Technical Report YALEU/DCS/TR-1487

End-to-End Verification of Stack-Space Bounds for C Programs

Quentin Carbonneaux Jan Hoffmann Tahina Ramananandro Zhong Shao
Yale University

Abstract

Verified compilers guarantee the preservation of semantic properties and thus enable formal verification of programs at the source level. However, important quantitative properties such as memory and time usage still have to be verified at the machine level where interactive proofs tend to be more tedious and automation is more challenging.

This article describes a framework that enables the formal verification of stack-space bounds of compiled machine code at the C level. It consists of a verified CompCert-based compiler that preserves quantitative properties, a verified quantitative program logic for interactive stack-bound development, and a verified stack analyzer that automatically derives stack bounds during compilation.

The framework is based on event traces that record function calls and returns. The source language is CompCert Clight and the target language is x86 assembly. The compiler is implemented in the Coq Proof Assistant and it is proved that crucial properties of event traces are preserved during compilation. A novel quantitative Hoare logic is developed to verify stack-space bounds at the CompCert Clight level. The quantitative logic is implemented in Coq and proved sound with respect to event traces generated by the small-step semantics of CompCert Clight. Stack-space bounds can be proved at the source level without taking into account low-level details that depend on the implementation of the compiler. The compiler fills in these low-level details during compilation and generates a concrete stack-space bound that applies to the produced machine code. The verified stack analyzer is guaranteed to automatically derive bounds for code with non-recursive functions. It generates a derivation in the quantitative logic to ensure soundness as well as interoperability with interactively developed stack bounds.

In an experimental evaluation, the developed framework is used to obtain verified stack-space bounds for micro benchmarks as well as real system code. The examples include the verified operating-system kernel CertiKOS, parts of the MiBench embedded benchmark suite, and programs from the CompCert benchmarks. The derived bounds are close to the measured stack-space usage of executions of the compiled programs on a Linux x86 system.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; D.3.3 [*Programming Languages*]: Processors—Compilers

General Terms Verification, Reliability

Keywords Formal Verification, Compiler Construction, Program Logics, Stack-Space Bounds, Quantitative Verification

1. Introduction

It has been shown that formal verification can greatly improve software quality [25, 38, 35]. Consequently, formal verification is the subject of ongoing research and there exist sophisticated tools that can verify important program properties automatically. However, the most interesting program properties are undecidable and user interaction is therefore inevitable in formal verification.

If a software system is (partly or entirely) developed in a high-level language then the question arises on which language level the verification should be carried out. Verification at the source level has the advantage that a developer can interact with the verification tools using the code she has developed. This is beneficial because the compiled code can substantially differ from the source code and low-level code is harder to understand. Moreover, even fully automatic tools profit from the control-flow information and the structure that is available at higher abstraction layers. The disadvantage of verification at the source level is that tools such as compilers have to be part of the trusted computing base and that the verified properties are not directly guaranteed for the code that is executed on the system.

Formally verified compilers [24, 11] such as the CompCert C Compiler [27] guarantee that certain program properties of the source programs are preserved during compilation. As a result, CompCert enables source-level verification of the preserved properties of the compiled code without increasing the size of the trusted computing base.[1] In fact, this has been one of the main motivations for the development of CompCert [27]. However, important quantitative properties such as memory and time consumption are not modeled nor preserved by CompCert and other verified compilers [24, 11]. Such quantitative properties are nevertheless crucial in the verification of safety-critical embedded systems. For example, the DO-178C standard, which is used by in the avionics industry and by regulatory authorities, requires verification activities to show that a program in executable form complies with its requirements on stack usage and worst-case execution time (WCET) [30].

Quantitative program requirements such as stack usage and WCET are usually directly checked at the machine or assembly-code level "since only at this level is all necessary information available" [37]. For stack-space bounds there exist commercial abstract interpretation–based tools—such as Absint's *StackAnalyzer* [14]—that operate directly on machine code. While such tools can derive many simple bounds automatically, they rely on user annotations in the machine code to obtain bounds for more involved programs. The produced bounds are usually not parametric in the input, and the analysis is not modular and only applies to specific hardware platforms. Additionally, the used analysis tools rely on the correctness of the user annotations and are not formally verified.

In this article, we 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" [13]. Moreover, stack-memory is the only dynamically allocated memory in many embedded systems and the stack usage depends on the implementation of the compiler. While we focus exclusively on stack bounds in this article, our framework

---

[1] If we assume that all verification is carried out with the same trusted base.

is developed with other quantitative resources in mind. Many of the developed techniques can be applied to derive bounds for resources such as heap memory or clock cycles. However, for clock-cycle bounds there is a lot of additional work to be done that is beyond the scope of this article (e.g., developing a formal model for hardware caches and instruction pipelines).

The main innovation of our framework is that it enables the formal verification of stack bounds for compiled x86 assembly code at the C level. To gain the benefits of source-level verification without the entailed disadvantages, we have to deal with three main challenges.

1. We have to model the stack consumption of programs at the C level and we have to formally prove that our model is consistent with the stack consumption of the compiled code.
2. We have to design and implement a C-level verification mechanism that allows users to derive parametric stack-usage bounds in an interactive and flexible way.
3. We have to minimize user interaction during the verification to enable the verification of large systems.

To meet Challenge 1, we use event traces and verified compilation. Our starting point is the CompCert C Compiler. It relies on event traces to prove that a compiled program is a refinement of the source program. We extend event traces with events for function calls and returns and define a *weight* for event traces. The weight describes the stack-space consumption of one program execution as a function of a cost metric which assigns a cost to individual call and return events. The idea is that a user or an (semi) automatic analysis tool derives bounds on the weights of event traces that depend on the stack-frame sizes of the program functions. During compilation the compiler produces a specific cost metric that guarantees that the weight of an event trace computed with this metric is an upper bound on the stack-space usage of the compiled assembly program which produces this trace. As a result, we derive a verified upper bound if we instantiate the derived memory bound with the cost metric produced by the compiler.

We implemented the extended event traces for full CompCert C and all intermediate languages down to x86 assembly in Coq. We extended CompCert's soundness theorem to take into account the weights of traces. In addition to CompCert's refinement theorem for the original event traces, we prove that compiled programs produce extended event traces whose weights are less or equal to the weights of the traces at the source level. This means that we allow reordering or deletion of call and return events as long the weight of the trace is reduced or unchanged. To relate the weight of traces to the execution on a system with finite stack space, we modified the CompCert x86 assembly semantics into a more realistic x86 assembly that features a finite stack, and reimplemented the assembly generation pass of CompCert to our new x86 assembly semantics.

To meet Challenge 2, we have developed and implemented a novel quantitative Hoare logic for CompCert Clight in Coq. To account for memory consumption, the assertions of the logic generalize the usual boolean-valued assertions of Hoare logic. Instead of the classic *true*, our quantitative assertions return a natural number that indicates the amount of memory that is needed to execute the program. The boolean *false* is represented by ∞ and indicates that there are no guarantees provided for the future execution.

We proved the soundness of our quantitative Hoare logic with respect to Clight and CompCert's continuation-based small-step semantics. The soundness theorem states that Hoare triples that are derived with our inference rules describe sound bounds on the weights of traces. The logic can be used for interactive stack-bound development or as a backend for verified static analysis tools.

For clarity, we do not prove the safety of programs and simply assume that this is done using a different tool such as Appel's separation logic for Clight [3]. It would be possible to integrate our logic into a separation logic for safety proofs. This would however diminish the deployability of the quantitative logic as a backend for static stack-bound analysis tools since they would be required to also prove memory safety.

To meet Challenge 3, we implemented an automatic stack analyzer for C programs. To verify the soundness of the stack analyzer each successful run generates a derivation in the quantitative Hoare logic. This does not only simplify the verification but also allows interoperability with stack bounds that have been interactively developed in the logic or derived by some other static analysis. Conceptually, our stack analyzer is rather simple but we have proved that it derives sound bounds for programs without recursion and function pointers. This is already sufficient for many programs used in embedded systems. Using our automatic analysis we have created 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()`.

We have successfully used our quantitative Hoare logic, the extended C compiler, and the automatic stack analyzer to verify end-to-end memory bounds for micro benchmarks and system software. Our main example is the CertiKOS [15] operating system kernel that is currently under development at Yale. Our automatic analyzer finds stack bounds for all functions in the simplified development version of CertiKOS that is currently verified. Other examples are taken from Leroy's CompCert benchmarks and the MiBench embedded benchmark suite [17]. To evaluate the quality of the verified stack-space bounds, we experimentally compared the automatically and manually verified bounds with the actual stack-space consumption during the execution of the compiled C programs. Our experiments indicate that both the manually and automatically derived bounds over-approximate the stack usage by exactly four bytes. More details can be found in Section 6.

In summary, we make the following contributions.

- We introduce a methodology that uses cost metrics to link event traces to resource consumption. This approach enables us to link source-level code to the resource consumption of compiled target-level code.
- We develop a novel quantitative Hoare logic to reason about the resource consumption of programs at the source level. We have formally verified the soundness of the logic with respect to CompCert Clight in Coq.
- We introduce *Quantitative CompCert*, a modified version of the verified CompCert C Compiler, in which parametric stack bounds are preserved during compilation. Furthermore, Quantitative CompCert creates a cost metric so that the instantiation of the bounds with the metric forms an upper bound on the memory consumption of the compiled code.
- We have implemented and verified an automatic stack analyzer that is guaranteed to compute stack bounds for non-recursive programs.
- We have evaluated the practicability of our framework with experiments using micro benchmarks and system code.

The complete Coq development and the implemented tools are well documented and publically available on the authors' websites. The *PLDI Artifact Evaluation Committee* reproduced samples of our experiments and tested the implemented tools on additional programs. The reviewers unanimously stated that our implementation *exceeded their expectations.*

```
typedef unsigned int u32;
u32 a[ALEN];
u32 seed = SEED;

u32 search(u32 elem, u32 beg, u32 end) {
        u32 mid = beg + (end-beg) / 2;

        if (end-beg <= 1) return beg;

        if (a[mid] > elem) end = mid;
        else beg = mid;

        return search(elem, beg, end);
}
u32 random() {
        seed = (seed * 1664525) + 1013904223;
        return seed;
}
void init() {
        u32 i, rnd, prev = 0;

        for (i=0; i<ALEN; i++) {
                rnd = random();
                a[i] = prev + rnd % 17;
                prev = a[i];
        }
}
int main() {
        u32 idx, elem;
        init();
        elem = random() % (17 * ALEN);
        idx = search(elem, 0, ALEN);
        return a[idx] == elem;
}
```

Figure 1. An illustrative example for static stack-bound computation. Constant stack bounds for the non-recursive functions are derived automatically. The logarithmic bound for the function search is derived with a hand-crafted proof in our quantitative Hoare logic.

2. An Illustrative Example

In this section, we sketch the verification of stack-space bounds for an example program in our framework. Figure 1 shows a C program with two integer parameters: ALEN and SEED.

This program will fill an array of size ALEN with an increasing sequence of pseudo random integers and search through it. The random numbers are created by a linear congruential generator initialized by the SEED parameter. The search procedure used is a binary search implemented in the recursive function search.

Our goal is to derive stack bounds for the compiled x86 assembly code of the program that are verified with respect to our accurate x86 model in Coq. The first step is to create an abstract syntax tree of the code in Coq. This can be done automatically, for instance by using CompCert's parsing mechanism. The second step is to use our quantitative Hoare logic to prove bounds on the function calls that are performed when executing main.

To relate function calls and returns at different abstraction levels during compilation we use call and return events. For instance, an execution of main could produce the following trace.

$$\begin{gathered}\mathsf{call}(\mathsf{main}), \mathsf{call}(\mathsf{init}), \mathsf{call}(\mathsf{random}), \mathsf{ret}(\mathsf{random}), \mathsf{ret}(\mathsf{init}),\\ \mathsf{call}(\mathsf{search}), \mathsf{call}(\mathsf{search}), \mathsf{ret}(\mathsf{search}), \mathsf{ret}(\mathsf{search}), \mathsf{ret}(\mathsf{main})\end{gathered}$$

From such a trace and a metric M that maps each function name in the program to its stack-frame size, we can obtain the stack usage of the execution that produced the trace. For the previous example trace, we can for instance derive the following stack usage.

$$M(\mathsf{main}) + \max\{M(\mathsf{init}) + M(\mathsf{random}), 2 \cdot M(\mathsf{search})\}$$

In classical Hoare logic, assertions map program states to Booleans. In our quantitative Hoare logic assertions map program states to non-negative numbers. Intuitively, the meaning of a quantitative Hoare triple $\{P\}\, S\, \{Q\}$ is the following. For every program state σ, $P(\sigma)$ is a upper bound on the stack consumption of the statement S started in state σ. Furthermore, Q describes the stack space that has become available after the execution, as a function of the final program state. This is similar as in type systems and program logics for amortized resource analysis [21, 5].

We implemented a function in Coq that automatically computes a derivation in the quantitative logic for a program without recursive functions. Using this automatic stack analyzer, we derive for instance the following triple for the function call init().

$$\{M(\mathsf{init}) + M(\mathsf{random})\}\, \mathsf{init}()\, \{M(\mathsf{init}) + M(\mathsf{random})\}$$

For functions making use of recursion such as search, we derive a quantitative triple interactively using Coq. For search we derive

$$\begin{aligned}&\{\mathrm{L}(\mathsf{end} - \mathsf{beg})\}\, \mathsf{search}(\mathsf{elem}, \mathsf{beg}, \mathsf{end})\, \{\mathrm{L}(\mathsf{end} - \mathsf{beg})\}\\ &\text{where } \mathrm{L}(\Delta) = \mathrm{M}(\mathsf{search}) \cdot (2 + \log_2(\Delta))\ .\end{aligned}$$

Since the mathematical $\log_2$ function is undefined on non-positive values, we take as convention that $\log_2(\Delta) = +\infty$ when $\Delta < 0$ and $\log_2(0) = 0$. This trick allows us to simulate a logical precondition stating that beg must be lower or equal to end before calling search.

For main we combine the previous results and derive the bound

$$\begin{aligned}&\{M(\mathsf{main}) + N\}\, \mathsf{main}()\, \{M(\mathsf{main}) + N\}\\ &\text{where } N = \max(M(\mathsf{init}) + M(\mathsf{random}), \mathrm{L}(\mathsf{ALEN}))\ .\end{aligned}$$

To be able to derive this bound on the main function we have to require that $0 < \mathsf{ALEN} \leqslant 2^{32} - 1$, in the Coq development this is stated as a section hypothesis which will later be instantiated when ALEN is chosen by the user before compiling.

The third and final step in the derivation of the stack bounds is to compile the program with Quantitative CompCert, our modified CompCert C Compiler. The compiler produces x86 assembly code and a concrete metric M_0. It follows from CompCert's correctness theorem that the compiled code is a semantic refinement of our source program. In addition, we have formally verified that the metric M_0 correctly relates the abstractly defined stack consumption—using the event traces—to the actual stack consumption in our abstract x86 machine. Moreover, we have verified that applying M_0 to the preconditions in the triples of the quantitative Hoare logic results in sound stack bounds on the x86 machine. The final bounds that we obtain by compilation for our examples are for instance 32 bytes for init() and $112 + 40 \cdot \log_2(\mathsf{ALEN})$ bytes for main().

3. Quantitative CompCert: Verified Stack-Aware Compilation

In this section, we introduce our new technique for verifying *quantitative compiler correctness* and its implementation in Quantitative CompCert. We focus on stack-space usage but believe that similar techniques can be used to bound the time and heap-space requirements of programs.

Our development is highly influenced by the design of CompCert [27], a verified compiler for the C language. CompCert C accepts most of the ISO-C-90 language and produces machine code for the IA32 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.

3.1 Quantitative Compiler Correctness

In the following, we show how to extend trace-based compiler-correctness proofs to also cover stack-space consumption. In short, our 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.

We now formalize and elaborate on these points.

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.

Memory events	$\mu ::= \mathsf{call}(x) \mid \mathsf{ret}(x)$

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.

Events	$e ::= \nu \mid \mu$
Finite event traces	$t ::= \epsilon \mid e \cdot t$
Coinductive event traces	$T ::= \epsilon \mid e \cdot T$
Behaviors	$B ::= \mathsf{conv}(t,n) \mid \mathsf{div}(T) \mid \mathsf{fail}(t)$

We 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.

Weights of Behaviors For a behavior B, we define the set of finite prefix traces $\mathit{prefs}(B)$ 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 resource metric

$$M : \mathcal{E} \to \mathbb{Z}$$

that maps events to integers (bytes). The purpose of the metric in our work is to relate memory events to the sizes of the stack frames of functions in the target code. To this end, we only use stack metrics, that is, metrics M such that for all functions f and for all external functions g

$$0 \leqslant M(\mathsf{call}(f)) = -M(\mathsf{ret}(f)) \quad \text{and} \quad M(g(\vec{v} \mapsto v)) = 0 \; .$$

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.

Before, we define the weight, we first inductively define the valuation $V_M(t)$ of a finite trace t.

$$\begin{aligned} V_M(\epsilon) &= 0 \\ V_M(\alpha \cdot t) &= V_M(t) + M(\alpha) \end{aligned}$$

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 1. *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 2 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, we define $n + \infty = \infty$.

Lemma 2. *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 $W_M(t \cdot T) \geqslant W_M(t)$. □

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

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

Under a stack metric M the weight of t is $W_M(t) = 4 \cdot M(\mathsf{call}(f))$.

In the next example, we 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.

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

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

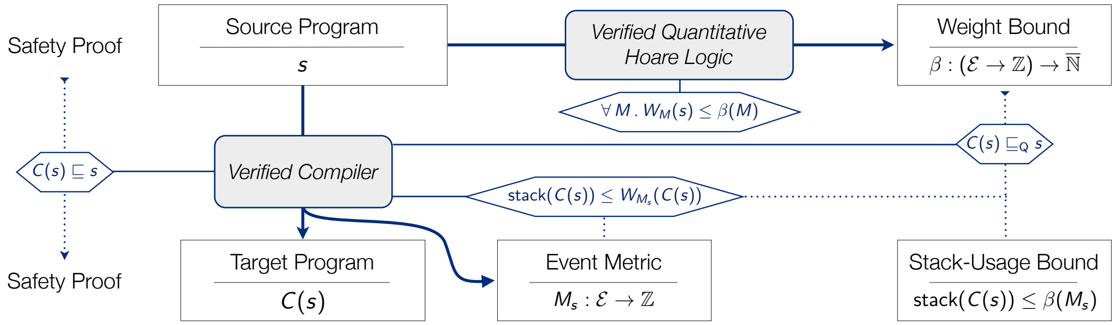

Figure 2. Overview of our quantitative verification framework. We write $W_M(s) = \sup\{W_M(B) \mid B \in [\![s]\!]\}$ for the weight of the program s under the metric M. Furthermore, we write $\mathsf{stack}(s)$ for the smallest number n so that s runs without stack overflow if executed with a stack of size n. All metrics in the figure are stack metrics.

Quantitative Refinement For our description of quantitative refinements we 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 we define the pruned behavior as the behavior $\overline{B}$ that results from deleting all memory events ($\mathsf{call}(x)$ or $\mathsf{ret}(x)$) from B. We first inductively define pruned finite traces as follows. As always, ν denotes an I/O event and μ denotes a memory event.

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

Similarly, we coinductively define pruning for possibly infinite traces.

$$\overline{\mu_1 \cdot \dots \cdot \mu_n \cdot \epsilon} = \epsilon$$
$$\overline{\mu_1 \cdot \dots \cdot \mu_n \cdot \nu \cdot T} = \nu \cdot \overline{T}$$
$$\overline{\mu_1 \cdot \dots \cdot \mu_n \cdot \dots} = \epsilon$$

Finally, we define pruned behaviors as follows.

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

In CompCert, compiler correctness is formalized through the notion of *refinement*. A (target) program s' is a refinement of a (source) program s, written $s' \prec s$, 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 $t \in \mathit{prefs}(\overline{B})$.[2] 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 $s' \prec_Q s$ 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 $\overline{t} \in \mathit{prefs}(\overline{B})$.

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

Verifying Stack-Space Usage Figure 2 shows how we verify the stack-space usage of a program in our framework. First, we prove a bound $\beta : (\mathcal{E} \to \mathbb{Z}) \to \mathbb{N}$ on the weights of the event traces that a program can produce. This bound is parameterized by an event metric $M : \mathcal{E} \to \mathbb{Z}$. 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 2) 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 (see explanation in Section 3.2).

3.2 Verification and Implementation

We implemented the verification framework that we outlined in Section 3.1 for the CompCert C compiler using the proof assistant Coq. The verification consists of about 5000 lines of Coq code that we integrated into CompCert 1.13 (which originally consists of about 90000 lines of Coq code) to obtain a modified version that we call *Quantitative CompCert*. The source-level language is CompCert C 1.13 and the target language is CompCert x86 assembly.

Overview of CompCert CompCert 1.13 is decomposed into 20 passes between 11 intermediate languages. Here, we describe a subset of these passes.

1. First, 3 passes compile Compcert C to Clight, a subset of C where expressions have no side effects.
4. Then, 2 passes translate Clight to Cminor, a C-like language where addressable local variables, formerly independent of each other, are grouped together into a single per-function call memory region called the *stack frame*. In contrast, temporary results are stored in an unbounded number of non-addressable local variables, or *pseudo-registers*.
6. Then, 2 passes compile Cminor code to RTL (Register Transfer Language), similar in principle to Cminor, but more amenable to

---

[2] In fact, it is enough to prove that $\overline{B'} \sim \overline{B}$ (bisimilarity of infinite traces), because $[\![s]\!]$ is closed by bisimilarity.

optimizations due to its 3-address instructions and control-flow–graph shape.

8. Then, 5 distinct[3] optimizations are performed on RTL.
13. Then, 1 pass of register allocation, producing LTL (Location Transfer Language) code actually tags each such location to distinguish between true machine registers (of which there are only a fixed number), and "virtual" stack slots, which are still pseudo-register-like locations separated from the memory. However, values of machine registers are still proper to each function call.
14. Then, 3 passes linearize the code down to the LTLin language.
17. Then, 2 passes introduce reloading instructions to turn LTLin into Linear, a language in which the values of machine registers become common to the whole execution of the program.
19. Then, 1 pass compiles code from Linear into Mach, where the "virtual" stack slots are actually turned into true memory accesses, thus making spilling and reloading concrete.
20. Finally, 1 pass turns CFG-based Mach code into x86 assembly code.

Each of these passes is proved correct with respect to the CompCert trace refinement relation (see Section 3.1). Basically, if the source program does not go wrong, then all traces of the target program are traces of the source program. Each compiler pass and its proof are independent of other passes.

The problem: stack consumption in CompCert For each language starting from Cminor, including the CompCert x86 assembly language, each function call allocates a memory region—called the *stack frame*—to store its addressable local variables, and (starting from Mach) the spilling locations and the function arguments to handle the calling conventions. This stack frame is freed upon function return. However, even though each stack frame is finite, there may well be an unbounded number of such allocations, even for embedded function calls. Indeed, in CompCert, allocating a stack frame always succeeds, thus CompCert does not model *stack overflow*, and the following CompCert C function:

```
void f (int* pi) { int i = 0; f(&i); return; }
```

has a valid diverging semantics without failing or overflowing. However it allocates an unbounded number of stack frames.

Our solution: Quantitative CompCert In Quantitative CompCert, we overcome this issue by modifying the semantics of the target assembly language. We preallocate a finite memory region for the whole stack, into which all stack frames shall be merged together during the execution instead of being individually allocated.

By contrast, we still want the source and intermediate languages to allocate an individual stack frame per function call. First, we want to change CompCert only if necessary so as to still support all features of CompCert C. Second, it would not be very meaningful to introduce a finite stack at a high language level since it is unclear how to model stack sizes. The only major change we bring to those languages is to introduce our call and return events into the trace.

As shown in Figure 3, this leads us to split CompCert into two parts. In the first part, we compile CompCert C down to Mach by adapting the proofs of existing passes to quantitative refinement. In the second part, we perform two passes to merge all stack frames together. The key point of our work is that this second part will require the Mach traces to not stack overflow, which justifies the use of quantitative refinement for the first part.

The source code of our complete Coq development is publically available [9]. In the following, we will highlight some of the challenges that we faced in the implementation.

Quantitative Refinement In the first part of the compiler, from CompCert C down to Mach, we add call and return events to the semantics of each language, at the level of each function call and return (as described in Section 3.1). This change is uniform in all languages between CompCert C to Mach: indeed, in each small-step operational semantics, there is only one rule responsible for internal function call. Until Cminor, due to functions returning void and implicit returns when reaching the end of a function, there are three rules responsible for internal function return; starting from RTL, there is only one such rule.

Then, thanks to these changes, we support all of CompCert 1.13 passes except two optional optimizations (see Section 3.3), and, with no significant changes to the proofs, we prove that they exactly preserve traces with function call events.

Generation of Target Cost Metric The semantics of CompCert C allocates a separate memory region for each addressable local variable. In Mach, all those variables as well as the spilling locations, the function arguments, and the return address are stored in a stack frame. Actually, the stack frame of a Mach function call is completely laid out, so that no additional memory is necessary when generating the CompCert x86 assembly code. This means that, at the level of Mach, we already know the stack size necessary for a function call (thanks to the fact that the original CompCert does not support some C features, see 3.3): for a given function, this size is constant and does not depend on the arguments nor the input. So, we can use the sizes of Mach stack frames as cost metric for functions to accurately estimate stack bounds at the source level: the weight of a trace with such instantiation actually models the exact stack consumption of the corresponding execution at the level of Mach. Consequently, we modify CompCert in such a way that, in addition to the assembly code, it returns the mapping of Mach stack frame sizes for each function.

In fact, to cope with the generation of assembly code (see below), we slightly modified the Linear-to-Mach stack-layout pass, to introduce the return address only at the next pass. So, if the Mach stack frame size of a function f excluding the space for return address is $\mathsf{SF}(f)$, then the actual cost metric is $M(f) = \mathsf{SF}(f) + 4$ (as we focus on x86 32-bit machines), taking advantage of the fact that CompCert already computes $\mathsf{SF}(f)$ in such a way that $M(f)$ is a multiple of 8 (or 16, if strong alignment is required by the user), to keep the actual contents of stack frames correctly aligned.

To sum up so far, our modified CompCert ensures that CompCert C code compilation down to Mach code is correct with respect to quantitative refinement. Then, by instantiating the cost metric to the sizes of Mach stack frames, it follows that the actual stack consumption of the produced Mach code is indeed lower than the bound computed at the level of CompCert C.

Generation of Assembly Code Recall that CompCert x86 assembly language is not realistic enough as it does not prevent from allocating an infinite number of stack frames. Our goal, as one of our main applications of our quantitative refinement, is to make the CompCert x86 assembly language more realistic by having it model a contiguous finite stack that is preallocated at the beginning of the program. The semantics (but not the syntax) of our new CompCert x86 assembly is parameterized by the size $sz + 4$ [4] of the whole stack (provided, in most cases, by the host operating system). We call this new x86 semantics ASM_{sz}. We design it in such a way that an execution goes wrong if the program tries to access more than sz

---

[3] In fact, some of those optimizations such as constant propagation or common subexpression elimination are performed more than once.

[4] sz is the stack size actually consumed by the program starting from `main`, but we have to account for the return address of the "caller" of `main`.

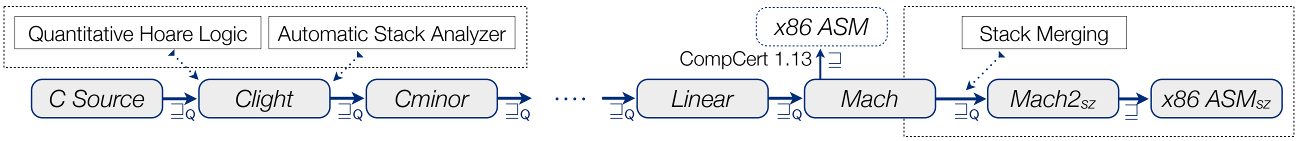

Figure 3. Quantitative CompCert, our modified stack-aware CompCert C compiler. We replace CompCert's x86 assembly with the more realistic x86 assembly semantics ASM_{sz} with finite stack. Pseudo assembly instructions are not needed anymore.

bytes of stack. In other words, stack overflow becomes possible in ASM_{sz}.

Because the notion of function call is no longer relevant (there is no "control stack"), we lose the ability to extend this semantics with call and return events. So, rather than quantitative refinement, we are actually interested in whether a CompCert C source program can run on ASM_{sz} without going wrong because of stack overflow. The correctness of our Quantitative CompCert compiler is formalized by the following theorem.

Theorem 1. *Let $sz + 4 \in [4, 2^{32})$ be the size of the whole target stack. Consider a CompCert C source program S and assume the following:*

1. *S does not go wrong in the ordinary setting of unbounded stack space, that is, $\nexists t, \mathsf{fail}(t) \in [\![S]\!]$.*
2. *Quantitative CompCert produces a Mach intermediate target code I, with the sizes of stack frames*[5] *SF and the subsequent cost metric $M(f) = \mathsf{SF}(f) + 4$.*
3. *The stack bounds of S inferred at the source level and are lower than sz under the Match cost metric M: $\forall B \in [\![S]\!], W_M(B) \leqslant sz$.*
4. *From I, our compiler produces a target assembly code T.*

Then, when run in ASM_{sz}, T refines S in the sense of CompCert: $\forall B' \in [\![T]\!]_{sz}, \exists B \in [\![S]\!], B' = \overline{B}$. In particular, T cannot go wrong and thus does not stack overflow.

It is important to first prove that S cannot go wrong in unbounded stack space. Indeed, the correctness of our assembly generation depends on the fact that the weights of Mach traces are lower than sz. If S were to have a wrong behavior $\mathsf{fail}(t)$ then I might actually have a behavior $t \cdot B$ whose weight could well exceed sz even though $W_M(\mathsf{fail}(t))$ does not. As each pass is proved independently of the others, it is not possible to track the behaviors of I that could potentially come from wrong behaviors of S, so they have to be excluded.

To explain our transformation in more detail, we first informally describe the CompCert memory model [29]. In CompCert, memory is not one contiguous array of bytes, but a (finite but unbounded) 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 $(b, o + \delta)$, so that CompCert guarantees that such arithmetics will never cross block boundaries. Moreover, once a memory block is freed, it is never reused. To ensure that, $\mathsf{NB}(m)$ gives the sequence number of the next block available for allocation, so that all blocks with sequence number at least $\mathsf{NB}(m)$ are not yet allocated in m, and pointers to them are dangling; allocating a block increases $\mathsf{NB}(m)$ by one.

Thanks to this memory model, CompCert makes it possible to allocate one block for each addressable local variable in CompCert C. By contrast, Cminor allocates only one block per function call, into which all those local variables are merged by the compilation passes from CompCert C to Cminor. This single memory block per function call actually corresponds to the 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* $\mathsf{Pallocframe}$ and $\mathsf{Pfreeframe}$ responsible of allocating and freeing the corresponding memory block, even though those pseudo-instructions are then turned into real x86 assembly instructions performing pointer arithmetics with the ESP stack pointer register. This latter transformation cannot be proved correct in CompCert, because pointer arithmetics cannot cross block boundaries in the CompCert memory model. Therefore this transformation is 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 block of size $sz + 4$ at the beginning of the program to hold the whole stack, and our assembly generation pass ensures that the value of ESP 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 ASM_{sz} assembly language.

As an interesting side effect, accessing the function arguments is now simpler in our assembly language. Indeed, in the x86 calling convention, a function has to look for its arguments in the stack frame of its caller. Because in the original CompCert, stack frames are independent memory blocks, it was necessary for the callee to have a pointer to the caller stack frame, called the *back link*, in its own stack frame. The callee could then access its arguments by one indirection through this back link. In our new ASM_{sz} assembly language, stack frames are no longer independent, so that the callee can access its arguments directly by pointer arithmetics within the whole stack block. Consequently, the back link is no longer necessary, and we removed it from the stack layout. Again, this is possible because the sizes of stack frames are constant and there are no dynamic stack allocation (`push`/`pop`, etc.). In other words, the code produced by CompCert does not make use of the EBP frame pointer, which basically stays constant across the execution of the whole program).

To formalize and prove this pass, we actually have to prove that we can merge the memory blocks corresponding to stack frames into one single memory block, through a memory transformation called a *memory injection* [29, 5.4]. It models the merging of several source blocks into one target block. This transformation ensures that it is possible to reuse the target memory locations of freed source stack frames for further new stack frames.

---

[5] In CompCert Mach, the syntax of a program p includes a finite map SF such that, for any function f defined in p, the operational semantics of Mach allocates a stack frame of $\mathsf{SF}(f)$ bytes whenever f is entered.

Then, rather than proving a correctness pass from CompCert assembly to ASM_{sz}, we chose to modify the Mach to assembly pass by splitting it into two passes through an intermediate Mach2$_{sz}$ language which is a reinterpretation of the semantics of a Mach control-flow graph but with the stack frames merged into a single stack frame. Indeed, our proof still needs to know about a "control stack", which still exists in Mach but no longer in assembly. The Mach-to-Mach2$_{sz}$ pass is not a translation pass, but a proof that the reinterpretation of the semantics of Mach into Mach2$_{sz}$ is sound if the weights of the traces of the Mach semantics of the control-flow graph are lower than sz.

The main parts of the proof of this pass are about function calls and returns, which take full advantage of the properties of memory injections. Thanks to the fact that we know that the weight of the whole traces of the source program are lower than sz, the compilation invariant of our proof ensures that the stack pointer is of the form (b, o), where b is the sequence number identifying the memory block corresponding to the whole stack. The offset o is such that $0 \leqslant o \leqslant sz$ and for any behavior B starting from the current execution state, $o - W_M(B) \geqslant 0$ (in x86 the stack grows from sz down to 0). In fact, if the program starts from the initial state and produces a finite trace t. Then the stack pointer is actually equal to $(b, sz - V_M(t))$.

Finally, the proof of assembly generation between Mach2$_{sz}$ and ASM$_{sz}$ brings no significant changes from the original Mach to CompCert x86 assembly except for the correctness of function call and return, where the pointer arithmetics are actually performed.

3.3 Limitations

Stack frame size Neither the original CompCert nor Quantitative CompCert do support variable stack-frame size: C features such as variable-length arrays or dynamic stack allocation (`alloca` special library functions) are not supported. Thus, the size of the stack frame of a Mach function can be computed statically, and can be used to define the cost metric of the program. Moreover, the subsequently produced assembly code does not need to use `push` or `pop`, so any change to the stack pointer is done only through pointer arithmetics.

Optional optimizations Quantitative CompCert currently does not support the following optional two optimization passes (that are present in the original CompCert): tail-call recognition and function inlining. Here we show how to deal with those passes (their implementation is underway):

- For tail-call recognition, we know that a sequence of tail calls from a caller function f into a callee function g which in turn tail calls a function h, actually produces $\mathsf{call}(f) \cdot \mathsf{ret}(f) \cdot \mathsf{call}(g) \cdot \mathsf{ret}(g) \cdot \mathsf{call}(h) \cdot \mathsf{ret}(h) \cdot \epsilon$ instead of $\mathsf{call}(f) \cdot \mathsf{call}(g) \cdot \mathsf{call}(h) \cdot \mathsf{ret}(h) \cdot \mathsf{ret}(g) \cdot \mathsf{ret}(f) \cdot \epsilon$. Thus, by accumulating the anticipated return on an auxiliary stack, we can match the reorderings of return events by coinductively designing a custom refinement relation on potentially infinite traces. In the relation, θ is a finite trace that collects return events.

  $$\begin{array}{rl} \epsilon \sqsubseteq_\theta \epsilon & \\ e \cdot T' \sqsubseteq_\theta e \cdot T & \text{if } T' \sqsubseteq_\theta T \\ \mathsf{ret}(f) \cdot \epsilon \sqsubseteq_\theta \epsilon & \\ \mathsf{ret}(f) \cdot e \cdot T' \sqsubseteq_\theta e \cdot T & \text{if } T' \sqsubseteq_{\mathsf{ret}(f)\cdot\theta} T \\ T' \sqsubseteq_{\mathsf{ret}(f)\cdot\theta} \mathsf{ret}(f) \cdot T & \text{if } T' \sqsubseteq_\theta T \end{array}$$

  It is easy to see that, if $T \sqsubseteq_\theta T'$, then $\overline{T} \sim \overline{T'}$, and for any finite trace θ that only has return events (so that $V_M(\theta) \leqslant 0$) and for any finite prefix t' of T', there is a finite prefix t of T such that $V_M(t') + V_M(\theta) \leqslant V_M(t)$, so $W_M(T') + V_M(\theta) \leqslant W_M(T)$. Thus, to prove quantitative refinement, it suffices to prove that, for any trace T' of the target language, there is a source trace T of the source program such that $T' \sqsubseteq_\epsilon T$.
- Similarly, for function inlining, we know that, when a function call is inlined, its corresponding $\mathsf{call}()$ and $\mathsf{ret}()$ events are removed *in matching pairs*, so we can design a similar refinement relation. However, we have to be careful because the target code may actually produce fewer events. So, to make the relation coinductively productive, we first design a transition relation $\leadsto$ to be applied only finitely many times in a row (even though it may be applied infinitely many times overall) to consume finitely many call and return events from the source trace without producing them on the target trace (due to function inlining):

  $$\begin{array}{rl} (\mathsf{call}(f) \cdot T, \theta) & \leadsto (T, \mathsf{call}(f) \cdot \theta) \\ (\mathsf{ret}(f) \cdot T, \mathsf{call}(f) \cdot \theta) & \leadsto (T, \theta) \end{array}$$

  Then we can design the following coinductive relation (where θ is finite), which is indeed productive:

  $$\begin{array}{rr} \epsilon \sqsubseteq_\theta \epsilon & \\ e \cdot T' \sqsubseteq_\theta t \cdot e \cdot T & \text{if } T' \sqsubseteq_{\theta'} T \\ & \text{and } (t \cdot e \cdot T, \theta) \leadsto^* (e \cdot T, \theta') \\ \epsilon \sqsubseteq_\theta T & \text{if } \epsilon \sqsubseteq_{\theta'} T' \\ & \text{and } (T, \theta) \leadsto^+ (T', \theta') \end{array}$$

  It is easy to see that, if $T \sqsubseteq_\theta T'$, then $\overline{T} \sim \overline{T'}$, and for any finite trace θ that only has call events (so that $V_M(\theta) \geqslant 0$) and for any finite prefix t' of T', there is a finite prefix t of T such that $V_M(t') - V_M(\theta) \leqslant V_M(t)$, so $W_M(T') - V_M(\theta) \leqslant W_M(T)$. Thus, to prove quantitative refinement, it suffices to prove that, for any trace T' of the target language, there is a source trace T of the source program such that $T' \sqsubseteq_\epsilon T$.

The main challenge is due to the simulation diagrams [28, 2.1] used to prove the passes. Indeed, CompCert uses *forward (or downward) simulation* diagrams for each pass[6]: each execution step in the source program corresponds to one or more steps in the target program. But refinement actually requires the converse, namely *backward (or upward) simulation*: each execution step of the target program has to be associated to one or more steps in the target program. Because RTL is deterministic up to input values generated by I/O external function call events, forward simulation diagrams can be turned to backward simulation diagrams [34, 4.6], but, whereas it is accurate for pruned traces, it is not clear whether it is still true if the traces between the two languages differ, which is the case for both tail-call recognition and function inlining, as we saw above.

4. Quantitative Hoare Logic for CompCert Clight

In this section, we describe the novel quantitative program logic for CompCert Clight. The logic has been formalized and proved sound using Coq. At some points, we simplify the presented logic in comparison to the implemented version to discuss general ideas instead of technical details. Some of our design decisions have been influenced by the development of the verified operating-system kernel CertiKOS [15], which has been our first main application of the quantitative program logic. For instance, we focus on the subset of Clight that is actually used in the implementation of CertiKOS.

Some particularities of the logic can be better understood with respect to Clight and the continuation-based small-step semantics for Clight programs that is used in CompCert.

---

[6] Except the first one, which determinizes the semantics of CompCert C.

$$\frac{\Delta(x) = \ell \qquad [\![E]\!]^{\Delta}_{\sigma} = v \qquad H(\ell) \neq \blacksquare \qquad \sigma = (\theta, H)}{(\Sigma, \Delta) \vdash (x = E, K, \sigma) \to_{\epsilon} (\mathsf{skip}, K, (\theta, H[\ell \mapsto v]))}\ (\text{E:AssignG})$$

$$\frac{x \in \mathrm{dom}(\theta) \qquad [\![E]\!]^{\Delta}_{\sigma} = v \qquad \sigma = (\theta, H)}{(\Sigma, \Delta) \vdash (x = E, K, \sigma) \to_{\epsilon} (\mathsf{skip}, K, (\theta[x \mapsto v], H))}\ (\text{E:AssignL})$$

$$\frac{\sigma = (\_, H) \qquad [\![E]\!]^{\Delta}_{\sigma} = v \qquad \sigma' = (\theta[x \mapsto v], H)}{(\Sigma, \Delta) \vdash (\mathsf{return}\, E, \mathsf{Kcall}\, x\, f\, \theta\, K, \sigma) \to_{\mathsf{ret}(f)} (\mathsf{skip}, K, \sigma')}\ (\text{E:RetCall})$$

$$(\Sigma, \Delta) \vdash (\mathsf{return}\, E, \mathsf{Kseq}\, S\, K, \sigma) \to_{\epsilon} (\mathsf{return}\, E, K, \sigma)\ (\text{E:RetSeq})$$

$$(\Sigma, \Delta) \vdash (\mathsf{return}\, E, \mathsf{Kloop}\, S\, K, \sigma) \to_{\epsilon} (\mathsf{return}\, E, K, \sigma)\ (\text{E:RetLoop})$$

$$(\Sigma, \Delta) \vdash (\mathsf{break}, \mathsf{Kseq}\, S\, K, \sigma) \to_{\epsilon} (\mathsf{break}, K, \sigma)\ (\text{E:BreakSeq}) \qquad (\Sigma, \Delta) \vdash (\mathsf{break}, \mathsf{Kloop}\, S\, K, \sigma) \to_{\epsilon} (\mathsf{skip}, K, \sigma)\ (\text{E:BreakLoop})$$

$$\frac{\Sigma(f) = (x_1, \ldots, x_n, S_f) \qquad \forall i : [\![E_i]\!]^{\Delta}_{\sigma} = v_i \qquad \theta_f = x_1 \mapsto v_1, \ldots, x_n \mapsto v_n \qquad \sigma = (\theta, H)}{(\Sigma, \Delta) \vdash (x = f(E_1, \ldots, E_n), K, \sigma) \to_{\mathsf{call}(f)} (S_f, \mathsf{Kcall}\, x\, f\, \theta\, K, (\theta_f, H))}\ (\text{E:Call})$$

$$(\Sigma, \Delta) \vdash (\mathsf{skip}, \mathsf{Kseq}\, S\, K, \sigma) \to_{\epsilon} (S, K, \sigma)\ (\text{E:SkipSeq}) \qquad (\Sigma, \Delta) \vdash (S_1; S_2, K, \sigma) \to_{\epsilon} (S_1, \mathsf{Kseq}\, S_2\, K, \sigma)\ (\text{E:Seq})$$

$$\frac{[\![E]\!]^{\Delta}_{\sigma} = \mathsf{int}\, 1}{(\Sigma, \Delta) \vdash (\mathsf{if}\, (E)\, \mathsf{then}\, S_1\, \mathsf{else}\, S_2, K, \sigma) \to_{\epsilon} (S_1, K, \sigma)}\ (\text{E:Cond1})$$

$$\frac{[\![E]\!]^{\Delta}_{\sigma} = \mathsf{int}\, 0}{(\Sigma, \Delta) \vdash (\mathsf{if}\, (E)\, \mathsf{then}\, S_1\, \mathsf{else}\, S_2, K, \sigma) \to_{\epsilon} (S_2, K, \sigma)}\ (\text{E:Cond0}) \qquad (\Sigma, \Delta) \vdash (\mathsf{skip}, \mathsf{Kloop}\, S\, K, \sigma) \to_{\epsilon} (\mathsf{loop}\, S, K, \sigma)\ (\text{E:SkipLoop})$$

$$(\Sigma, \Delta) \vdash (\mathsf{loop}\, S, K, \sigma) \to_{\epsilon} (S, \mathsf{Kloop}\, S\, K, \sigma)\ (\text{E:Loop})$$

Figure 4. Rules of the small-step semantics.

4.1 CompCert Clight

CompCert Clight is the most abstract intermediate language used by CompCert. Mainly, it is a subset of C in which loops can only be exited with a break statement and expressions are free of side effects. Using Clight instead of C simplifies the definition of our quantitative program logic and is also in line with the design of CompCert and the verification of CertiKOS.

Expressions As in Clight, we consider a subset of C expressions without side-effects.

Expressions	$E, E_1, E_2 ::= n$	integer constant
	$\mid \&x$	address of variable
	$\mid *E$	pointer dereference
	$\mid uop\ E$	unary operation
	$\mid E_1\ bop\ E_2$	binary operation

We skip the definitions of unary and binary operations $unop$ and $binop$. They are not important for the results in our paper and can be found in the CompCert source code.

Statements We use a subset of Clight to focus on the main ideas of our program logic.

For simplicity, all loops are infinite unless they are terminated using a break command. As mentioned, only variables x can appear on the left-hand side of assignments in our language. We do not consider function pointers, goto statements, continue statements, and switch statements (see 4.4).

Statements

$S, S_1, S_2 ::= \mathsf{skip}$	do nothing
$\mid x = E$	assignment
$\mid x = f(E^*)$	assignment & function call
$\mid S_1; S_2$	sequential composition
$\mid \mathsf{if}\ (E)\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2$	conditional
$\mid \mathsf{loop}\ S$	infinite loop
$\mid \mathsf{break}$	break loop
$\mid \mathsf{return}\ E$	return from function

Programs Like in C, a program consists of a list of global variable declarations, a list of (internal) function declarations, and the identifier of the main statement, which is the entry point of the program.

Variable declaration $vdec ::= T\, x$

Function declarations $fdec ::= T\, x(vdec^*)\{vdec^*; S\}$

Programs $prog ::= \{fdec \mid vdec\}^*; \mathsf{main} = x$

4.2 Operational Semantics

CompCert Clight's semantics is based on small-step transitions and continuations. Expressions—which do not have side effects—are evaluated in a big-step fashion. We use a simplified version of Clight's semantics that is sufficient for our subset. It is easy to relate evaluations in our simplified version to evaluations in the original semantics and we have implemented a verified compiler from our simple Clight to Clight with CompCert's original semantics.

Values A value is either an integer or a memory address.

$$Val ::= \mathsf{int}\ n \mid \mathsf{adr}\ \ell$$

Here, we have $\ell \in$ Loc and $n \in \mathbb{Z}$.

Memory Model In the Coq development we use CompCert's memory model. However, the main ideas of the logic can be described with a simple abstract memory model in which locations are mapped to values and labels $\blacksquare$, $\square$.

$$H : Mem = \mathrm{Loc} \to Val \cup \{\blacksquare, \square\}$$

The label $\blacksquare$ is used to indicate that a location has been freed and can no longer be used. This is not only beneficial to prove compiler correctness but also in line with the C standard.

The label $\square$ is used to initialize new memory cells. A cell that contains $\square$ cannot be read until a proper value $v \in Val$ has been stored in it.

Evaluating Expressions In contrast to C and CompCert Clight, we do not distinguish between *l*-value and *r*-value positions because expressions are always in *r*-value positions in this article.

Expressions are evaluated with respect to a memory $H : Mem$ and two environments

$$\theta : VID \to Val \qquad \text{and} \qquad \Delta : VID \to \mathrm{Loc}\ .$$

The local environment θ maps local variables to values and the global environment Δ maps global variables to locations. We assume that always $\mathrm{dom}(\Delta) \cap \mathrm{dom}(\theta) = \varnothing$.

The semantics $[\![E]\!]^{\Delta}_{(\theta,H)} = v$ of an expression E under a global environment Δ, a local environment θ, and a memory H is defined by induction on the structure of E.

$$\begin{array}{lll} [\![n]\!]^{\Delta}_{(\theta,H)} & = \mathsf{int}\, n & \text{if } n \in \mathbb{Z} \\ [\![\&x]\!]^{\Delta}_{(\theta,H)} & = \theta(x) & \text{if } x \in \mathrm{dom}(\theta) \\ [\![*E]\!]^{\Delta}_{(\theta,H)} & = H(\ell) & \text{if } [\![E]\!]^{\Delta}_{(\theta,H)} = \mathsf{adr}\, \ell \\ & \dots & \end{array}$$

Continuations The small-step transition relation for statements is based on continuations. Continuation handle the local control flow within a function body such as sequences, loops as well as the logical call stack.

$$K ::= \mathsf{Kstop} \mid \mathsf{Kseq}\, S\, K \mid \mathsf{Kloop}\, S\, K \mid \mathsf{Kcall}\, x\, f\, \theta\, K$$

A continuation K is either the empty continuation Kstop, a sequence $\mathsf{Kseq}\, S\, K$, a loop $\mathsf{Kloop}\, S\, K$, or a stack frame $\mathsf{Kcall}\, x\, f\, \theta\, K$.

Evaluating Statements Statements are evaluated under a program state $(\theta, H) \in State = (VID \to Val) \times Mem$ and a *global environment*

$$(\Sigma, \Delta) : FID \to ([VID] \times \mathcal{S}) \times (VID \to \mathrm{Loc})$$

that maps (internal) functions to their definitions—a list of argument names and the function body—and global variables to values.

The small-step evaluation rules are given in Figure 4. They define a transition

$$(\Sigma, \Delta) \vdash (S, K, \sigma) \to_{\{\mu|\nu|\epsilon\}} (S', K', \sigma')$$

where μ is a memory event, ν is an I/O event, ϵ denotes no event, S, S' are statements, K, K' are continuations, and $\sigma, \sigma' \in State$ are program states.

From the small-step transition relation we derive the following many-step relation in which t is a finite trace. We write

$$(\Sigma, \Delta) \vdash (S_1, K_1, \sigma_1) \to^n_t (S_{n+1}, K_{n+1}, \sigma_{n+1})$$

if $t = a_1, \dots, a_n$ and there exists (S_i, K_i, σ_i) such that for all i

$$(\Sigma, \Delta) \vdash (S_i, K_i, \sigma_i) \to_{a_i} (S_{i+1}, K_{i+1}, \sigma_{i+1})\ .$$

For a statement S and a continuation K, we define the weight $(\Sigma, \Delta, M) \vdash W_\sigma(S, K)$ under the global environment (Σ, Δ), the program state σ, and the metric M as

$$\begin{aligned} (\Sigma, \Delta, M) \vdash W_\sigma(S, K) = \sup\{V_M(t) \mid \exists S', K', \sigma', t, n\,.\,(\Sigma, \Delta) \\ \vdash (S, K, \sigma) \to^n_t (S', K', \sigma')\}\ . \end{aligned}$$

4.3 Quantitative Hoare Logic

In the following we describe a simplified version of the quantitative Hoare logic that we use in Coq to prove bounds on the weights of the traces of Clight programs. For a given statement S and a continuation K, our goal is to derive a bound $(\Sigma, \Delta) \vdash P(\sigma, M) \in \mathbb{N}$ such that $(\Sigma, \Delta) \vdash P(\sigma, M) \geqslant W_{(\sigma,M)}(S, K)$ for all program states σ and resource metrics M. In the remainder of this section we assume a fixed global environment (Σ, Δ).

We generalize classic Hoare logic to express not only classical boolean-valued assertions but also assertions that talk about the future stack-space usage. Instead of the usual assertions $P : State \to bool$ of Hoare logic we use assertions

$$P : State \to \mathbb{N} \cup \{\infty\}\ .$$

This can be understood as a refinement of boolean assertions where $false$ is interpreted by ∞ and $true$ is refined by $\mathbb{N}$. We write $Assn$ for $State \to \mathbb{N} \cup \{\infty\}$, and $\bot = (\_ \mapsto \infty)$. In the actual implementation, assertions have the type $State \to \mathbb{N} \to \mathrm{Prop}$. For a given $\sigma \in State$, such an assertion can be seen as a set $B \subseteq \mathbb{N}$ of valid bounds. We do this only to use Coq's support for propositional reasoning. The presentation here is easier to read.

The continuation-based semantics of a Clight requires that we distinguish pre- and postconditions in the logic to account for different possible ways to exit a block of code. This is approach is standard in Hoare logics and followed for instance in Appel's separation logic for Clight [3]. Our postconditions

$$Q = (Q^s, Q^b, Q^r) : Assn \times Assn \times (Val \to Assn)$$

provide one assertion Q^s for the case in which the block is exited by fall through, one assertion Q^b if the block is exited by a break, and a function Q^r from values to assertions in case the block is exited by a return. The function argument in the last case represents the return value and the intended meaning is that the resulting assertion is guaranteed to hold for every return value.

Since we have to deal with recursive functions, we also need a function context

$$\Gamma : FID \to ((Val \times Mem) \to \mathbb{N} \cup \{\infty\}) \times ((Val \times Mem) \to \mathbb{N} \cup \{\infty\})$$

that maps function names to their specifications, that is, pre- and postconditions. The precondition depends on the value that is passed to the function by the caller and the memory. The postcondition depends on the return value and the memory. For simplicity, we assume that a function has only one argument in this article. In the Coq implementation, an arbitrary number of function arguments is allowed.

In summary, a quantitative Hoare triple has the form

$$\Gamma \vdash \{P\}\, S\, \{Q\}$$

where Γ is a function context, $P : Assn$ is a precondition, $Q : Assn \times Assn \times (Val \to Assn)$ is a postcondition, and S is a statement.

Intuitively, an assertion can be seen as a *potential function* that maps a program state to a non-negative potential. The potential of the precondition P must be sufficient to cover the cost of the execution of the statement S and the potential Q after the execution of S (as in amortized resource analysis [19]).

$$\Gamma \vdash \{Q^s\}\,\mathsf{skip}\,\{Q\}\ (\text{Q:}\textsc{Skip}) \qquad \Gamma \vdash \{Q^b\}\,\mathsf{break}\,\{Q\}\ (\text{Q:}\textsc{Break}) \qquad \Gamma \vdash \{\lambda\sigma\,.\,Q^r\ \llbracket E\rrbracket^{\Delta}_{\sigma})\}\,\mathsf{return}\,E\,\{Q\}\ (\text{Q:}\textsc{Return})$$

$$\frac{P = \lambda(\theta,H)\,.\,Q^s(\theta[x \mapsto \llbracket E\rrbracket^{\Delta}_{(\theta,H)}],H)}{\Gamma \vdash \{P\}\,x = E\,\{Q\}}\ (\text{Q:}\textsc{AssignL}) \qquad \frac{P = \lambda(\theta,H)\,.\,Q^s(\theta,H[\Delta(x) \mapsto \llbracket E\rrbracket^{\Delta}_{(\theta,H)}])}{\Gamma \vdash \{P\}\,x = E\,\{Q\}}\ (\text{Q:}\textsc{AssignG})$$

$$\frac{\Gamma \vdash \{P\}\,S_1\,\{(R,Q^b,Q^r)\} \qquad \Gamma \vdash \{R\}\,S_2\,\{Q\}}{\Gamma \vdash \{P\}\,S_1;S_2\,\{Q\}}\ (\text{Q:}\textsc{Seq})$$

$$\frac{\Gamma \vdash \{\lambda\sigma.(\llbracket E\rrbracket^{\Delta}_{\sigma} \neq 0) + P(\sigma)\}\,S_1\,\{Q\} \qquad \Gamma \vdash \{\lambda\sigma.(\llbracket E\rrbracket^{\Delta}_{\sigma} = 0) + P(\sigma)\}\,S_2\,\{Q\}}{\Gamma \vdash \{P\}\,\mathsf{if}\,(E)\,\mathsf{then}\,S_1\,\mathsf{else}\,S_2\,\{Q\}}\ (\text{Q:}\textsc{Cond})$$

$$\frac{\Gamma \vdash \{I^s\}\,S\,\{I\}}{\Gamma \vdash \{I^s\}\,\mathsf{loop}\,S\,\{(I^b,\bot,I^r)\}}\ (\text{Q:}\textsc{Loop})$$

$$\frac{\Gamma(f) = (P_f,Q_f) \qquad P = \lambda(\theta,H)\,.\,P_f(\llbracket E\rrbracket^{\Delta}_{(\theta,H)},H) \qquad Q = \lambda(\theta,H)\,.\,Q_f(\llbracket x\rrbracket^{\Delta}_{(\theta,H)},H)}{\Gamma \vdash \{P + M(f)\}\,x = f(E)\,\{(Q + M(f),\bot,\bot)\}}\ (\text{Q:}\textsc{Call})$$

$$\frac{\begin{array}{c}\Gamma' = \Gamma, f : (P_f,Q_f) \qquad \Sigma(f) = (x,S_f)\\ \Gamma' \vdash \{P\}\,S\,\{Q\} \qquad \Gamma' \vdash \{P'\}\,S_f\,\{\bot,\bot,Q'\} \qquad P' = \lambda(\theta,H)\,.\,P_f(\theta(x),H) \qquad Q' = \lambda(\theta,H)\,.\,\lambda r\,.\,Q_f(r,H)\end{array}}{\Gamma \vdash \{P\}\,S\,\{Q\}}\ (\text{Q:}\textsc{Abstract})$$

$$\frac{c \geqslant 0 \qquad \{P\}\,S\,\{Q\}}{\{P + c\}\,S\,\{Q + c\}}\ (\text{Q:}\textsc{Frame}) \qquad \frac{P \geqslant P' \qquad \{P'\}\,S\,\{Q'\} \qquad Q' \geqslant Q}{\{P\}\,S\,\{Q\}}\ (\text{Q:}\textsc{Conseq})$$

Figure 5. Rules of the quantitative program logic.

Rules In the rules of the program logic, we use the usual extensions of the operations $+$ and $\geqslant$ on $\mathbb{N} \cup \{\infty\}$. We have $\infty + n = n + \infty = \infty$ and $\infty \geqslant n$ for all $n \in \mathbb{N} \cup \{\infty\}$. In an assertion, we interpret Boolean b as an element of $\overline{b} \in \mathbb{N} \cup \{\infty\}$. We define $\overline{false} = \infty$ and $\overline{true} = 0$. We also lift the operations $+$ and $\geqslant$ pointwise to assertions $P, Q : Assn$. A constant $c \in \mathbb{N} \cup \{\infty\}$ is sometimes used as the constant assertion $P(\sigma) = c$. We define $\bot : Assn$ and $\top : Assn$ to be the assertions with $\bot(\sigma) = \infty$ and $\bot(\sigma) = 0$ for all σ, respectively. As mentioned we fix an event metric M and a global environment (Σ, Δ).

In the rule Q:SKIP, we do not have to account for any stack consumption. As a result, the precondition can be any (potential) function. After the execution, the skip part of the postcondition must be valid on the same (unchanged) program state. So we have to make sure that we do not end up with more potential and simply use the precondition as the skip part of the postcondition. The break and return parts of the postcondition are not reachable and can therefore be arbitrary.

The rules Q:BREAK and Q:RETURN are similar to the rule Q:SKIP. The only difference is that we replace requirement of the skip part of the postcondition with analogue requirements on the break and return part, respectively. The aforementioned rules not trigger any stack-space consumption nor release of stack-space. This might seem strange in the return case but will become clear in the description of the Q:CALL rule later on.

The rules Q:ASSIGNL and Q:ASSIGNC for local and global assignment follow the traditional backward style of Hoare logic. It is ensured that the precondition takes the same value on the initial state as the skip part of the postcondition on the state modified by assignment.

Despite its seemingly simplicity, the Q:SEQ rule must not be overlooked to understand how the quantitative Hoare logic works. We have to define it in such a way that it accounts for early exits in statements. For instance, if S_1 contains a break statement then S_2 will never be executed so we must ensure in the break part of S_1's postcondition that the break part of $S_1; S_2$ holds. For the same reason, the return part of S_1's postcondition is special.

The Q:LOOP rule uses the same principles as the SEQ rule to tweak the final postcondition. In the case of Q:LOOP, we simply ensure that the break part of the inner statement becomes the skip part of the overall statement. We use $\bot$ as the break part of the loop S statement since its operational semantics prevent it from terminating differently than by a skip or a return.

The Q:CALL rule accounts for the actual stack-space usage of programs. It enforces that enough stack space is available to call the function f by adding $M(f)$ to the precondition and the postcondition. The pre- and postconditions are taken from the context Γ. This context is extended using the Q:ABSTRACT rule described below. The assertions in the context are parametric with respect to both the function argument value and the return value. This allows to specify a bound for a function whose recursion depth depends on an input parameter. The argument parameter is instantiated by the call rule using the result of the evaluation of the argument expression in the current state. Note the symmetry of the rule Q:CALL: $M(f)$ is added on both sides to account for the stack space that becomes available after the call. This justifies that the Q:RETURN rule does not account for stack-space release.

Finally, we describe the rules which are not syntax directed. The rule Q:ABSTRACT allows to make a proof on any statement with an extra hypothesis in the derivation context provided that we have a proof that this hypothesis is true. We can see as the quantitative interpretation of the usual logical Modus Ponens.

There are two weakening rules available in the quantitative Hoare logic. The framing rule Q:FRAME is designed to weaken a statement by stating that if S needs P bytes to run and leaves Q bytes free at its end, then it can very well run with $P + c$ bytes and return $Q + c$ bytes. It comes very handy when we want to prove tight bounds using the $\max$ function as demonstrated in Figure 6. The

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma(f) = (\lambda(v,H)\,.\,0, \lambda(v,H)\,.\,0)}{\Gamma \vdash \{(m_f\}\,\mathsf{f}()\,\{(m_f,\bot,\bot)\}}\;(\text{Q:CALL})}{\Gamma \vdash \{(m_f+X_f\}\,\mathsf{f}()\,\{(m_f+X_f,\bot,\bot)\}}\;(\text{Q:FRAME})}{\Gamma \vdash \{\max(m_f,m_g)\}\,\mathsf{f}()\,\{Q\}}\;(\text{EQ}) \qquad \dfrac{\dfrac{\dfrac{\Gamma(g) = (\lambda(v,H)\,.\,0, \lambda(v,H)\,.\,0)}{\Gamma \vdash \{(m_g\}\,\mathsf{g}()\,\{(m_g,\bot,\bot)\}}\;(\text{Q:CALL})}{\Gamma \vdash \{(m_g+X_g\}\,\mathsf{g}()\,\{(m_g+X_g,\bot,\bot)\}}\;(\text{Q:FRAME})}{\Gamma \vdash \{\max(m_f,m_g)\}\,\mathsf{g}()\,\{Q\}}\;(\text{EQ})}{\Gamma \vdash \{\max(m_f,m_g)\}\,\mathsf{f}();\mathsf{g}()\,\{Q\}}\;(\text{Q:SEQ})$$

where $\quad M(\mathsf{call}(f)) = m_f \quad M(\mathsf{call}(g)) = m_g \quad Q = (\max(m_f,m_g),\bot,\bot) \quad X_\theta = \max(m_f,m_g) - m_\theta)$ for $\theta \in \{f,g\}$

Figure 6. An example derivation of a stack-space bound in the quantitative logic.

consequence Q:CONSEQ rule is directly imported from classical Hoare logics except that instead of using the logical implication $\Rightarrow$ we use the quantitative $\geqslant$. This rule indeed weakens the statement since it requires more resource to run the statement and yields less than what has been proved to be available after its termination.

Auxiliary State As mentioned, our account of the logic in this article is slightly simplified compared to the Coq development to improve readability.

The main difference between the implemented logic and the logic described here is that the latter does not have an *auxiliary state*. Auxiliary state is a classic extension of Hoare logic (see for example [32]). It is used to share information between the pre- and postcondition of a triple. In a logic without auxiliary state (or similar techniques) it is not possible to relate program states before and after a statement. For example, you cannot specify that the function `int twice () {i = i+i;}` doubles the value of the variable i. With an auxiliary variable Z it is possible specify this fact in Hoare logic using the triple $\{i = Z\}\,\mathsf{twice}()\,\{i = 2 \cdot Z\}$.

In the Coq implementation, assertions have the type $\mathit{State} \times \mathit{Aux} \to \mathbb{N} \cup \{\infty\}$. The auxiliary state is an arbitrary type in Coq and can be instantiated by the user. Most of the rules of the logic remain unchanged in the presence of auxiliary state. The only exception is the consequence rule Q:CONSEQ that reads as follows.

$$\dfrac{\{P'\}\,S\,\{R\} \qquad \begin{array}{r} \forall\sigma\, a. P(\sigma,a)\in\mathbb{N} \implies \exists a'. P(\sigma,a) \geqslant P'(\sigma,a') \\ \wedge (\forall \sigma'\, i. R^i(\sigma',a') \geqslant Q^i(\sigma',a)) \end{array}}{\{P\}\,S\,\{Q\}}$$

This is the quantitative version of a consequence rule that has been introduced in the context of Hoare logic and is attributed to Martin Hofmann [32]. Its typical use case is when we apply the rule Q:CALL to a recursive call. In this case, the Hoare triple for the function call is proved by an assumption from the derivation context with a slightly different auxiliary state. In the example derivation in Figure 7 this different state is $Z - 1$. Adapting the derivation hypothesis to prove the recursive call is enabled in our logic by the extended consequence rule introduced above.

Stack Framing A minor difference in the function application rule is that we only present the rule for function calls with a single argument and without *framing of stack assertions*. The latter is necessary in the code of a caller to carry over information on the local environment from the precondition of the function call to the postcondition of the function call. This is a general problem in Hoare logic and stack framing is a well known solution. For instance, we would like to prove something like $\{y\}\,f(x)\,\{y\}$ where y is a local variable. This is not possible with the rule Q:CALL but it is with the more general rule in the Coq implementation [9].

Soundness The soundness of our quantitative logic can be simply expressed by the following theorem.

Theorem 2. *For a fixed global environment* (Σ, Δ)*, a derivation in our quantitative logic for a statement* S *implies a bound on the weight of* S*, that is,*

$$\cdot \vdash \{P\}\,S\,\{Q\} \implies \forall\sigma, M\,.\, P(\sigma, M) \geqslant W_{(\sigma,M)}(S, \mathsf{Kstop}).$$

Naturally, we have to prove a stronger statement that takes postconditions and continuations into account to justify the soundness of the rules of the logic. This is not unlike as in program logics for low-level code [22] and Hoare-style logics for CompCert Clight [3]. Furthermore, we have to assume that we have a non-empty function context Γ; and finally, we have to step-index the correctness statement in order to prove its soundness by induction.

For a precondition P, a statement S, and a continuation K, we define $\mathit{safe}(P, S, K, n)$ through

$$\forall\sigma, M\,.\, P(\sigma, M) \geqslant \max\{V_M(t) \mid \exists m \leqslant n\,.\,(S, K, \sigma) \to_t^m \_\}$$

For a postcondition Q, and a continuation K, $\mathit{safeK}(Q, K, n)$ is defined as

$$\mathit{safe}(Q^s, \mathsf{skip}, K, n) \wedge \mathit{safe}(Q^b, \mathsf{break}, K, n) \wedge \forall\sigma, M, E\,.$$
$$Q^r([\![E]\!]_\sigma)(\sigma, M) \geqslant \max\{V_M(t) \mid \exists m \leqslant n\,.(\mathsf{return}\,E, K, \sigma) \to_t^m \_\}$$

Now we can define the validity of our quantitative Hoare triples. Like in Vafeiadis' soundness proof of concurrent separation logic [36] we bake the quantitative frame rule into the definition of validity. We say that a triple $\{P\}\,S\,\{Q\}$ is valid for n steps and write $\mathit{valid}(P, S, Q, n)$ if the following holds.

$$\forall m \leqslant n, K, c\,.\, \mathit{safeK}(Q{+}c, K, m) \implies \mathit{safe}(P{+}c, S, K, m)$$

Here, $Q{+}c$ is short for $(Q^s{+}c, Q^b{+}c, \lambda v\,.\,Q^r(v){+}c)$.

An interesting detail in the definition is the natural number m. We simply use it to ensure that $\mathit{valid}(P, S, Q, n+1) \implies \mathit{valid}(P, S, Q, n)$. This would not be the case if we replaced all occurrences of m by n in the definition of validity.

We say a context Γ is valid for the global environment, and write $(\Sigma, \Delta) \models \Gamma$ if the following holds.

$$\Gamma(f) = (P_f, Q_f) \wedge \Sigma(f) = (x, S_f) \implies \forall n\,.\,\mathit{valid}(P, S_f, Q, n)$$

Here we define $P(\theta, H) = P_f(\theta(x), H)$ and

$$Q = (\top, \top, \lambda v\,.\,\lambda(\theta, H)\,.\,Q_f(v, H))\ .$$

Finally, we say that a triple $\{P\}\,S\,\{Q\}$ is valid under the function signature Γ and write $\Gamma \models \{P\}\,S\,\{Q\}$ if for every global environment (Σ, Δ) we have $(\Sigma, \Delta) \models \Gamma \implies \forall n\,.\,\mathit{valid}(P, S, Q, n)$.

Of course we prove in Coq that the intuitive validity, as formulated in (2), is a consequence of our stronger formulation of validity.

Examples Figure 6 contains an example derivation for the statement $\mathsf{f}();\mathsf{g}()$ in our logic. We assume that we have already verified that the function bodies of f and g do not allocate stack space, that is, $\Gamma(g) = \Gamma(f) = (\lambda(v, H)\,.\,0, \lambda(v, H)\,.\,0)$.

Our goal is to derive the quantitative Hoare triple $\Gamma \vdash \{\max(m_f, m_g)\}\,\mathsf{f}();\mathsf{g}()\,\{(\max(m_f, m_g), \bot, \bot)\}$ which expresses that $\max(m_f, m_g)$, the maximum of the stack frame sizes of f and g, is a bound on the stack usage; and that after the execution $\max(m_f, m_g)$ stack space is available. Since the effect of break and return statements cannot leak outside of a function body, we know that no break or return will occur in the execution of the statement $\mathsf{f}();\mathsf{g}()$. Therefore the corresponding postconditions can be arbitrary and we simply use $\bot$.

```
$\{Z = \log_2(\mathsf{h}_\sigma - \mathsf{l}_\sigma) \Rightarrow M_b \cdot Z)\}$
bsearch(x,l,h) {
  if (h-l <= 1) return l;
  $\{(Z>0 \wedge Z = \log_2(\mathsf{h}_\sigma - \mathsf{l}_\sigma)) \Rightarrow M_b \cdot Z\}$
  m = (h+l)/2;
  $\{(Z>0 \wedge Z = \log_2(\mathsf{h}_\sigma - \mathsf{l}_\sigma) \wedge \mathsf{m}_\sigma = \frac{\mathsf{h}_\sigma + \mathsf{l}_\sigma}{2}) \Rightarrow M_b \cdot Z\}$
  if (a[m]>x) h=m else l=m;
  $\{[Z-1 = \log_2(\mathsf{h}_\sigma - \mathsf{l}_\sigma) \Rightarrow M_b \cdot (Z-1)] + M_b\}$
  return bsearch(x,l,h);
  $\{[M_b \cdot (Z-1)] + M_b\}$
}
$\{M_b \cdot Z\}$
```

Figure 7. Derivation for the bsearch function.

To derive our goal, we first have to apply the rule Q:SEQ for sequential composition. In the derivation of the function call f(), we first reorder the precondition to get it in a form in which we can apply the rule Q:FRAME to eliminate the max operator. We than have a triple that is amenable to an application of the rule Q:CALL that uses the specification of the body of f in Γ. The derivation the function call f() is very similar.

More examples can be found in our Coq development [9].

4.4 Limitations

In our program logic described in this section, we do not consider function pointers, goto statements, continue statements, and switch statements, even though our Quantitative CompCert compiler still supports all of these. It would be possible to add these features to our logic by building on the ideas of advanced program logics like XCAP [31].

5. Automatic Stack Analyzer

In larger C programs a manual, interactive verification with a program logic is too tedious and time-consuming to be practical. Therefore we have developed an automatic stack analysis tool that operates at the Clight level to enable the analysis of real system code. We view this automatic tool mainly as a proof of concept that demonstrates the value of the logic for formal verification of static analysis tools. In the future, we will extend our automatic analyzer with advanced techniques like amortized resource analysis [21, 5]. This is however beyond the scope of this article.

The basic idea of our 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 our quantitative program logic. This verifies 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.

Given our verified quantitative logic, the implementation of auto_bound is straightforward. For trivial commands like assignments ore skip, auto_bound simply generates the bound 0 and a derivation like $\{0\}\,\mathsf{skip}\,\{(0,0,0)\}$. For a sequential composition $S_1; S_2$ we inductively apply auto_bound to S_1 and S_2, and derive the bounds $\{B_i\}\,S_i\,\{(B_i^s, B_i^b, B_i^r)\}$ for $i{=}1,2$. We then return the precondition $\max\{B_1, B_2\}$ and the postcondition $(\max\{B_1^s, B_2^s\}, \max\{B_1^b, B_2^b\}, \max\{B_1^s, B_2^s\})$ for $S_1; S_2$. The derivation of this bound is similar to the example derivation that is sketched in Figure 6. The computation of the bound for the conditional works similar. For loops we can use the bound derived for the loop body to obtain a bound for the loop. 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, we apply auto_bound to every function definition in the well-founded topological order that is given by the call graph. We then use the resulting bounds to successively generate the context of known function bounds for the following calls of auto_bound. We envision, that the quantitative logic can be a useful backend to verify more sophisticated static analyses. For our simple, automatic stack analyzer the logic was already very convenient and enabled us to verify the analyzer almost without additional effort. If one considers more involved programs with recursion and function pointers then more verification effort is inevitable but based on our experiences, we believe that the quantitative logic will be quite helpful.

We have combined our automatic stack analyzer with our 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 computed by the automatic stack analysis for main() (compare Point 3 of Theorem 1). During compilation, the Stack-Aware CompCert Compiler prints the computed stack bound for every function and the overall stack requirement for the program.

The Stack-Aware CompCert Compiler is part of our Coq development [9].

6. Experimental Evaluation

To validate the practicality of our framework for stack-bound verification, we have performed an experimental evaluation with more than 3000 lines of C code from different sources. The C programs used to evaluate the quantitative Hoare logic and the automatic stack analyzer include hand written code, programs from the CompCert test suite[7], programs from the MiBench [17] embedded software benchmarks, and modules from the simplified development version of the CertiKOS operating system kernel which is currently being verified.

Tables 1 and 2 show a representative compilation of the experiments. Table 2 consists of bounds that where automatically derived with the stack analyzer. Table 1 contains 8 bounds that where interactively derived using the quantitative logic with occasional support of the automation. The size of the analyzed example files varies from 8 lines of code (fib.c) to 819 lines of code (proc.c). In general, the automatic stack-bound analysis runs very efficiently and needs less than a second for every example file on a Linux workstation with 32G of RAM and processor with 16 cores at 3.10Ghz.

In Table 2, the first column shows the file name of the examples together with the number of lines, the second column contains the name of selected functions from that file, and the third column contains the verified bound. The interactively-derived bounds in Table 1 are presented as symbolic expressions parametric in the functions' arguments. These symbolic expressions are slight simplifications of the real pre- and postconditions of the functions that we proved in Coq. The actual Hoare triples proved in Coq carry a logical meaning which does, for instance, require that the qsort function be called

---

[7] The files mandelbrot.c and nbody.c are originally from The Great Computer Language Shootout.

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 1. Manually verified stack bounds for C functions.

on a valid sub array. The file sizes of the manual verified examples range from 8 to 52 lines of code.

Our main application of the automatic stack-analyzer is the CertiKOS operating system kernel [15]. Currently, the stack in CertiKOS is preallocated and proving the absence of stack-overflow is essential in the verification of the reliability of the system. Since CertiKOS does not make use of recursion, we can use the automatic analysis to derive precise stack bounds. Using our Quantitative CompCert compiler, we were, for instance, able to compile and compute bounds for the virtual memory management module (certikos/vmm.c) and the process management module (certikos/proc.c). In total more than 1500 lines of system C code were processed without any human interaction. Because of the large number of functions in CertiKOS, only a sample of the analyzed functions is displayed in Table 2.

Testing the quantitative Hoare logic and the compiler on CompCert test suite was a natural choice since our compiler builds on CompCert's architecture. This also allowed us to make sure that we did not introduce any regression with respect to the original CompCert compiler. To stress the expressivity of the logic we focused on test programs with recursive functions. The functions fib and qsort in Table 1 are for instance from the CompCert test suite. 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.

We also made sure that the experiments stressed our ability to handle safety critical software. Embedded software typically runs in a highly memory constrained environment and, in the most critical cases, running out of stack space could turn into threat for human lives. The MiBench [17] benchmark we used for this purpose is free, publicly available, and representative for embedded software. The use of recursion in MiBench programs is relatively rare, which makes them a great target for our automatic stack analyzer. The analyzed examples we present in Table 2 include for instance Dijkstra's single-source shortest-path algorithm (dijkstra.c), and the cryptographic algorithms Blowfish (blowfish.c) and MD5 (md5).

Finally, Table 1 contains some recursive functions that demonstrate the expressivity of our quantitative logic. The function bsearch is, for example, a recursive binary search with logarithmic recursion depth. The function fib computes the Fibonacci sequence using an exponential algorithm and the function qsort implements a recursive version of the quicksort algorithm. In both cases the asymptotically tight linear bounds could be proved. The verification of the function fact_sq shows the modularity of the logic: We 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^2). 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. The modularity of the logic enables us to reuse the logarithmic bound that we already derived for bsearch in the proof. The verification of some functions is still underway. The bounds for the functions recid, bsearch, fib, and qsort are already completely verified.

File Name / Line Count	Function Name	Verified Stack Bound
mibench/net/dijkstra.c	enqueue	40 bytes
(174 LOC)	dequeue	40 bytes
	dijkstra	88 bytes
mibench/auto/bitcount.c	bitcount	16 bytes
(110 LOC)	bitstring	32 bytes
mibench/sec/blowfish.c	BF_encrypt	40 bytes
(233 LOC)	BF_options	8 bytes
	BF_ecb_encrypt	80 bytes
mibench/sec/pgp/md5.c	MD5Init	16 bytes
(335 LOC)	MD5Update	168 bytes
	MD5Final	168 bytes
	MD5Transform	128 bytes
mibench/tele/fft.c	IsPowerOfTwo	16 bytes
(195 LOC)	NumberOfBitsNeeded	24 bytes
	ReverseBits	24 bytes
	fft_float	160 bytes
certikos/vmm.c	palloc	48 bytes
(608 LOC)	pfree	40 bytes
	mem_init	72 bytes
	pmap_init	176 bytes
	pt_free	80 bytes
	pt_init	152 bytes
	pt_init_kern	136 bytes
	pt_insert	80 bytes
	pt_read	56 bytes
	pt_resv	120 bytes
certikos/proc.c	enqueue	48 bytes
(819 LOC)	dequeue	48 bytes
	kctxt_new	72 bytes
	sched_init	232 bytes
	tdqueue_init	208 bytes
	thread_init	192 bytes
	thread_spawn	96 bytes
compcert/mandelbrot.c	main	56 bytes
(92 LOC)		
compcert/nbody.c	advance	80 bytes
(174 LOC)	energy	56 bytes
	offset_momentum	24 bytes
	setup_bodies	16 bytes
	main	112 bytes

Table 2. Automatically verified stack bounds for C functions.

Our experiments have shown that the automatic stack analyzer works effectively for our main application, the CertiKOS OS kernel. The reason is that we designed the quantitative logic to include exactly the subset of Clight that is needed for CertiKOS. It turned out that this subset is also sufficient for many examples in the CompCert test suite and the MiBench embedded software benchmarks. If a program is not interactively analyzable in our logic then this due to unsupported language constructs such as switch statements and functions pointers. Many of these language features could easily be supported by relatively small additions to the logic. An exception to this are function pointers which would require more work, following for example XCAP [31].

Accuracy of the Derived Bounds We have evaluated the precision of the automatically and manually derived bounds by comparing our verified upper bounds with the actual stack-space consumption during the execution of the compiled C programs. Our experiments

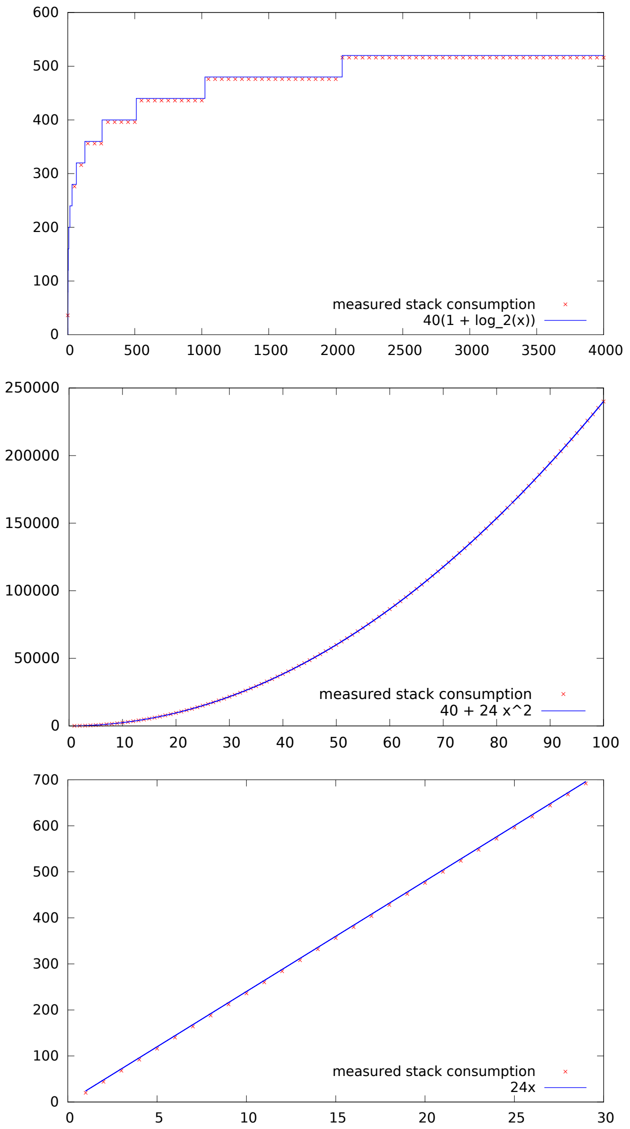

Figure 8. Experimental evaluation of the accuracy of hand-derived bounds. The plots compare the derived bounds (blue lines) for the functions bsearch (at the top), fact_sq (in the middle), and fib (at the bottom) with the measured stack usage of the execution of the respective function for different inputs (red crosses). The experiments indicate that the derived bounds over-approximate the actual stack usage by a small constant factor.

show that the derived bounds are very precise: Both the manually and automatically derived bounds over-approximate the stack usage by exactly four bytes (see the following explanation).

Figure 8 shows the results of three experiments we made with hand-derived stack bounds using the quantitative logic. We plotted the derived bounds for the functions bsearch, fib 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 an integer argument (fib and fact_sq) or the length of an input array (for bsearch). The y-axis shows the stack usage in bytes. The experiments show that the logic is expressive enough to get very tight bounds on the recursive programs. The bsearch example shows that the logarithmic bound derived by the logic is very close the program requirements; the fact_sq example makes the point that our logic is indeed compositional.

We also experimentally proved the efficiency of our 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 compiled and monitored by our tool as a regular Linux program. Further experiments may be possible by using, for instance, an instrumented virtual machine.

As mentioned, all the derived bounds are off by four bytes. The reason for this is that stack frames always reserve four bytes for a potential function call: The return address needs to be pushed by a call instruction in the callee. Obviously, the last function in the function call chain does not call any other function. So these four bytes remain unused. A different point of view is to see these four bytes as the return address of main. Indeed, before main is called, its return address is pushed on the stack. But, as described below, our tool takes the stack pointer at the function prologue as a reference point. So the return address is already on the stack and four bytes are not counted in the experiment.

Various technical problems make the measurement of stack consumption during the execution of compiled C code a complex task on today's systems. These problems involve security features of the host operating system and implicit management of the stack pointer by C compilers. Indeed, instructions allocating and freeing stack space can be emitted by the compiler at any place in the assembly code and can take several forms.

To our knowledge, no tool available today can monitor the stack consumption of running programs with the precision required to evaluate our bounds. For this purpose, we implemented a small program able to monitor resources used by any function of a Linux executable. It uses the `ptrace` system call[8]. This system call allows one Linux process (we will call it the *parent*) to have a very precise control on the execution of another process (called *child*). It is meant to be used by common debugging tools like gdb or `strace`.

Our tool works as follows. We first retreive the location of the entry point of the monitored function using standard ELF files dissection tools. Once we have this address, we can set up a breakpoint by replacing the function prologue with an x86 trap instruction (`ptrace` allows to poke in the child's address space). This trap instruction plays the role of a breakpoint and when the child executes it, control is given back to the monitoring process by the kernel. At this point, we inspect the registers of the child process to get the value of the stack pointer. This will become the stack *reference point*. Now we can restore the function prologue that was overwritten in the first step and proceed with the execution of the child in step-by-step mode. At each executed assembly instruction the control is given back to the parent process which inspects the value of the stack pointer and tracks its watermark. When the stack pointer becomes smaller than the reference point, we know that the child process returned from the tracked function. At this point we stop monitoring the stack pointer and display the stack watermark.

One obvious weakness of this method is that it stops the control of the child process at every assembly instruction, and thus, is very slow. However, for our purposes, this has not been an issue.

7. Related Work

In the following we discuss research that is related to our contributions in verified compilation, program logics, and automatic resource analysis.

---

[8] This monitoring program is available at `http://zoo.cs.yale.edu/~qc35/data/mon.c`

Verified Compilation Soundness proofs of compilers have been extensively studied and we focus on *formally verified* proofs here. Klein and Nipkow [24] developed a verified compiler from an object-oriented, Java-like language to JVM byte code. Chlipala [11] 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 [26, 27]. 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 [6]. Admittedly, there exists a clever annotation mechanism [6] 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 [34]. 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 [18], Jost et al. [23] 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 [32] 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 [32]. Appel's separation logic for CompCert Clight [3] has been a blueprint for the general structure of the quantitative logic. Since we do not deal with memory safety, our logic is much simpler and it would be possible to integrate it with Appel's logic. The continuation passing style that we use in the quantitative logic is not only used by Appel [3] but also in Hoare logics for low-level code [31, 22].

There exist quantitative logics that are integrated into separation logic [5, 20] 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. [4]. 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 on low-level code [8, 33, 10] as well as commercial tools such as the *Bound-T Time and Stack Analyser*[9] and Absint's *StackAnalyzer* [14]. 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 [23, 12, 19, 1]. 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 [39, 16] 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. [2] 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. [7] have verified a loop bound analysis for CompCert's 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.

8. Conclusion

Embedded software has always been a target of verified compilers. As a result, aiding verification of quantitative properties remains a major goal for verified compilation. In one of the earliest articles [26] on CompCert, Leroy stated:

> "[...] it is hopeless to prove a stack memory bound on the source program and expect this resource certification to carry out to compiled code: stack consumption, like execution time, is a program property that is not preserved by compilation."

Ironically, Leroy's groundbreaking work on CompCert has been the main inspiration in our development of a framework that enables exactly such a resource certification of stack-consumption bounds for compiled x86 assembly code *at the C level*.

We have developed Quantitative CompCert, a realistic, verified C compiler which shows how verified compilation enables the verification of quantitative properties of compiled programs at the source level. We have implemented and formally verified a novel quantitative Hoare logic for CompCert Clight which is an ideal backend for static analysis tools. This is demonstrated through the implementation of a verified, automatic stack-analysis tool that computes derivations in the quantitative logic. Finally, we have shown through experiments that our framework can be applied to derive precise stack bounds for typical system code.

Our work opens the door for the verification of powerful static analysis tools for quantitative properties that operate on the C level rather than on the machine code. There are multiple future research directions that we plan to explore on the basis of the present development. For one thing, we want to use our quantitative Hoare logic to verify more powerful analysis tools that can automatically derive stack-space bounds for recursive functions. For another thing, we plan to generalize the developed concepts to apply our technique to other resource such as heap-memory and clock-cycle consumption.

Formal verification and machine-checked proves have been a vital tool during the development of this work, which in some ways challenges the prevalent opinion. Formal verification not only convinced us that our implementation does not contain bugs; a formally proved theorem in Coq is also an argument that proves useful in discussions with skeptical practitioners.

---

[9] `http://www.bound-t.com`

Acknowledgments

This research is based on work supported in part by NSF grants 1319671 and 1065451, and DARPA grants FA8750-10-2-0254 and FA8750-12-2-0293. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

[1] E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, and G. Puebla. Cost Analysis of Concurrent OO Programs. In *Prog. Langs. and Systems - 9th Asian Symposium (APLAS'11)*, pages 238–254, 2011.

[2] E. Albert, R. Bubel, S. Genaim, R. Hähnle, and G. Román-Díez. Verified Resource Guarantees for Heap Manipulating Programs. In *Fundamental Approaches to Software Engineering - 15th Int. Conf. (FASE'12)*, pages 130–145, 2012.

[3] A. W. Appel et al. *Program Logics for Certified Compilers*. Cambridge University Press, 2013.

[4] D. Aspinall, L. Beringer, M. Hofmann, H.-W. Loidl, and A. Momigliano. A Program Logic for Resources. *Theor. Comput. Sci.*, 389(3):411–445, 2007.

[5] R. Atkey. Amortised Resource Analysis with Separation Logic. In *19th Euro. Symp. on Prog. (ESOP'10)*, pages 85–103, 2010.

[6] R. Bedin França, S. Blazy, D. Favre-Felix, X. Leroy, M. Pantel, and J. Souyris. Formally Verified Optimizing Compilation in ACG-based Flight Control Software. In *Embedded Real Time Software and Systems (ERTS 2012)*, 2012.

[7] S. Blazy, A. Maroneze, and D. Pichardie. Formal Verification of Loop Bound Estimation for WCET Analysis. In *Verified Software: Theories, Tools, Experiments - 5th Int. Conf. (VSTTE'13)*, 2013. To appear.

[8] D. Brylow, N. Damgaard, and J. Palsberg. Static Checking of Interrupt-Driven Software. In *23rd Int. Conf. on Software Engineering (ICSE'01)*, pages 47–56, 2001.

[9] Q. Carbonneaux, J. Hoffmann, T. Ramananandro, and Z. Shao. End-to-End Verification of Stack-Space Bounds for C Programs. `http://cs.yale.edu/~hoffmann/files/veristack.zip`, 2013. Coq Development.

[10] W.-N. Chin, H. H. Nguyen, C. Popeea, and S. Qin. Analysing Memory Resource Bounds for Low-Level Programs. In *7th Int Symp. on Memory Management (ISMM'08)*, pages 151–160, 2008.

[11] A. Chlipala. A Certified Type-Preserving Compiler from Lambda Calculus to Assembly Language. In *28th Conf. on Prog. Lang. Design and Impl. (PLDI'07)*, pages 54–65, 2007.

[12] K. Crary and S. Weirich. Resource Bound Certification. In *27th ACM Symp. on Principles of Prog. Langs. (POPL'00)*, pages 184–198, 2000.

[13] Express Logic, Inc. Helping you avoid stack overflow crashes! White Paper, 2014. URL `http://rtos.com/images/uploads/Stack_Analysis_White_paper.1_.pdf`.

[14] C. Ferdinand, R. Heckmann, and B. Franzen. Static Memory and Timing Analysis of Embedded Systems Code. In *3rd Europ. Symp. on Verification and Validation of Software Systems (VVSS'07)*, 2007.

[15] L. Gu, A. Vaynberg, B. Ford, Z. Shao, and D. Costanzo. CertiKOS: A Certified Kernel for Secure Cloud Computing. In *Asia Pacific Workshop on Systems (APSys'11)*, 2011.

[16] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *36th ACM Symp. on Principles of Prog. Langs. (POPL'09)*, pages 127–139, 2009.

[17] 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 *IEEE International Workshop on Workload Characterization (WWC'01)*, pages 3–14, 2001.

[18] K. Hammond and G. Michaelson. Hume: A Domain-Specific Language for Real-Time Embedded Systems. In *Generative Programming and Component Engineering, 2nd Int. Conf. (GPCE'03)*, pages 37–56, 2003.

[19] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate Amortized Resource Analysis. *ACM Trans. Program. Lang. Syst.*, 2012.

[20] J. Hoffmann, M. Marmar, and Z. Shao. Quantitative Reasoning for Proving Lock-Freedom. In *28th ACM/IEEE Symposium on Logic in Computer Science (LICS'13)*, 2013.

[21] M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *30th ACM Symp. on Principles of Prog. Langs. (POPL'03)*, pages 185–197, 2003.

[22] J. B. Jensen, N. Benton, and A. Kennedy. High-Level Separation Logic for Low-Level Code. In *40th ACM Symp. on Principles of Prog. Langs. (POPL'13)*, pages 301–314, 2013.

[23] S. Jost, H.-W. Loidl, K. Hammond, N. Scaife, and M. Hofmann. Carbon Credits for Resource-Bounded Computations using Amortised Analysis. In *16th Symp. on Form. Meth. (FM'09)*, pages 354–369, 2009.

[24] G. Klein and T. Nipkow. A Machine-Checked Model for a Java-Like Language, Virtual Machine, and Compiler. *ACM Trans. Program. Lang. Syst.*, 28(4):619–695, 2006.

[25] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an Operating-System Kernel. *Commun. ACM*, 53(6):107–115, 2010.

[26] X. Leroy. Formal Certification of a Compiler Back-End, or: Programming a Compiler with a Proof Assistant. In *33rd Symposium on Principles of Prog. Langs. (POPL'06)*, pages 42–54, 2006.

[27] X. Leroy. Formal Verification of a Realistic Compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[28] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[29] X. Leroy and S. Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41(1):1–31, 2008.

[30] Y. Moy, E. Ledinot, H. Delseny, V. Wiels, and B. Monate. Testing or Formal Verification: DO-178C Alternatives and Industrial Experience. *IEEE Software*, 30(3):50–57, 2013. ISSN 0740-7459.

[31] Z. Ni and Z. Shao. Certified Assembly Programming with Embedded Code Pointers. In *33th ACM Symp. on Principles of Prog. Langs. (POPL'06)*, pages 320–333, 2006.

[32] T. Nipkow. Hoare Logics in Isabelle/HOL. In *Proof and System-Reliability*, volume 62 of *NATO Science Series*, pages 341–367. Springer, 2002.

[33] J. Regehr, A. Reid, and K. Webb. Eliminating Stack Overflow by Abstract Interpretation. *ACM Trans. Embed. Comput. Syst.*, 4(4):751–778, 2005.

[34] J. Sevcík, V. Vafeiadis, F. Z. Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *J. ACM*, 60(3), 2013.

[35] Z. Shao. Certified software. *Commun. ACM*, 53(12):56–66, 2010.

[36] V. Vafeiadis. Concurrent Separation Logic and Operational Semantics. *Electr. Notes Theor. Comput. Sci.*, 276:335–351, 2011.

[37] R. Wilhelm et al. The Worst-Case Execution-Time Problem — Overview of Methods and Survey of Tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.

[38] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and Understanding Bugs in C Compilers. In *32nd Conf. on Prog. Lang. Design and Impl. (PLDI'11)*, pages 283–294, 2011.

[39] F. Zuleger, M. Sinn, S. Gulwani, and H. Veith. Bound Analysis of Imperative Programs with the Size-change Abstraction. In *18th Int. Static Analysis Symposium (SAS'11)*, 2011.