ is admissible when Γ is admissible.

We can now finish the proof of Proposition 2. Let $\epsilon := (\ell, \sigma) \to^* (\ell', \sigma')$ be an execution trace where $\sigma \in S_\ell$ and let $\Phi \in \Gamma(p)$. We now apply Proposition 1 on Λ with the trace ϵ and $z = \Phi_\ell(\sigma)$. This proves that $\sigma \in S_\ell \implies \sigma' \in S_{\ell'} \wedge \Phi_{\ell'}(\sigma') \leqslant \Phi_\ell(\sigma)$, and thus $\Phi_\ell(\sigma) \geqslant \Phi_{\ell'}(\sigma')$. $\square$

5.6 Rewrite Functions

Because of admissibility condition (A3), any automated analysis using the potential method needs to enforce *weakening* conditions of the form $\Phi \geqslant \Phi'$ on potential functions. In this section, I present a new principled approach to weakening in automated potential-based systems: *rewrite functions*. They subsume all the existing potential-weakening and potential-rewriting mechanisms. Moreover, they provide a language for a user to interact with the automated potential-based system.

Definition 4 (Rewrite function)**.** *The function $F_\ell \in (\Sigma \to \mathbb{Q})$ is a rewrite function at a program point ℓ when for any valid state $\sigma \in S_\ell$, $F_\ell(\sigma) \geqslant 0$.*

In other words, F_ℓ is the left-hand side of a program invariant $F_\ell(\sigma) \geqslant 0$.

Example. Assume that the potential function at program point ℓ is $\Phi := 2y + z$, and we are looking for a weakening $\Phi' := k' + k'_x \cdot x + k'_y \cdot y + k'_z \cdot z$ such that $\Phi \geqslant \Phi'$ holds. We will assume that nothing is known about the sign of variables x, y, and z, and thus pointwise constraints $2 \geqslant k'_y$, $1 \geqslant k'_z$, $0 \geqslant k'_x$, and $0 \geqslant k'$ would not ensure that $\Phi \geqslant \Phi'$ holds.

Now assume that $y \geqslant z + x$ and $z \geqslant x$ are invariants that hold at ℓ, which means that we have two rewrite functions $F_1 := -x + y - z$ and $F_2 := -x + z$. Write Φ as follows:

$$\Phi = 2y + z - (0 \cdot F_1 + 0 \cdot F_2). \tag{5.7}$$

Because F_1 and F_2 are rewrite functions at ℓ we have that $-x+y-z \geqslant 0$ and $-x+z \geqslant 0$, respectively. We obtain a value that is less than or equal to Φ by choosing any positive coefficients to replace either/both of the 0s in Equation (5.7). For instance, by choosing both coefficients to be 2, we have

$$\Phi = 2y + z - (0 \cdot F_1 + 0 \cdot F_2) \geqslant 2y + z - (2 \cdot F_1 + 2 \cdot F_2) = 4x + z,$$

and thus we can choose Φ' to be $4x + z$. (Rather than making specific choices for the coefficients $\{u_i\}$, however, we will leave it to the linear-programming solver to choose values that allow it to solve the overall constraint system generated for the program.)

In short, we can systematize potential weakening as a two-step process.

1. If the original potential at ℓ is V, write the weakened potential as $V - \sum u_i \cdot F_i$, where $\{F_i\}$ is the set of rewrite functions available at ℓ. (In the example, $V = 2y + z$.)

2. Choose values for the coefficients $\{u_i\}$ such that $u_i \geqslant 0$, for all i. (In the example, $u_1 = 2 \geqslant 0$, and $u_2 = 2 \geqslant 0$.)

The strength of this process is that it can be fully expressed using standard linear algebra, and thus, fed as-is to a linear-programming solver. Let the coefficients of Φ (i.e., $\vec{k}$), Φ' ($\vec{k}'$), and $\vec{u}$ be column vectors, and let the matrix F represent the set $\{F_i\}$ of rewrite functions, with each F_i a column of F. The constraints generated to express the allowable rewrites of Φ into Φ' as per items (1) and (2) are

$$(\vec{k}' = \vec{k} - F\vec{u}) \wedge (\vec{u} \geqslant 0). \tag{5.8}$$

In the example above, we have

$$\begin{pmatrix} k' \\ k'_x \\ k'_y \\ k'_z \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 2 \\ 1 \end{pmatrix} - \begin{pmatrix} 0 & 0 \\ -1 & -1 \\ 1 & 0 \\ -1 & 1 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} \wedge \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} \geqslant \begin{pmatrix} 0 \\ 0 \end{pmatrix}.$$

There are many solutions to this system. The linear-programming solver is free to pick any one that allows the overall system of constraints to be satisfied.

Rewrite idempotence. An interesting consequence of the algebraic formulation of weakenings introduced above is that, for a fixed set of rewrite functions, multiple compositions of the linear weakening system are never necessary. In practice, this property means that the automated system never needs to apply a weakening operation twice consecutively: once is always sufficient.

We call the set of coefficients for the initial and final potential in a satisfying assignment of a linear weakening system a *solution.* Then the following property holds:

Proposition 3 (Rewrite idempotence)**.** *The set of solutions for one weakening application and the set of solutions for two or more composed weakening applications are identical.*

Rewrite hints. In practice, a system needs a source of rewrite functions to use at the different points in the program. Because rewrite functions are obtained from program invariants, abstract interpretation [CC77]—using various abstract domains—can be used to obtain rewrite functions for the different points in the program: given such a source of invariants, a system would employ heuristics to choose which invariants should be used as rewrite functions in the constraint system passed to the linar-programming solver.

Occasionally, however, a program requires a complex transfer of potential. With previous implementations of amortized analysis, one was faced with two choices: either rewrite the program, or modify the analysis. Both alternatives have drawbacks.

— Rewriting the program provides only an indirect means for obtaining the desired effect, and it can be hard to understand whether a given program rewriting will allow the analyzer to establish the desired bound.

— Modifying the analysis to enable more complex transfers of potential requires the whole soundness proof to be redone as well as a good knowledge of the implementation of the analyzer.

Rewrite functions offer a third option. A programmer can manually specify rewrite functions as hints to be used to analyze the complex parts of a program. These hints, in contrast with typical assertions, both have no runtime effect and do not compromise soundness. In particular, before using a rewrite function, the analyzer would ask an oracle (e.g., an SMT solver or an abstract interpreter) if the value of the user-supplied rewrite function is provably non-negative at that program point. This approach provides a good fallback mechanism in case the heuristics of the analyzer are not sophisticated enough to identify an appropriate set of rewrite functions.

5.7 Automatic Potential Inference

In this section, I present the general framework to automate the inference of potential. To emphasize the modularity of the method, I left unspecified the base actions and the base potential functions

$(b_i)_{1 \leqslant i \leqslant N}$. However, to ensure that automation is possible, two formal requirements must be satisfied.

Requirement 1: Basis stability. Ideally, we would like the set of base potential functions to be linearly stable under all base actions. However, this requirement is too strong in practice. As an escape hatch, for a base action b, we require an *exclusion set* X_b containing all the base functions that are not expressible as linear combinations of the basis after the action of b. Thus, if $b_k \notin X_b$, then there is a family of coefficients (k_i) such that $b_k(\llbracket b \rrbracket(\sigma)) = \sum_i k_i \cdot b_i(\sigma)$. In a nutshell, the first requirement asks that for all base actions b, all base function except for the ones in the exclusion set X_b are left linearly stable under the action of b.

As an example, we show that the requirement is met when the base action is an increment $v \leftarrow u + c$ where $c \in \mathbb{Z}$, the base functions $(b_i)_i$ are all the monomials over program variables of total degree d or less, and $X_{v \leftarrow u+c} = \varnothing$ for all increments. In that case, if the maximum power of v in b is v^n (with $n \leqslant d$), then

$$b \circ \llbracket v \leftarrow u + c \rrbracket = \sum_{0 \leqslant j \leqslant n} \binom{n}{j} c^{n-j} \frac{u^j b}{v^n}.$$

In this sum, the products $\binom{n}{j} c^{n-j}$ are the constant coefficients (k_i) above and the multiplicands $u^j b / v^n$ are base functions. Indeed, they are monomials over program variables of degree $n \leqslant d$. Note that when v does not appear in b, n is 0 and the above sum correctly degenerates to b.

The exclusion set enables the implementation of practical tools. For example, during pre-processing it is often desirable to abstract a statement into one or more non-deterministic assignments "$v \leftarrow \star$". These assignments cause any base function that depends on the assigned variable to not be linearly stable. This situation leads us to define $X_{v \leftarrow \star} = \{b_i \mid v \in b_i\}$, where I write $v \in b_i$ to express that b_i depends on v.

Requirement 2: Rewrite functions. We also assume that we are provided with a set of rewrite functions for every program point. Recall that, by Definition 4, for any program point ℓ and any program state σ valid at that point, such a rewrite function F_ℓ satisfies $F_\ell(\sigma) \geqslant 0$.

5.7.1 Generating a Linear System

The goal of the automation procedure is to find an admissible IPA without user interaction. Additionally, one potential annotation Φ for the main procedure of the program must have its final potential annotation set to a goal g specified by the user. If the automation sucessfully finds such an admissible IPA, the annotation of the entry point in Φ is a bound on the goal, and is reported to the user.

Overview. The search for the admissible IPA is split in two phases. A “template” admissible IPA is first generated, where all potential annotations are linear combinations of base functions with coefficients in a set LP of linear-programming variables. Together with this template IPA comes a set of constraints that encodes a sufficient condition for the IPA to be admissible. The constraints are then fed to a linear-programming solver that either successfully finds a solution or returns with an error. A solution yields directly the admissible IPA we are looking for and, in particular, the bound on the potential goal.

Procedure annotations. The IPAs of Section 5.5 allow to have multiple annotations for a single procedure and the admissibility conditon (A4) for IPAs permits the use of multiple annotations for an edge with a call action. This freedom corresponds to many design decisions that must be taken by the implementer of an automated system. These choices, however, are largely independent from the linear constraints that are generated, so I will leave them unspecified in the next paragraph. In Section 5.8, I describe and justify the choices made in Pastis, my implementation of the framework.

Constraints generated for the program edges. The potential function associated with each program point of each procedure is a template potential function $\Phi_\ell(\sigma) = \sum_i k_i \cdot b_i(\sigma)$, where each k_i belongs to LP, and $(b_i)_{1\leqslant i\leqslant N}$ is the family of base functions. For each program edge (ℓ, a, ℓ'), we constrain the coefficients $(k_i)_i$ and $(k'_i)_i$ of Φ_ℓ and $\Phi_{\ell'}$ by case analysis on the kind of the action a. To be able to use matrix notation, we define the column vectors $\vec{k} := (k_1, \dots, k_N)^\intercal$ and $\vec{k'} := (k'_1, \dots, k'_N)^\intercal$. Each set of constraints comes with a justification that it encodes a linear system making the IPA admissible as per Definition 3.

• **Case** b. Because of Requirement 1, the action of b on a base function $b_i \notin X_b$ is a linear operation. Without loss of generality, assume that the exclusion set X_b is $\{b_i \mid N_X < i \leqslant N\}$. The constraints generated are

$$(\vec{k} = Q\vec{k'}) \wedge \bigwedge_{N_X < i \leqslant N} k'_i = 0.$$

The $N\times N$ matrix Q contains zeroes everywhere but in the first N_X columns. The j^{th} column ($j \leqslant N_X$) is the result of the composition $b_j \circ [\![b]\!]$; that is: $b_j([\![b]\!](\sigma)) = \sum_i q_{i,j} \cdot b_i(\sigma)$ for any memory state σ. The expansion exists because $b_j \notin X_{v \leftarrow e}$ when $j \leqslant N_X$. The coefficients $(q_{i,j})$ are constants known before the generation of the linear program, and only depend on the choice of the basis and the base action.

We now justify why this constraint set enforces the admissiblity condition for edges with base actions. The function $\Phi_{\ell'} \circ [\![b]\!]$ is constrained to be linearly expressible in the basis by setting the coefficients of base functions in the exclusion set to zero. This linear transformation on $\Phi_{\ell'}$ is then precisely specified as the matrix multiplication $\vec{k} = Q\vec{k'}$. Thus, for all states σ, $\Phi_\ell(\sigma) = (\Phi_{\ell'} \circ [\![b]\!])(\sigma) = \Phi_{\ell'}([\![b]\!](\sigma))$ and the admissibility condition (A1) is satisfied.

• **Case guard** e**.** We require that $\vec{k'} = \vec{k}$, and the admissibility condition (A2) is trivially satisfied.

• **Case weaken.** To encode a weakening via a set of linear constraints, we make use of the rewrite functions provided in Requirement 2. As explained in Section 5.6, we relate the potential functions Φ_ℓ and $\Phi_{\ell'}$ using an ℓ-specific set of unknowns, $\vec{u}_\ell$, as coefficients in a linear combination of the rewrite functions available at ℓ. Following Equation (5.8), let F_ℓ denote the matrix of rewrite functions available at ℓ (in which the i^{th} column of F is the i^{th} rewrite function F_i). for the (unknown) coefficients of the potential functions In matrix notation, the constraints generated are

$$(\vec{k'} = \vec{k} - F_\ell \vec{u}_\ell) \wedge (\vec{u} \geqslant 0).$$

The fact that these constraints enforce the admissibility condition (A3) was already discussed in Section 5.6.

• **Case call** p**.** When calling a procedure, we need to know what base functions depend only on global variables and what depend only on local variables of the caller. I call these two sets GF and LF, respectively. Note that GF and LF do not form a partition: some potential functions in the complement of $GF \cup LF$ depend on both local and global variables. As explained earlier, I am going to assume a set A of potential annotations to use for the callee procedure p. For any annotation $\Phi_a \in A$, I will write $(a_i^e)_i$ and $(a_i^x)_i$ for the coefficients of the entry and exit potential function of Φ_a. With this notation, the analysis generates the following system of constraints.

— The potential associated with local variables before the call can be recovered after:

$$\bigwedge_{i \in LF} k'_i = k_i.$$

— The potential associated with global variables only is passed from the caller to the callee, and fetched back after the return:

$$\bigwedge_{i \in GF} \left[k_i = \sum_{\Phi_a \in A} a_i^e \wedge \sum_{\Phi_a \in A} a_i^x = k_i' \right].$$

— Finally, we consider the coefficients of base functions that depend on both local and global variables. Such coefficients are constrained to be zero at the call and return sites in the caller because we do not know how their base function will evolve through the call:

$$\bigwedge_{i \notin GF \cup LF} \left[k_i = 0 \wedge k_i' = 0 \right].$$

To justify that these constraints enforce the admissibility condition (A4), we observe that they ensure that the initial and final annotations are $\Phi_\ell = \Psi + \sum_{\Phi_a \in A} \Phi_{a\,\ell_e}$ and $\Phi_{\ell'} = \Psi + \sum_{\Phi_a \in A} \Phi_{a\,\ell_x}$. In these expressions, Ψ only depends on local variables and is defined as $\sum_{b_i \in LF} k_i \cdot b_i$. These two constraints indeed match the form required by condition (A4) when the coefficient k_Φ is defined to be 1 if $\Phi \in A$ and 0 otherwise.

Other constraints. To be useful to the user, one annotation Φ for the main procedure has to have its final potential Φ_{ℓ_x} be constrained to be the goal g. The goal is written as a linear combination of base functions and the coefficients obtained are used to constrain the coefficients of Φ_{ℓ_x}.

Finally, to respect the last clause of Definition 3, all the base functions that do not depend only on global variables have their coefficients constrained to be 0 in all the potential functions of procedure entry and exit points.

5.7.2 Objective Function for the Linear-Programming Solver

The generated constraints can be solved by an off-the-shelf linear-programming solver. To derive the best possible bound, we would like to minimize the potential $\sum_i k_i^e \cdot b_i(\sigma)$ that is associated with the entry point of the main procedure. When all the base functions $b_i(\sigma)$ are non-negative, this minimization can be achieved using a weighted sum $\sum_i w_i \cdot k_i^e$ as objective function in the linear program. The constant weights w_i should be set to assign a higher priority to the coefficients of higher-degree base functions b_i.

A more robust method for non-negative base functions that prioritizes the minimization of the coefficients of high-degree base functions is to use the support for efficient iterative solving that is provided by modern linear-programming solvers. First, use the objective function $\sum_i k^e_{j_i}$ to minimize the coefficients $k^e_{j_i}$ of the base functions b_{j_i} with the highest degree. If the solver finds a solution, then add the constraint $\sum_i k^e_{j_i} = o_0$ where o_0 is the objective value, and re-run the solver to optimize the coefficients for the lower-degree base functions.

When some base functions could be negative in the initial state, use a linear system to rewrite the initial potential as $X + \sum k_i \cdot F_i$ where $(F_i)_i$ is a family of rewrite functions in the initial state. Then, constrain X to be 0 and minimize the coefficients k_i as described before.

5.8 Pastis: A Practical Implementation

I implemented the framework of Section 5.7 in a new tool called Pastis (polynomial amortized size-tracking inference system) that computes polynomial resource bounds for imperative integer programs. Programs are internally represented as described in Section 5.2, but the tool accepts as input both a minimal imperative language and LLVM bitcode. The base actions allowed are assignments $v \leftarrow e$ where $v \in \mathcal{V}$ is a program variable and e is a simple expression, and non-deterministic assignments $v \leftarrow *$ to represent unsupported operations like shifts and divisions. The simple expressions are additions, subtractions, and multiplications of constants and program variables. The base functions are picked among the monomials M defined below.

$$\begin{array}{lll} \text{(Monomials)} & M := 1 \mid v \mid M_1 \cdot M_2 \mid \max(0, P) & v \in \mathcal{V} \\ \text{(Polynomials)} & P := k \cdot M \mid P_1 + P_2 & k \in \mathbb{Q} \end{array}$$

In the following, I still use the Python-like syntax of previous chapters; the control-flow graphs are derived in a standard way from this syntax.

5.8.1 Examples Programs and Bounds

Simple polynomial example. The program shown in Figure 5.2 has a polynomial bound on the loop counter z. I will use it to demonstrate the application of potential reasoning and rewrite functions. As before, potential annotations are represented between curly braces; in the annotations, I use a saturating subtraction operation $a \dot{-} b := \max(0, a - b)$. (In the implementation, $\max(0, \cdot)$ is used, but this notation makes the presentation lighter.) As we will see, the annotation of this

```
1  def nested():
2      ${\binom{n \dot{-} 0}{2} + z}$
3      while n > 0:
4          $I_1 = \{\binom{n \dot{-} 0}{2} + z\} \quad \geqslant \quad \{\binom{n \dot{-} 1}{2} + (n \dot{-} 1) + z\}$
5          n = n - 1
6          $\{\binom{n \dot{-} 0}{2} + (n \dot{-} 0) + z\}$
7          m = n
8          $\{\binom{n \dot{-} 0}{2} + (m \dot{-} 0) + z\}$
9          while m > 0:
10             $I_2 = \{\binom{n \dot{-} 0}{2} + (m \dot{-} 0) + z\} \quad \geqslant \quad \{\binom{n \dot{-} 0}{2} + (m \dot{-} 1) + (z{+}1)\}$
11             m = m - 1
12             $\{\binom{n \dot{-} 0}{2} + (m \dot{-} 0) + (z{+}1)\}$
13             z = z + 1
14             $\{\binom{n \dot{-} 0}{2} + (m \dot{-} 0) + z\} = I_2$
15         $\{\binom{n \dot{-} 0}{2} + (m \dot{-} 0) + z\} \quad \geqslant \quad \{\binom{n \dot{-} 0}{2} + z\} = I_1$
16     $\{\binom{n \dot{-} 0}{2} + z\} \quad \geqslant \quad \{z\}$
```

Figure 5.2: Polynomial example.

example is admissible. It was found by solving the system of linear constraints derived using the method of Section 5.7 on this program text. In this example, because there are no procedure calls, there are only two kinds of admissibility checks to do: (i) assignment checks (base actions), and (ii) potential-rewrite checks (weakenings). The potential rewrites are all marked with a "$\geqslant$" sign (lines 4, 10, 15, and 16). Note that the translation to a control-flow graph would introduce guard edges to express conditionals, but those never change the potential functions, according to the admissibility condition (A2).

It is best to understand the annotation in terms of "potential transfers," i.e., how the potential flows from one base function to another through the program execution. Here, the core idea is that the quadratic potential associated with the counter of the outer loop will, at each decrement (line 5), generate a linear potential used to pay for the increments of the counter z in the inner loop. The potential behavior of the inner loop is simply to shuffle potential from $m \dot{-} 0$ to the loop counter z. In all the potential functions of the inner loop, the quadratic part $\binom{n \dot{-} 0}{2}$ is carried through. It is used at the end of the inner loop to restore the invariant I_1 of the outer loop.

The validity of the weakenings can be justified with rewrite functions as explained in Section 5.6. For example, on line 4, we use the rewrite function $F := \binom{n \dot{-} 0}{2} - \binom{n \dot{-} 1}{2} - (n \dot{-} 1)$ to show that $\binom{n \dot{-} 0}{2} + z \geqslant \binom{n \dot{-} 1}{2} + (n \dot{-} 1) + z$. Indeed, the right-hand side of the inequality can be rewritten as $\binom{n \dot{-} 0}{2} + z - (1 \cdot F)$. Note that the rewrite function F can be used on line 4 because it is under the check "`while n > 0`". It could not be used on line 2, where no information about n is known yet.

```
 1  def qsort(l, h):
 2      $\Phi_e := \{\binom{h \dot{-} l}{2} + z\}$
 3      if l < h:
 4          $\{\binom{h \dot{-} l}{2} + z\}$
 5          hint   $\geqslant$   $\{\binom{h \dot{-} (l+1)}{2} + (h \dot{-} (l+1)) + z\}$
 6          m = l
 7          $\{\binom{h \dot{-} (l+1)}{2} + (h \dot{-} (m+1)) + z\}$
 8          while m < h-1 and random:
 9              m = m + 1
10              z = z + 1
11          $\{\binom{h \dot{-} (l+1)}{2} + z\}$
12          hint   $\geqslant$   $\{\binom{h \dot{-} (m+1)}{2} + \binom{m \dot{-} l}{2} + z\}$
13          qsort(l, m)
14          $\{\binom{h \dot{-} (m+1)}{2} + z\}$
15          qsort(m+1, h)
16          $\{z\}$
```

Figure 5.3: Quicksort core.

Polynomial example with recursion. Figure 5.3 contains an implementation of the core of the Quicksort algorithm. All array operations are abstracted away and we look for a bound on the variable z at the end of the procedure. Note that the procedure `qsort` has two arguemnts. In the implementation, as in the programs of Section 5.2, the arguments are passed via global variables. This approach is similar to machine calling conventions that use registers to pass arguments.

The bound Φ_e on z is quadratic. It can be expressed precisely using the binomial basis. Indeed, we will see below that Φ_e is a tight worst-case bound on z. I left all the annotations on the inner loop unspecified because they follow exactly the same pattern as the ones in the inner loop of the previous example. The interesting parts of the derivation are the two weakening hints on lines 5 and 12, and the two recursive calls. The first weakening uses the binomial identity $\binom{X+1}{2} = \binom{X}{2} + X$. This identity is applied with $X := h \dot{-} (l+1)$ using the fact that, in this branch, $h \dot{-} (l+1) = h \dot{-} l - 1$. The current implementation is not able to infer this rewrite function from the body of the recursive function alone; we thus use an explicit rewrite hint. Similarly on line 12, we use a hint to prove that

$$\binom{X+Y}{2} \geqslant \binom{X}{2} + \binom{Y}{2} + X \cdot Y \geqslant \binom{X}{2} + \binom{Y}{2}, \tag{5.9}$$

with $X := h \dot{-} (m{+}1)$ and $Y := m \dot{-} l$. At this point, $X + Y = h \dot{-} (l+1)$ because $l \leqslant m < h$.

Let us now discuss the recursive calls on lines 14 and 16. As in Section 5.7, I describe the potential before and after the calls piecewise. On line 14, after the arguments are assigned, $\binom{m \dot{-} l}{2} + z$ will become $\binom{h \dot{-} l}{2} + z = \Phi_e$. This quantity is the potential passed to the recursive call. The term $\binom{h \dot{-} (m+1)}{2}$ only depends on local variables and, as such, remains unchanged by the call; it is thus picked as a frame. Finally, on line 14, the final potential of qsort—z—is returned and added to the frame. The recursive call of line 16 follows a similar logic, but this time without any frame.

The procedure qsort exhibits its worst-case behavior when the internal loop goes all the way from l to $h - 1$. Note that all the weakenings of the derivation are actually pure potential rewrites ($\geqslant$ is in fact $=$), except for the one on line 12. On that line, the term $X \cdot Y$ of Equation (5.9) is lost potential. However, in the worst-case scenario, $m = h - 1$ on line 12 and thus $X \cdot Y = 0$, making the weakening a pure potential rewrite, too. Thus, in the worst case of Quicksort, no potential is ever lost and the bound $\binom{h \dot{-} l}{2} + z$ is exact: $\Phi_e(\sigma_e) = \sigma_x(z)$, where σ_e and σ_x are, respectively, the entry and exit states of qsort.

5.8.2 Heuristics for Base and Rewrite Functions

In Pastis, I make use of invariants generated by abstract interpretation to generate basis and rewrite functions. For example, if some variable v can be proved non-negative at one program point, the base function $\max(0, v)$ is added together with a set of rewrite functions that will be needed to transfer potential to and from this base function. For instance, the rewrite function $\max(0, v) - \max(0, v-1) - 1$

```
if v > 0:
  {max(0, v)} ⩾
  {max(0, v−1)+1}
  v = v - 1
  {max(0, v)+1}
```

Figure 5.4

is registered. As shown in Figure 5.4, this rewrite function can be used before a decrement when $v \geqslant 1$. Higher-degree base functions are introduced by considering successive powers and products of linear base functions.

By choosing as base functions the lengths of intervals that can be formed by pairs of program variables (e.g., $\max(0, b - a)$), Pastis strictly generalizes the interval system. Any derivation in that system can be encoded in the framework of Section 5.7. Moreover Pastis is more general because it allows higher-degree base functions, as well as base functions that do not match exactly the interval pattern.

5.8.3 Procedure Calls Support

In Section 5.7.1, I left unspecified the choice of potential annotations on procedure call edges. In Pastis, procedure annotations are generated on-demand and at each call site encountered while trying to analyze the main procedure. This section presents the precise algorithm used by Pastis and explains the two problems that it solves: first, different call sites often require different procedure annotations; and second, recursive programs sometimes require so-called resource-polymorphic recursion.

```
def P():
    x = x - 1
    z = z + 1
```

```
while x > 0:
    {$z + 2\max(0, x) + \max(0, y)$}
    P()
    {$z - 1 + 2\max(0, x + 1) + \max(0, y)$}
    y = y + 1
while y > 0:
    {$z + \max(0, y)$}
    P()
    {$z - 1 + \max(0, y)$}
    y = y - 1
{$z$}
```

Figure 5.5: The procedure `P` needs different annotations depending on the call site.

Handling diverse call sites. In practice, using only one annotation for one procedure is too restrictive because procedures can be executed in different contexts. For example, on Figure 5.5, the same procedure `P` needs two different annotations for its two call sites. This is because the procedure is used differently in both cases: in the first loop, the change it makes to x must be accounted for to explain the loop behavior; in the second loop, the action of `P` on x is irrelevant, and only its action on the loop counter z matters. One method to handle multiple call sites in the absence of recursion is to generate a new annotation as soon as a procedure call edge is encountered (this essentially amounts to inlining all the procedure calls).

However this simplistic strategy is not applicable in the presence of recursive procedures; it would lead to an infinite recursion in the analyzer. If the analyzer processes procedures recursively starting from the main procedure, it can keep an active set of the annotations that are currently being generated; as soon as a procedure call for a procedure in the active set is encountered, the annotation from the active set is used (instead of recursively analyzing the procedure body).

Handling resource-polymorphic recursion. In practice, the annotation used around a recursive call and at the entry and exit points of a procedure often need to be different: some framing is often required at call sites. This framing lets potential "go through" the recursive call to be used after it.

An example of such a situation is displayed in Figure 5.6(a). In this example, we look for an upper bound on n at the program exit. It is easy to show by induction that n is invariant across a call of `P`. However, if the analysis naively uses the equality constraints $\Phi_e = \Phi_c$ and $\Phi_r = \Phi_x$, no bound can be found. Indeed, Φ_x has to be set to the goal $\{n\}$, and then $\Phi_r = \Phi_x \circ [\![n \leftarrow n+1]\!] = \{n{+}1\}$ by the admissibility condition (A1); but $n+1 \neq n$, preventing $\Phi_r = \Phi_x$. One solution to this problem is to allow a framing to be performed at call sites. In the example in Figure 5.6(a), the frame used is 1.

If the frame required only depends on local variables, it can be passed unchanged, because local variables of the caller cannot be changed in the callee. However, sometimes, the frame has to depend on global variables. In that case, we cannot merely add a term like $5n$ to a potential function before and after a procedure call because the global variable n could be modified by the procedure. We need to infer a size change for this variable. But that is precisely what our system is designed for. Thus, a sound and practical approach that I use in Pastis and that has been pioneered in [HH10] is to use as a frame an annotation obtained by another run of the analysis on the callee procedure.

```
def P():
    $\Phi_e := \{n\}$
    if n > 0:
        n = n - 1
        $\Phi_c := \{n + 1\}$
        P()
        $\Phi_r := \{n + 1\}$
        n = n + 1
    $\Phi_x := \{n\}$
```

(a)

```
def Q():
    $\Phi_e := \{z + \binom{n}{2}\}$
    if n > 0:
        n = n - 1
        $\Phi_c := \{z + \binom{n}{2} + n\}$
        Q()
        $\Phi_r := \{z + n\}$
        z = z + n
        n = n + 1
    $\Phi_x := \{z\}$
```

(b)

Figure 5.6: Procedures where resource-polymorphic recursion is necessary to infer a bound.

Consider, for example, the procedure `Q` in Figure 5.6(b). It is a variant of `P` in which the assignment "`z = z + n`" is added after the recursive call. Assume that the potential goal is $\Phi_x = \{z\}$. If z_0 and n_0 are the values of z and n before the call of `Q` then we have $z = z_0 + \binom{n_0}{2}$ after the call. Consequently, Φ_e is a sound potential annotation for the entry point of `Q`. To justify this potential annotation, we attempt to use the same annotation Φ_e for the potential before the recursive call. Note, however, that there is some additional potential n available in Φ_c. To transfer this potential to the return point Φ_r we have to analyze how n changes during a call to `Q`. As with the example on Figure 5.6(a), n is invariant across the call, and we can perform a similar analysis on `Q` to derive $\Phi'_e = \{n\}$ and $\Phi'_x = \{n\}$ for the entry and exit points of `Q`. The annotations before and after the recursive call can now use the combined annotations $\Phi_e + \Phi'_e$ and $\Phi_x + \Phi'_x$, respectively. These compound annotations for the call and return points make full use of the flexibility offered by the admissibility condition (A4) by adding together multiple annotations for the same procedure. Intuitively, each annotation is about the size change of a different variable; in the example, z and n.

Putting it together: generation of IPAs in Pastis. The complete algorithm to generate an IPA from an input control-flow graph is displayed as pseudocode in Figure 5.7. It generates one annotation per call site (line 11), except for recursive calls where an annotation from the active set (line 8) is used in combination with a frame obtained by recursively calling the analysis function

```
let rec go degree active p =
  let $\Phi$ = new_template_annot degree p in
  if degree = 1 then ($\Phi_{\ell_e}$, $\Phi_{\ell_x}$) else
  for ($\ell$, a, $\ell'$) in p.edges do
    match a with
    | Call q ->
      let lo, (gc, gr) = fresh_local_global () in
      if $\exists$ (e, x). {q $\mapsto$ (e, x)} $\in$ active
        then let (e', x') = go (degree - 1) [] q (* Frame *)
             in constrain (gc = e + e' $\wedge$ gr = x + x')
        else let (e, x) = go degree ({p $\mapsto$ ($\Phi_{\ell_e}$, $\Phi_{\ell_x}$)} :: active) q
             in constrain (gc = e $\wedge$ gr = x)
        constrain ($\Phi_{\ell}$ = gc + lo $\wedge$ $\Phi_{\ell'}$ = gr + lo)
    | ... -> (* Constrain other actions following Section 5.7.1 *)
  done
  ipa $\leftarrow$ { p $\mapsto$ $\Phi$ } :: ipa (* Add the new annotation for p to the IPA *)
  ($\Phi_{\ell_e}$, $\Phi_{\ell_x}$) (* Return entry and exit potential functions *)

(* The main analysis function *)
let analysis cfg goal max_degree =
  let (e, x) = go max_degree [] cfg.main in
  constrain (x = goal)
  e (* Return the entry potential function *)
```

Figure 5.7: Pseudocode of the algorithm used in Pastis to generate an IPA for the input program cfg. The algorithm addresses the need to have multiple annotations for multiple call sites and handles resource-polymorphic recursion.

with a smaller set of base functions (line 9). This smaller set is obtained by decreasing the maximum degree of the base functions used. This process of limitation ensures the termination of the analysis. Moreover, in practice, I do not know of any program requiring a frame of the same degree as the main procedure annotation (the one fetched from the active set). RAML implements a similar mechanism.

The pseudocode implementation also supports framing of potential assicated with local variables only. On line 7, the function `fresh_local_global` is used to create three fresh template potential functions. The first one—`lo`—depends only on local variables, and the last two ones—`(gc, gr)`—depend only on global variables. Following the constraints for call edges described in Section 5.7, the local part of the initial and final potential Φ_ℓ and $\Phi_{\ell'}$ is constrained to be the same using the fresh `lo` (line 13); and their global parts `gc` and `gr`, respectively, are constrained to be a sum of entry and exit annotations for the caller—either fetched from the active set, or created by a recursive call to the analysis (lines 10 and 12).

The analysis procedure always terminates because each recursive call makes the pair (`degree`, #`active`) decrease in the lexicographic sense. This termination argument might raise doubts about the efficiency of the overall procedure. In practice, however, many optimizations limit the number of variables in the linear program and, more importantly, most programs have bounds with relatively small degrees. The biggest bottleneck in Pastis turned out to be the imprecision of the heuristics to infer base and rewrite functions. They often overload the system with too many candidates.

5.8.4 Experimental Evaluation

Benchmark set. To evaluate the performance of Pastis, I used the benchmark suite of the paper that presented Loopus'15 [SZV15]. That paper compares Loopus'15 to Loopus'14 [SZV14], KoAT [BEF$^+$14], and CoFloCo [FMH14]. The comprehensive program set used is based on the compiler optimization Collective Benchmark (cBench) which contains 211,892 lines of C code. The C files are pre-processed to extract all functions into independent files; those files are then translated to the various input formats of the tools compared. Pastis accepts directly LLVM bitcode and processes it to extract a control-flow graph as described in Section 5.2. The benchmark was run in intraprocedural mode because only KoAT and Pastis support procedure calls.

Machine. The experimental evaluation was run on a machine equipped with an Intel Xeon CPU clocked at 3.10GHz and 32GB of memory.

Tool	Bounded	$O(1)$	$O(n)$	$O(n^2)$	$> O(n^2)$	Timeout
Loopus'15	806	205	489	97	15	6
Pastis	459	187	229	43	0	127
Loopus'14	431	200	188	43	0	20
KoAT	430	253	138	35	4	161
CoFloCo	386	200	148	38	0	217

Table 5.1: Experimental evaluation of Pastis on 1659 functions from the cBench benchmark. The tools were run with a timeout of 60 seconds.

Results. Table 5.1 contains a digest of the experiments. The results for Loopus'14, KoAT, and CoFloCo were taken from the evaluation done by Sinn et al. [SZV15]. By the number of examples only, Pastis is in the same ballpark as the majority of other tools. However, as explained in the next chapter, Pastis is the only resource-analysis tool able to generate proof certificates for the bounds it infers. Pastis infers bounds quickly in most cases: 98% are found in less than 3 seconds.

Loopus'15 deserves special mention as it performs remarkably well; its versatility on this benchmark set is impressive. In addition to its sophisticated intraprocedural loop-analysis algorithm, Loopus'15 implements many practical ad hoc features. Among others, the C types are retrieved from the debugging information in the LLVM bitcode; some heuristics to identify loops on null-terminated C strings and files are built in; and finally, at the expense of compositionality, large sections of code can be represented symbolically with a Scheme-like syntax in the case where top-level loops are preceded with complex straight-line code. Similar features are not yet implemented in Pastis.

5.9 Conclusion

The core contribution of this chapter is the generic framework for the automation of resource-analysis tools. It generalizes the interval system of the previous chapter and even, in some sense, all the previous potential-based automated systems. Indeed, even though the implementation Pastis is limited to polynomial bounds, IPAs and the general method to generate constraints could be adapted to, for instance, compute logarithmic bounds. Now, I briefly sketch how this extension would go. Instead of only considering monomials as base functions, we would add a new "logarithm" base function $\mathrm{L}_2(\cdot)$ that returns the smallest integer n such that 2^n is greater or equal to its argument.

Then, rewrite functions could be added to encode the fact that

$$\mathrm{L}_2(2 \cdot x - 1) \geqslant \mathrm{L}_2(x) + 1, \qquad \text{and} \quad \mathrm{L}_2(2 \cdot x) \geqslant \mathrm{L}_2(x) + 1,$$

when $x \geqslant 2$. Using those, one could for instance derive a logarithmic bound for a binary search procedure, as shown below.

```
$\{\mathrm{L}_2(b-a)\}$
z = 0
while a < b - 1:
    $\{\mathrm{L}_2(b-a) + z\}$
    if random:
        $\{\mathrm{L}_2(b-a) + z\} \quad \geqslant \quad \{\mathrm{L}_2(2 \cdot (b - (a+b)/\!/2) - 1) + z\}$
        a = (a + b) / 2
        $\{\mathrm{L}_2(2 \cdot (b-a) - 1) + z\} \quad \geqslant \quad \{\mathrm{L}_2(b-a) + z + 1\}$
    else:
        $\{\mathrm{L}_2(b-a) + z\} \quad \geqslant \quad \{\mathrm{L}_2(2 \cdot ((a+b)/\!/2 - a)) + z\}$
        b = (a + b) / 2
        $\{\mathrm{L}_2(2 \cdot (b-a) + z\} \quad \geqslant \quad \{\mathrm{L}_2(b-a) + z + 1\}$
    z = z + 1
    $\{\mathrm{L}_2(b-a) + z\}$
$\geqslant \{z\}$
```

In the annotations, I used $x/\!/2$ to denote the integer division of x by 2. Note that the treatment of the two branches is subtly different, and that both of the rewrite functions mentioned above are used—the first in the `if` branch, and the second in the `else` branch. This is because of the behavior of the integer division: for any integer x, $x - 1 \leqslant 2 \cdot (x/\!/2) \leqslant x$.

Finally, while the performance of Pastis is not setting a new record, its real strength lies in the clean framework of potential annotations it is based on. This framework is generic—as demonstrated above—, provides full support for possibly mutually recursive procedures, and enjoys a rigorous formal definition which allows to generate mechanized proof certificates with very little effort—as detailed in the next chapter.

Chapter 6

Mechanized Certificates for Resource Bounds

For certain applications, like avionics and other critical embedded systems, it is desirable or even mandatory to get strong garantees about the resource behavior of programs. If this information about the program is obtained with a tool, there are two ways one could go to obtain this high level of confidence. A first approach is to make sure that the tool itself is correct and only outputs correct results. The gold standard to achieve correctness is to use formal verification; this is for instance what the Verasco project [JLB$^+$15] does, their authors have constructed a complete static analyzer inside the Coq proof assistant. Another approach is to process by *validation* and have the resource-bound tool generate *certificates* that justify the validity of the results. Once the tool has run and produced a resource bound and a certificate, one simply has to check the validity of the certificate, either manually, or with another trustworthy tool. Validation is used extensively in CompCert [TL10, TL09] to check the results of some complex optimization passes.

In this chapter, I will explain how I implemented a validation scheme in Pastis using the Coq proof assistant as a checking tool. The Coq system has a minimal and time-tested trusted computing base, it is thus an ideal target for certificates. The core of a certificate generated by Pastis is an interprocedural potential annotation, as defined in Chapter 5. These annotations are a natural choice for certificates; indeed, together with their soundness result (Proposition 2), they fully justify the validity of resource bounds. Moreover, in the implemenation of Pastis, these annotations are pervasive, thus, collecting and emitting them at the end of the execution required only a small implementation effort. To the user, the most interesting part of the certificates generated is a theorem stating formally that the derived bounds are sound with respect to the operational semantics of the input interprocedural control-flow graph. The generated certificates do not rely on any assumptions; in particular, they also include a soundness proof of the invariants derived using abstract interpretation. Finally, they can be checked in batch mode, without modifications, by the Coq proof checker.

6.1 Motivations for Mechanized Certificates

The benefits of checking bounds with a proof assistant are three fold.

First, as explained in the introduction, it greatly increases the confidence in generated bounds, which is especially critical considering it is not rare that linear-programming solvers silently overflow and return an unsound solution. All resource-analysis tools using linear-programming solvers are currently vulnerable to this issue. While working on the implementation of Pastis, proof certificates helped me find one bug in the abstract-interpretation procedure. The fine grain of the certificates generated helped me pinpoint the issue very quickly.

Second, proof certificates are a license to implement aggressive heuristics and optimizations in the tool without risking unnoticed soundness issues.

Third, certificates allow the integration of resource bounds into larger formal developments. I believe that automating the inference of resource-bound theorems will enable a new class of software verification where not only the correctness is proved, but also quantitative properties such as real-time guarantees and memory usage, which are often neglected.

6.2 Overview of the Certificates Generated by Pastis

Because many proofs and tactics are shared among the files generated by Pastis, all the Coq files generated import a shared library which I describe in Section 6.3; I will refer to it as *the support library*. A generated file starts by defining the program analyzed as a control-flow graph with procedure calls. Programs are represented in the same way internally in Pastis and in the Coq formalization; serializing them is a simple matter of pretty-printing in Coq's syntax, and no transformations have to be performed. Then, for each procedure and program point, the logical contexts that result from the abstract-interpretation procedure are listed; they are expressed as

Coq terms of type `state` → `Prop`. Like in Chapter 5, only one logical context is associated to each location. After this, the potential annotations are listed; they are represented as maps from program points to potential functions. Potential functions are Coq terms of type `state` → `Q`. From these two pieces of data, a complete interprocedural annotation is defined as in the proof of Proposition 2 in Section 5.5. In Coq syntax, the complete definition of an interprocedural annotation for a program with a single `start` procedure that has two potential annotations would be:

```
Inductive proc: Type :=
  P_start. (* "start" is the only procedure of the program *)
(* ... Definitions of ai_start, annot0_start, and annot1_start ... *)
Definition ia: IA := fun p ⇒
  match p with
  | P_start ⇒
    [mkPS Q (fun n z s ⇒ ai_start n s ∧ annot0_start n s ⩽ z);
     mkPS Q (fun n z s ⇒ ai_start n s ∧ annot1_start n s ⩽ z)]
  end.
```

In the code snippet, `mkPS` is the constructor for a procedure specification $[\forall z \in Z. P_z]$. In that case the annotation defined has the set of rational numbers—`Q` in Coq—as auxiliary state Z, and P_z is represented as a proposition parameterized by, a node `n`, an auxiliary state `z`—here, a rational number—, and a memory state `s`. In this example, `ai_start` is the the collection of logical contexts that resulted from the abstract-interpretation procedure, and `annot0_start` and `annot1_start` are two potential annotations for the `start` function.

Then, the interprocedural annotation `ia` is proved to satisfy the admissibility conditions of Definition 2. In the Coq file, it is expressed as follows.

```
Theorem admissible_ia: IA_VC ia.
Proof. prove_ia_vc. Qed.
```

The proof of admissibility is always a single call to the tactic `prove_ia_vc` imported from the support library distributed with Pastis. This tactic call is where the bulk of the proving happens. On bigger examples, it can take several seconds to finish.

Finally, a user-readable theorem expresses the soundness of the bound that was generated and displayed to the user.

```
Theorem bound_valid: ∀ s1 s2,
  steps P_start (proc_start P_start) s1 (proc_end P_start) s2 →
  s2 G_z ⩽ (3#2) * s1 G_g + (1#2) * s1 G_g^2.
Proof. prove_bound ia admissible_ia P_start. Qed.
```

In this theorem, s1 and s2 are the initial and final program states, respectively, of the execution of the start procedure (which appears in the hypothesis as "steps P_start ..."). In this example, the goal was set to $\{z\}$, i.e., the value of the global variable z at the end of the execution (shown as "s2 G_z"). It is bounded in terms of the initial value of another global variable g. A rational number a/b is represented in Coq using the notation a#b. The proof of this theorem leverages both the Proposition 1 about admissible interprocedural annotations—proved once and for all in the support library—and one annotation of the start procedure that has the goal as final potential.

6.3 Coq Support Library

The support library contains all the Coq code that must be part of any certificate generated; it makes the amount of boilerplate in certificates minimal. The library is very compact: less than 400 lines of definitions, and less than 150 lines of proofs according to the program coqwc. I attribute this concision to an extensive use of the automation built into Coq, to the simplicity of the generic framework of Chapter 5, and also to some simplifying assumptions like the lack of overflow semantics for integer operations and the simplistic memory model (e.g., no pointers and no structured data). This section presents the main components of the shared library.

6.3.1 Programs and Annotations

The shared library contains formal definitions for both the control-flow graphs presented in Section 5.2 and the interprocedural annotations presented in Section 5.3. Intreprocedural annotations are represented as Coq terms of type IA; the admissibility conditions of Definition 2 are expressed as a predicate IA_VC: IA → Prop, which is designed to be easily checked automatically; and finally, the core Proposition 1 about admissible interprocedural annotations is proved.

6.3.2 Rewrite Functions

Rewrite functions, introduced in Section 5.6, are simply defined as functions of the memory state that are non-negative at their point of use. Let us now discuss the requirements on the Coq representation of rewrite functions. In the certificates, rewrite functions have play two roles with different needs:

1. They have to be linearly combined with potential functions.

2. They have to be proven non-negative at their point of use automatically.

To meet the first requirement, one could plainly represent rewrite functions as maps from memory states to rational numbers. However, this would make proving their non-negativity challenging. Indeed, for rewrite functions acting on non-linear base functions such as

$$\binom{\max(0,x)}{2} - \binom{\max(0,x-1)}{2} - \max(0,x-1), \tag{6.1}$$

it might be the case that no built-in Coq tactic can automate the non-negativity proof (in that case, assuming $x \geqslant 1$). Coq only provides complete support for linear integer arithmetic.

In the support library, the solution to this challenge was to use a domain-specific language. The main idea is to remark that, in the programs handled by Pastis, non-negativity proofs can often be reduced to checking linear inequalities on integers using elementary lemmas about binomial coefficients and the max function. For example, the rewrite function (6.1) can be proved non-negative by remarking that ($\star$) the function $x \mapsto \binom{x}{k}$ is monotonic for non-negative arguments when $k \in \mathbb{N}$, and that ($\star\star$) if $a \geqslant \delta \geqslant 0$, then $\max(a) \geqslant \max(a-\delta)+\delta$. Composing ($\star$) and ($\star\star$), we can conclude the non-negativity of (6.1) under the assumption that $x \geqslant 1$.

Now, I will use a domain-specific language to encode the proof we just did. In Coq, this language is deeply embedded as the inductive type `rewrite_func`.

```
Definition rewr: rewrite_func := binom_monotonic 2 (max0_sub x 1).
```

Here, `(max0_sub x 1)` represents ($\star\star$) with $a = x$ and $\delta = 1$, and `binom_monotonic 2 (...)` represents ($\star$) with $k = 2$ applied to the inequality $\max(x) \geqslant \max(x-1)+1$ that is the result of its second argument. We have in fact defined a small proof language for simple facts about arithmetic. To give a meaning to the term `rewr` we define an *interpretation function* which acts on all terms of type `rewrite_func`. In the Coq implementation, the interpretation function returns a pair of a list of hypotheses inequalities and a conclusion inequality.

```
Inductive ineq := a_ge_b (a b: Q).
Fixpoint interpret (f: rewrite_func): (list ineq * ineq) :=
  match f with ... end.
```

By design of the domain-specific language, the hypotheses are a list of linear conditions under which the conclusion holds. The inequalites are represented by terms of type `ineq` that are essentially pairs of two rational numbers; the intended meaning is that the first number is greater than or equal to the second. (The term `(a_ge_b a b): ineq` is used in place of `(a ⩾ b): Prop` because the interpretation function is inductively defined and needs to inspect the result of recursive calls.) Finally, a Coq theorem formally states the semantics of the pairs computed by the interpretation function.

```
Definition ineq_prop (i: ineq): Prop := match i with (a_ge_b a b) ⇒ (a ⩾ b) end.
Theorem interpret_semantics:
  ∀ (f: rewrite_func),
  let (hyps, conc) := interpret f in
  (List.Forall ineq_prop hyps) → (ineq_prop conc)
```

When extracting to Coq, Pastis provides rewrite functions using this domain-specific language. During the checking of a certificate, the tactics described in the next section use the `interpret` function to retreive the actual rewrite function, prove it non-negative using the theorem `interpret_semantics` (proving the validity of the hypotheses with Coq's built-in support for linear arithmetic), and put it in the proof context as a hint for future use.

6.3.3 Automating Admissibility Proofs

To automate fully the checking of the certificates produced, a couple relatively small tactics are defined (110 lines of Coq); they are tightly coupled with the proof-generation module of Pastis. The challenge solved by these tactics is to mainly to check the interprocedural annotation that is emitted by Pastis. This interprocedural annotation, as explained in Section 6.2, is constructed like the one in the proof of Proposition 2 in Section 5.5. Thus, the checking performed by tactics follows essentially the same structure as what is described in that proof.

More precisely, the tactic to automate the admissibility proof splits its work in two tasks for each edge: (*i*) checking the validity of the abstract-state transformation, and (*ii*) checking the potential admissibility condition of Definition 3 that matches the edge type. For (*i*), the Coq decision procedure `lia` for linear-integer arithmetic is used. This tactic is sufficient because the logical contexts derived by abstract interpretation are merely linear constraints between program variables. For (*ii*), different tactics are applied depending on the action labelling the edge. Remember that, in all cases, the idea is to prove that the potential is non-increasing $\Phi_\ell(\sigma) \geqslant \Phi_{\ell'}(\sigma')$ under the assumption that a step is taken in the operational semantics $(\ell, \sigma) \to (\ell', \sigma')$. Because the coefficients of potential functions are constants (obtained from the linear-programming solver), the checking of inequalities is reduced to $\mathbb{Z}$ and automated using integer tactics.

— For base actions, the stability of potential functions described in Section 5.7 makes the checking possible using a tactic to prove equalities on rings (`ring`, in Coq).

— For guard actions, both the program state and the potential annotations are unchanged, so the check can be proved by the reflexivity of $\geqslant$ on rational numbers.

— For weakenings, rewrite functions have to be used. Pastis includes in the generated certificate all the rewrite functions that were used by the linear-programming solver. They are represented using the domain-specific language described in Section 6.3.2. Using the machinery of that section, they are passed to the `lia` tactic to prove the goal.

— Finally, for call edges, all the procedure specifications of the callee are fetched from the interprocedural annotation. Like in the previous item, the inequalities resulting from those procedure specification are passed to the `lia` tactic that finishes proving the goal.

Importantly, according to the Coq reference manual [Coq], the `lia` tactic is complete. This property means that a failure when checking a proof certificate can only be explained by an invalid certificate. Invalid certificates can be the result of a bug in Pastis—I found one in the abstract interpreter after adding the support for certificates—or of invalid coefficients in the potential annotations (e.g., because of overflows or rounding errors in the linear-programming solver).

6.4 Certificate Generation in Pastis

As far as the implementation of Pastis is concerned, the certificate-extraction feature required only minor modifications. Most of them implement the necessary book keeping to track of the potential annotations and the logical contexts inferred by abstract interpretation. Finally, a new pretty-printing module implemented with the help of Vilhelm Sjöberg outputs all the tracked information in Coq syntax. In the rest of this section, I give more information about the generation process.

Rewrite functions. In Pastis, the only decision procedure available to handle the logical contexts is for linear integer arithmetic. For that reason, rewrite functions are represented using the same domain-specific language used in the Coq implementation. In fact, this language first appeard in the implementation; it only accured to me later that it would greatly simplify the extraction process.

Floating-point numbers. In practice, off-the-shelf linear-programming solvers accept linear programs and return solutions using floating-point numbers. Each floating-point number is a rational number, but quite often, it is not the rational number desired for the certificate. For instance, a loop invariant might contain the coefficient 0.1; assuming no rounding errors happened elsewhere, the floating point number used to represent 0.1 is another rational number $r \neq 0.1$, in fact $|r - 0.1| < 10^{-17}$. For at least two reasons, using r in place of 0.1 is undesirable: first, it might break the loop invariant and make the certificate invalid; second, r must be represented as a fraction of two numbers so large that they would make the certificate checking intractable. Indeed, Coq's evaluation engine and standard library are not optimized for computations on large numbers.

The practical approach I took is to use an elegant algorithm based on Farey sequences that finds the rational number with a bounded denominator closest to an input number. The implementation currently bounds the denominator to be not bigger than 200, but this number can be changed at will. For the curious reader, the algorithm is listed below in OCaml code.

```
(* Returns the closest rational number to x ∈ [0,1] with denominator bounded by n *)
let closest_rational n x =
  let rec go (a, b) (c, d) =
    if b > n then (c, d)
    else if d > n then (a, b)
    else
        let mediant = float_of_int (a+c) /. float_of_int (b+d) in
        if x = mediant then
            if b + d <= n then (a+c, b+d)
            else if d > b then (c, d)
            else (a, b)
        else if x > mediant then
            go (a+c, b+d) (c, d)
        else
```

```
         go (a, b) (a+c, b+d)
  in go (0, 1) (1, 1)
```

Experimental results. For all the 459 programs that were successfully bounded in the experimental evaluation of Section 5.8.4, Pastis generated one associated certificate. Out of those 459 certificates, 424 were successfully checked by Coq. Coq usually processed the files quickly: 311 files take less than 10 seconds to check, and another 83 files take less than 20 seconds. The checking failures are caused by imprecisions in the potential functions resulting either from rounding errors in the linear-programming solver, or from the conversion from floating-point numbers to rational numbers. Especially on higher-degree problems, it is common to see a base function in a loop invariant assigned a small, non-zero, bogus coefficient. Past a certain threshold, the extraction mechanism will output a small rational number when zero is actually needed. On examples with large constants, I also observed overflows in the linear-programming solver leading to obviously unsound bounds. As a practical counter-measure, the LLVM-bitcode input module replaces all "large" constants with non-deterministic expressions.

Chapter 7

Related Work and Conclusion

Verified compilation. Soundness proofs of compilers have been extensively studied and we focus on *formally verified* proofs here. Klein and Nipkow [KN06] developed a verified compiler from an object-oriented, Java-like language to JVM byte code. Chlipala [Chl07] describes a verified compiler from the simply-typed lambda calculus to an idealized assembly language. In contrast to our work, the aforementioned works do not model nor preserve quantitative properties such as stack usage.

Our verified Quantitative CompCert compiler is an extension of the CompCert C Compiler [Ler06, Ler09]. Despite of being formally verified, important quantitative properties such as memory and time usage of programs compiled with CompCert have still to be verified at the assembly level [BBFF$^+$12]. Admittedly, there exists a clever annotation mechanism [BBFF$^+$12] in CompCert that allows to transport assertions on program states from the source level to the target machine code. However, these assertions can only contain statements about memory states but not bounds on the number of loop iterations and or recursion depth of functions. The novelty of our Quantitative CompCert extension to CompCert is that it enables us to reason about quantitative properties of event traces during compilation. Another novelty is that we model the assembly level semantics more realistically by using a finite stack. In particular, we do not have to use pseudo instructions anymore. This is similar to CompCertTSO [SVZN$^+$13]. However, we use event traces to get guarantees on the size of the stack that is needed to ensure refinement. On the other hand, it is always possible that the compiled code runs out of stack space in CompCertTSO.

In the context of the Hume language [HM03], Jost et al. [JLH$^+$09] developed a quantitative semantics for a functional language and related it to memory and time consumption of the compiled code for the Renesas M32C/85U embedded micro-controller architecture. In contrast to our work, the relation of the compiled code with functional code is not formally proved.

Program logics. In the development of our quantitative Hoare logic we have drawn inspiration from mechanically verified Hoare logics. Nipkow's [Nip02] description of his implementations of Hoare logics in Isabelle/HOL has been helpful to understand the interaction of auxiliary variables with the consequence rule. The consequence rule we use in our Coq implementation is a quantitative version of a consequence rule that has been attributed to Martin Hofmann by Nipkow [Nip02].

There exist quantitative logics that are integrated into separation logic [Atk10, HMS13] and they are closely related to our quantitative logic. However, the purpose of these logics is slightly different since they focus on the verification of bounds that depend on the shape of heap data structures. Moreover, they are only defined for idealized languages and do not provide any guarantees for compiled code. Also closely related to our logic is a VDM-style logic for reasoning about resource usage of JVM byte code by Aspinall et al. [ABH+07]. Their logic is more general and applies to different quantitative resources while we focus on stack usage. However, it is unclear how realistic the presented resource metrics are. On the other hand, our logic applies to system code written in C, is verified with respect to CompCert Clight, and can be used to derive bounds for x86 assembly.

Resource analysis. There exists a large body of research on statically deriving stack bounds for low-level code [BDP01, RRW05, CNPQ08] as well as commercial tools such as the *Bound-T Time and Stack Analyser*[1] and Absint's *StackAnalyzer* [FHF07] NOTE: those cannot derive parametric bounds, they also rely on unverified annotations in the machine code for involved program invariants. We are however not aware of any formally verified techniques.

For high-level languages there exists a large number of systems for statically inferring or checking quantitative requirements such as stack usage [JLH+09, CW00, HAH12, AAG+11]. However, they are not formally verified and do not apply to system code that is written in C. For C programs, there exist methods to automatically derive loop bounds [SZV15, GMC09] but the proposed methods are not verified and it is unclear if they can be used for computing stack bounds.

We are only aware of two verified quantitative analysis systems. Albert et al. [ABG+12a] rely on the KeY tool to automatically verify previously inferred loop invariants, size relations, and ranking functions for Java Card programs. However, they do not have a formal cost semantics and do not verify actual stack bounds. Blazy et al. [BMP14] have verified a loop bound analysis for CompCert's

[1]`http://www.bound-t.com`

RTL intermediate language. It is however unclear how the presented technique can be used to verify stack bounds or to formally translate bounds to a lower-level during compilation.

Automated resource analysis. Our work has been inspired by type-based amortized resource analysis for functional programs [HJ03, HH10, HAH12]. Here, we present the first automatic amortized resource analysis for C. None of the existing techniques can handle the example programs we describe in this work. The automatic analysis of realistic C programs is enabled by two major improvements over previous work. First, we extended the analysis system to associate potential with not just individual program variables but also multivariate intervals and, more generally, auxiliary variables. In this way, we solved the long-standing open problem of extending automatic amortized resource analysis to compute bounds for programs that loop on (possibly negative) integers without decreasing one individual number in each iteration. Second, for the first time, we have combined an automatic amortized analysis with a system for interactively deriving bounds. In particular, recent systems [HS14] that deal with integers and arrays cannot derive bounds that depend on values in mutable locations, possibly negative integers, or on differences between integers.

There exist many tools that can automatically derive loop and recursion bounds for imperative programs such as SPEED [GMC09, GZ10], KoAT [BEF$^+$14], PUBS [AAG$^+$12], Rank [ADFG10], ABC [BHHK10] and LOOPUS [ZGSV11, SZV14, SZV15]. These tools are based on abstract-interpretation-based invariant generation and/or term rewriting techniques, and they derive impressive results on realistic software. The importance of amortization to derive tight bounds is well known in the resource analysis community [ABG12b, SZV14]. Currently, the only other available tools that can be directly applied to C code are Rank and LOOPUS. As demonstrated, C4B and Pastis are more compositional than the aforementioned tools. Our technique, is the only one that can generate resource specifications for functions, deal with resources like memory that might become available, generate proof certificates for the bounds, and support user guidance that separates qualitative and quantitative reasoning.

There are techniques [BFGY08] that can compute the memory requirements of object oriented programs with region-based garbage collection. These systems infer invariants and use external tools that count the number of integer points in the corresponding polytopes to obtain bonds. The described technique can handle loops but not recursive or composed functions. We are only aware of two verified quantitative analysis systems. Albert et al. [ABG$^+$12a] rely on the KeY tool to automatically verify previously inferred loop invariants, size relations, and ranking functions for

Java Card programs. However, they do not have a formal cost semantics and do not prove the bounds correct with respect to a cost model. Blazy et al. [BMP14] have verified a loop bound analysis for CompCert's RTL intermediate language. However, this automatic bound analysis does not compute symbolic bounds. Furthermore, there is no way to interactively derive bounds or to deal with resources like memory usage.

Bibliography

[AAG$^+$11] Elvira Albert, Puri Arenas, Samir Genaim, Miguel Gómez-Zamalloa, and German Puebla. Cost analysis of concurrent OO programs. In *Proceedings of the 9th Asian Conference on Programming Languages and Systems*, APLAS '11, pages 238–254, 2011.

[AAG$^+$12] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. Cost analysis of object-oriented bytecode programs. *Theoretical Computer Science*, 413(1):142–159, January 2012.

[ABG$^+$12a] Elvira Albert, Richard Bubel, Samir Genaim, Reiner Hähnle, and Guillermo Román-Díez. Verified resource guarantees for heap manipulating programs. In *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering*, FASE '12, pages 130–145, 2012.

[ABG12b] Diego Esteban Alonso-Blas and Samir Genaim. On the limits of the classical approach to cost analysis. In *Proceedings of the 19th International Conference on Static Analysis*, SAS '12, pages 405–421, 2012.

[ABH$^+$07] David Aspinall, Lennart Beringer, Martin Hofmann, Hans-Wolfgang Loidl, and Alberto Momigliano. A program logic for resources. *Theoretical Computer Science*, 389(3):411–445, December 2007.

[ADFG10] Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *Proceedings of the 17th International Conference on Static Analysis*, SAS '10, pages 117–133, 2010.

[Atk10] Robert Atkey. Amortised resource analysis with separation logic. In *Proceedings of the*

19th European Conference on Programming Languages and Systems, ESOP '10, pages 85–103, 2010.

[BBFF$^+$12] Ricardo Bedin França, Sandrine Blazy, Denis Favre-Felix, Xavier Leroy, Marc Pantel, and Jean Souyris. Formally verified optimizing compilation in ACG-based flight control software. In *Proceedings of the 3rd European Congress on Embedded Real Time Software and Systems*, ERTS '12, 2012.

[BDP01] Dennis Brylow, Niels Damgaard, and Jens Palsberg. Static checking of interrupt-driven software. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 47–56, 2001.

[BEF$^+$14] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. Alternating runtime and size complexity analysis of integer programs. In *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '14, pages 140–155, 2014.

[BEF$^+$16] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. Analyzing runtime and size complexity of integer programs. *ACM Transactions on Programming Languages and Systems*, 38(4):13:1–13:50, August 2016.

[BFGY08] Víctor Braberman, Federico Fernández, Diego Garbervetsky, and Sergio Yovine. Parametric prediction of heap memory requirements. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, pages 141–150, 2008.

[BHHK10] Régis Blanc, Thomas A. Henzinger, Thibaud Hottelier, and Laura Kovács. Abc: Algebraic bound computation for loops. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR '10, pages 103–118, 2010.

[BMP14] Sandrine Blazy, André Maroneze, and David Pichardie. Formal verification of loop bound estimation for wcet analysis. In *Revised Selected Papers of the 5th International Conference on Verified Software: Theories, Tools, Experiments - Volume 8164*, VSTTE 2013, pages 281–303, 2014.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceed-*

ings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL '77, pages 238–252, 1977.

[Chl07] Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 54–65, 2007.

[CHRS14] Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. End-to-end verification of stack-space bounds for c programs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 270–281, 2014.

[CHRS17] Quentin Carbonneaux, Jan Hoffmann, Thomas Reps, and Zhong Shao. Automated resource analysis with Coq proof objects. In *Proceedings of the 29th International Conference on Computer Aided Verification*, CAV '17, 2017.

[CHS15] Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. Compositional certified resource bounds. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 467–478, 2015.

[CNPQ08] Wei-Ngan Chin, Huu Hai Nguyen, Corneliu Popeea, and Shengchao Qin. Analysing memory resource bounds for low-level programs. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, pages 151–160, 2008.

[Coq] Coq Development Team. Reference manual (v8.6). `https://coq.inria.fr/distrib/current/refman/index.html`. Accessed in May 2017.

[CW00] Karl Crary and Stephanie Weirich. Resource bound certification. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '00, pages 184–198, 2000.

[DH17] Ankush Das and Jan Hoffmann. ML for ML: learning cost semantics by experiment. In *Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '17, pages 190–207, 2017.

[Exp14] Express Logic, Inc. Helping you avoid stack overflow crashes! White Paper, 2014.

[FHF07] Christian Ferdinand, Reinhold Heckmann, and Bärbel Franzen. Static Memory and Timing Analysis of Embedded Systems Code. In *3rd European Symposium on Verification and Validation of Software Systems*, VVSS '07, 2007.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In *Proceedings of Symposia in Applied Mathematics*, volume 19, pages 19–32, 1967.

[FM16] Antonio Flores-Montoya. Upper and lower amortized cost bounds of programs expressed as cost relations. In *Proceedings of the 21st International Symposium on Formal Methods*, FM '16, pages 254–273, 2016.

[FMH14] Antonio Flores-Montoya and Reiner Hähnle. Resource analysis of complex programs with cost equations. In *Proceedings of the 12th Asian Symposium on Programming Languages and Systems*, APLAS '14, pages 275–295, 2014.

[Gan00] Jack G. Ganssle. *The Art of Designing Embedded Systems*. Newnes, 2000.

[GJK09] Sumit Gulwani, Sagar Jain, and Eric Koskinen. Control-flow refinement and progress invariants for bound analysis. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 375–385, 2009.

[GKR+15] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 595–608, 2015.

[GMC09] 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, pages 127–139, 2009.

[GRE+01] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the 4th IEEE International Workshop on Workload Characterization*, WWC '01, pages 3–14, 2001.

[GZ10] Sumit Gulwani and Florian Zuleger. The reachability-bound problem. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 292–304, 2010.

[HAH12] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. *ACM Transactions on Programming Languages and Systems*, 34(3):14:1–14:62, November 2012.

[HDW17] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. Towards automatic resource bound analysis for ocaml. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 359–373, 2017.

[HH10] Jan Hoffmann and Martin Hofmann. Amortized resource analysis with polymorphic recursion and partial big-step operational semantics. In *Proceedings of the 8th Asian Symposium on Programming Languages and Systems*, APLAS '10, pages 172–187, 2010.

[HJ03] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 185–197, 2003.

[HJ06] Martin Hofmann and Steffen Jost. Type-based amortised heap-space analysis. In *Proceedings of the 15th European Conference on Programming Languages and Systems*, ESOP '06, pages 22–37, 2006.

[HM03] Kevin Hammond and Greg Michaelson. Hume: A domain-specific language for real-time embedded systems. In *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering*, GPCE '03, pages 37–56, 2003.

[HMS13] Jan Hoffmann, Michael Marmar, and Zhong Shao. Quantitative reasoning for proving lock-freedom. In *Proceedings of the 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '13, pages 124–133, 2013.

[Hoa69] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[HS14] Jan Hoffmann and Zhong Shao. Type-based amortized resource analysis with integers and arrays. In Michael Codish and Eijiro Sumii, editors, *Proceedings of the 12th*

International Symposium on Functional and Logic Programming, FLOPS '14, pages 152–168, 2014.

[JHLH10] Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. Static determination of quantitative resource usage for higher-order programs. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 223–236, 2010.

[JLB+15] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A formally-verified c static analyzer. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 247–259, 2015.

[JLH+09] Steffen Jost, Hans-Wolfgang Loidl, Kevin Hammond, Norman Scaife, and Martin Hofmann. "Carbon credits" for resource-bounded computations using amortised analysis. In *Proceedings of the 2nd World Congress on Formal Methods*, FM '09, pages 354–369, 2009.

[KN06] Gerwin Klein and Tobias Nipkow. A machine-checked model for a java-like language, virtual machine, and compiler. *ACM Transactions on Programming Languages and Systems*, 28(4):619–695, July 2006.

[Kre14] Robbert Krebbers. An operational and axiomatic semantics for non-determinism and sequence points in C. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 101–112, 2014.

[LB08] Xavier Leroy and Sandrine Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41(1):1–31, July 2008.

[Lei10] Rustan Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, April 2010.

[Ler06] Xavier Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 42–54, 2006.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, July 2009.

[Nip02] Tobias Nipkow. Hoare Logics in Isabelle/HOL. In *Proof and System-Reliability*, volume 62 of *NATO Science Series*, pages 341–367. Springer, 2002.

[RRW05] John Regehr, Alastair Reid, and Kirk Webb. Eliminating stack overflow by abstract interpretation. *ACM Transactions on Embedded Computing Systems*, 4(4):751–778, November 2005.

[Sha10] Zhong Shao. Certified software. *Communications of the ACM*, 53(12):56–66, December 2010.

[SVZN$^+$13] Jaroslav Sevcik, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *Journal of the ACM*, 60(3):22, 2013.

[SZV14] Moritz Sinn, Florian Zuleger, and Helmut Veith. A simple and scalable static analysis for bound analysis and amortized complexity analysis. In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*, CAV '14, pages 745–761, 2014.

[SZV15] Moritz Sinn, Florian Zuleger, and Helmut Veith. Difference constraints: An adequate abstraction for complexity analysis of imperative programs. In *Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design*, FMCAD '15, pages 144–151, 2015.

[Tar75] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, April 1975.

[Tar85] Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.

[TL09] Jean-Baptiste Tristan and Xavier Leroy. Verified validation of lazy code motion. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 316–326, 2009.

[TL10] Jean-Baptiste Tristan and Xavier Leroy. A simple, verified validator for software pipelining. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 83–92, 2010.

[WEE$^+$08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—Overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):36:1–36:53, May 2008.

[ZGSV11] Florian Zuleger, Sumit Gulwani, Moritz Sinn, and Helmut Veith. Bound analysis of imperative programs with the size-change abstraction. In *Proceedings of the 18th International Conference on Static Analysis*, SAS '11, pages 280–297, 2011.