High-Level Separation Logic for Low-Level Code

Jonas B. Jensen
IT University of Copenhagen
jobr@itu.dk

Nick Benton Andrew Kennedy
Microsoft Research Cambridge
{nick,akenn}@microsoft.com

Abstract

Separation logic is a powerful tool for reasoning about structured, imperative programs that manipulate pointers. However, its application to unstructured, lower-level languages such as assembly language or machine code remains challenging. In this paper we describe a separation logic tailored for this purpose that we have applied to x86 machine-code programs.

The logic is built from an assertion logic on machine states over which we construct a specification logic that encapsulates uses of frames and step indexing. The traditional notion of Hoare triple is not applicable directly to unstructured machine code, where code and data are mixed together and programs do not in general run to completion, so instead we adopt a continuation-passing style of specification with preconditions alone. Nevertheless, the range of primitives provided by the specification logic, which include a higher-order frame connective, a novel read-only frame connective, and a 'later' modality, support the definition of derived forms to support structured-programming-style reasoning for common cases, in which standard rules for Hoare triples are derived as lemmas. Furthermore, our encoding of scoped assembly-language labels lets us give definitions and proof rules for powerful assembly-language 'macros' such as while loops, conditionals and procedures.

We have applied the framework to a model of sequential x86 machine code built entirely within the Coq proof assistant, including tactic support based on computational reflection.

Categories and Subject Descriptors F.3.1 [*Logics and meanings of programs*]: Specifying and Verifying and Reasoning about Programs—Assertions, Invariants, Logics of programs, Mechanical verification, Pre- and post-conditions, Specification techniques; D.3.2 [*Programming Languages*]: Language Classifications—Macro and assembly languages; D.2.4 [*Software Engineering*]: Software / Program Verification—Correctness proofs, formal methods

General Terms Languages, theory, verification

Keywords Separation logic, machine code, proof assistants

1. Introduction

Formal verification is one of the most important techniques for building reliable computer systems. Research in software verification typically, and quite reasonably, concerns reasoning about the high-level programming languages with which most programmers work. But to build genuinely trustworthy systems, one really needs to verify the machine code that actually runs, whether it be hand-crafted or the output of a compiler. This is particularly important for establishing security properties, since failures of abstraction between the high and low-level models often lead to vulnerabilities (and because hand-crafted machine code is often found in security-critical places, such as kernels). A further motivation for verifying low-level code is that real systems are composed of components written in many different languages; machine code is the only truly universal lingua franca by which we can reason about properties of such compositions. Finally, experience shows that hand-written low-level programs are simply much harder to get right than higher-level ones, increasing the credibility gap between formal and informal verifications.

Verifying low-level, unstructured programs [18, 39], and compilers that produce them [24], both have a long history. And since such verifications are, like low-level programs themselves, extremely lengthy and error-prone, some form of mechanical assistance is absolutely crucial. This assistance often takes the form of automated decision procedures for first-order logic and various more specialized theories, combined by an SMT solver [21]. Here, however, our focus is on deductive Hoare-style verification of machine-code programs using an interactive proof assistant, in our case Coq. This requires more manual effort on the part of the user, but allows one to work with much richer mathematical models and specifications, which are particularly important for modularity. Both approaches to mechanization also have considerable history, with much pioneering work applying interactive provers to low-level code having been done with the Boyer-Moore prover in the 1980s [25, 41]. Recently, however, an exciting confluence of advances in foundational theory, program logics (most notably separation logic [35]) and the technology of proof assistants, together with increased interest in formal certification, have led to an explosion of work on mechanized verification of real (or at least, realistic) software, including compilers and operating systems. Although many of these formalizations do involve reasoning about machine code or assembly language programs, program logics for low-level code are generally much less satisfactory, and more ad hoc, than those for high-level languages.

The design of a high-level program logic tends to follow closely the structure and abstractions provided by the language. Commands in a while-language, for example, may be modelled as partial functions from stores to stores, which are only combined in certain very restricted ways. The classical Hoare triple, relating a predicate on inputs to a predicate on outputs, is a natural (indeed, inevitable) and generally satisfactory form of specification for such functions. Furthermore, the structured form of programs leads to particularly elegant, syntax-directed program logic rules for composing verifications. Machine code, by contrast, has almost nothing in the way of inherent structure or abstractions to guide one, supports chal-

POPL'13, January 23–25, 2013, Rome, Italy.

lenging patterns of programming and also involves a host of messy complexities.

The messy complexities include large instruction sets with variable-length encodings, the need to work with bit-level operations and arithmetic mod 2^{32}, alignment, a plethora of flags, registers, addressing modes, and so on. These inevitably cause some pain, but are just the sort of thing proof assistants are good at checking precisely and, with a well-engineered formalization, removing some of the drudgery from.[1] There is, of course, complexity of a quite different order associated (at both high- and low-level) with concurrency – especially relaxed memory models on multiprocessors – which we do not address at all in this paper. Even in the sequential case, however, the lack of inherent structure in low-level code is a fundamental problem.

Machine code features unstructured control flow. A contiguous block of instructions potentially has many entry points and many exits, with the added complication that the same bytes may decode differently according to the entry point. Machine code is almost entirely untyped and higher-order, with no runtime tagging: any word in memory or a register may be treated as a scalar value, a pointer or a code pointer, and common coding patterns do make use of this flexibility: stealing bits in pointers, storing metadata at offsets from code pointers, computing branches, and so on. Finally, code and data live in the same heap, allowing code generation, self-modifying code and code examination, for example for interpreting instructions. The most basic abstractions, such as memory allocation or function calling, are not built in, but are conventions that must be specified, followed and verified at appropriate points. Furthermore, code that implements even the simplest of these abstractions, such as first-order function calls, uses features of machine code whose high-level analogues (higher-order, dynamically allocated local state) are challenging to reason about – and a subject of active research – even in very high-level languages such as ML.

Some logics, type systems and analyses for machine code deal with these complexities by imposing structure and restrictions on the code they deal with. For example, one can enforce a traditional basic block structure, hard-code memory allocation as a special pseudo-instruction or treat calling and a call-stack specially [5, 27, 31, 40]. Such techniques can work well for verifying code that looks like it came from a C compiler, but we would like something more generally applicable, able to verify smoothly higher-order code, systems code such as schedulers and allocators, and code that uses clever bit-level representation tricks. In previous work, for example on compiling a functional language to a rather idealized assembly language [7], one of us has proved useful results in Coq using a shallow embedding of step-indexed, separated predicates and relations, a notion of biorthogonality ('perping') for code pointers, explicit second-order quantification for framing, and a more-or-less ad hoc collection of lemmas for instructions, quantifier manipulation and entailment. Given sufficient effort, such an approach can undeniably be pushed through, but the proofs and specifications are very clumsy; although some of the connectives have respectable properties, there is certainly no sense that one is working in a well-structured program logic, with a well-behaved proof theory. Applying such a naive approach in the context of real machine code, with the above-mentioned messy complexities and in which we would clearly need to build numerous higher-level proof abstractions, seemed unlikely to work well.

Separation logics for higher-level languages, by contrast, do have a good proof theory. In particular, work on *higher-order* frame rules allows local reasoning about higher-order programs, allowing invariants to be framed onto commands in context by distributing them through the specifications of parameters [9]. A major goal of the work described here is to bring the power and concision of higher-order frame rules to reasoning about machine-code programs. At first sight, it may seem unclear how to incorporate even the first-order frame rule

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}}$$

into a system for reasoning about machine code. Firstly, the frame rule is typically justified using a global property of commands with respect to a semantics defined over partial heaps: if a command executes without faulting in some heap, then it does so in any extension of that heap and moreover, if it terminates it preserves the extension. Partial heaps in the semantics model a built-in allocator, but, as in our previous work in low-level code [6, 7], we do not wish to define the ground semantics (with respect to which we interpret specifications) using partial heaps: whatever memory is in the machine is there all the time, and the allocator is just another piece of code to be specified and verified in our framework.

Secondly, the postcondition of a triple corresponds to the single exit point of a first-order command. Machine code fragments do not have single exits, or even a natural, local notion of terminating execution. We are ultimately concerned with the observable behaviour of whole programs, and do not wish to restrict ourselves to a form of specification that relies on the non-observable, intensional property of reaching a particular intermediate program counter value. We thus take our basic form of safety specifications to be one that only involves a precondition: execution from a given address in a state satisfying the precondition is safe. As Chlipala [16] observes, it is not obvious how to attach a frame soundly to such a specification. We address these problems by going beyond a shallow embedding of specifications of individual program points, to an embedding of a fully-fledged specification logic [23, 34], making the context within which code fragments are proved explicit, and with a semantics that captures (but a surface notation which hides) the way in which frames are preserved.

The specification logic allows one to work with subtle patterns of invariant preservation, but does not impose particular forms of specification. Rather, it provides building blocks from which more complex patterns, including Hoare triples, may be built. The rich, well-behaved theory of the core logic allows derived rules for new forms of specification to be expressed and proved concisely.

We have formalized our specification logic in the Coq proof assistant, and instantiated it for the particular case of a model for sequential x86 machine code. Our formalization also includes a range of reflective tactics for solving separation entailments and performing specification logic proofs at a high level of abstraction. This paper mainly discusses the logic in a machine-independent way, but we use the x86 instantiation for examples and motivation.

In summary, the contributions of this work include:

1. A separation logic for unstructured machine code that supports both first- and higher-order frame rules.
2. Accounts of specification-level connectives including framing, a 'read-only' frame, a 'later' modality and a full range of intuitionistic connectives, all with good logical properties.
3. Examples of higher-level patterns, such as Hoare triples, and associated proof rules being defined smoothly within the logic.
4. An certified assembler, supporting convenient macro definitions with internal label generation and natural derived proof rules, with examples including while-program constructs and procedure calling.

[1] Logics for high-level languages often ignore fixed-length arithmetic, even when that is what is provided by real implementations.

5. A semantics involving no instrumention or other modifications to the underlying machine model. Memory can be total, and step-counting and auxiliary variables happen only in the logic.
6. All this is formalized in Coq, with an instantiation for x86 machine code and tactic support for high-level proving. The formalization is available via the authors' web pages.

2. Machine model

Our separation logic is not tied to any particular machine architecture, but in order to illustrate its application we will be presenting examples from 32-bit x86, the architecture for which we have built a model in Coq.[2] In this section we present enough concrete detail of this model to support subsequent sections.

We have modelled a subset of the 32-bit x86 instruction set, considering sequential execution only, but treating memory, registers and flags in sufficient detail to obtain accurate specification of its behaviour.

Machine words and arithmetic We model n-bit machine words simply as n-nary tuples of boolean values, deploying an indexed type in Coq for the purpose. The x86 architecture makes various use of 8-bit (BYTE), 16-bit (WORD), 32-bit (DWORD) and 64-bit (QWORD) values, and so nat-dependent types in Coq are a boon to specification. Logical and arithmetic operations are defined directly in terms of bits, although to prove useful properties of arithmetic it proved handy to map words into arithmetic modulo 2^n, making use of the `ssreflect` library for algebraic identities [20].

Machine state The state of the machine is described by a triple of registers, flags and memory state:

$$\begin{aligned}\mathbb{S} = {} & (\mathsf{reg} \to \mathsf{DWORD}) \times \\ & (\mathsf{flag} \to \{\mathsf{true}, \mathsf{false}, \mathsf{undef}\}) \times \\ & (\mathsf{DWORD} \to (\mathsf{BYTE} \uplus \{\mathsf{unmapped}\}))\end{aligned}$$

Register state is a straightforward mapping from the x86's nine core registers (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP and EIP) to 32-bit values. The EIP instruction pointer register is the x86's program counter and points to the next instruction to be decoded by the processor. Rather than model the special EFLAGS as a monolithic register, we split it up into boolean-valued flags. The undef value represents the undefined state in which many instructions leave flags. Any dependence of execution on an undefined state, such as in a conditional branch instruction, is then treated as unspecified behaviour. Memory is modelled straightforwardly as 32-bit-addressable bytes, with the possibility that any byte might be missing or inaccessible. For now, we are not interested in finer distinctions such as read-only or no-execute, though it would be a simple matter to incorporate these notions. It is however important to note that the 'partiality' of memory has nothing to do with the partial states of separation logic; indeed we could choose to model and fully specify the x86 support for fault handling, using the very same logic.

Instructions Any machine model must of course include a datatype of instructions. The x86 instruction set is notoriously large and baroque but by careful subsetting and factoring our datatype is made reasonably concise. Figure 1 presents the instruction datatype, with only some names changed for the purposes of this paper.

Instruction decoding The x86 instruction format is also complex, being variable in length and not canonical: a single instruction can sometimes be encoded in many ways. We have implemented an instruction decoder as a partial function

$$\mathsf{decode} : \mathsf{DWORD} \times (\mathsf{DWORD} \to (\mathsf{BYTE} \uplus \{\mathsf{unmapped}\})) \rightharpoonup \mathsf{DWORD} \times \mathsf{Instr}$$

such that if $\mathsf{decode}(i, m) = (j, \iota)$ then memory m from address i up to but not including address j is defined and decodes to instruction ι. The decoder reads memory incrementally, returning an undefined value if memory is inaccessible or out of range, or if the contents do not describe an instruction in our chosen subset. (There is no need to specify explicitly a maximum instruction length.) Lifting decode to machine states, and threading the updating of the EIP register through, yields a partial function in $\mathbb{S} \rightharpoonup \mathbb{S} \times \mathsf{Instr}$.

Instruction execution Instruction execution is given by a partial function on states $\mathbb{S} \times \mathsf{Instr} \rightharpoonup \mathbb{S}$, which when composed with instruction decoding gives rise to a small-step transition function on machine states $\mathsf{step} : \mathbb{S} \rightharpoonup \mathbb{S}$. When this function is undefined, it means that either a fault occurred (such as the decoding of an illegal instruction or an access to unmapped memory), or behaviour is simply unspecified (such as branching on an undefined flag).

$\mathsf{Typeof}\ d = \text{if } d \text{ then } \mathsf{DWORD} \text{ else } \mathsf{BYTE}$

$\mathsf{Scale} = \mathsf{S}_1 \mid \mathsf{S}_2 \mid \mathsf{S}_4 \mid \mathsf{S}_8$

$\mathsf{MemSpec} = \mathsf{Reg} \times \mathsf{option}\ (\mathsf{NonSPReg} \times \mathsf{Scale}) \times \mathsf{DWORD}$

$\mathsf{RegMem} = \mathsf{RegMemR}\ (r{:}\mathsf{Reg}) \mid \mathsf{RegMemM}(ms{:}\mathsf{MemSpec})$

$\mathsf{RegImm}\ d = \mathsf{RegImmI}\ (c{:}\mathsf{Typeof}\ d) \mid \mathsf{RegImmR}\ (r{:}\mathsf{Reg})$

$\mathsf{Src} = \mathsf{SrcI}\ (c{:}\mathsf{DWORD}) \mid \mathsf{SrcM}\ (ms{:}\mathsf{MemSpec}) \mid \mathsf{SrcR}(r{:}\mathsf{Reg})$

$\mathsf{DstSrc}\ d =$
$\mid \mathsf{RR}\ (dst{:}\mathsf{Reg})\ (src{:}\mathsf{Reg})$
$\mid \mathsf{RM}\ (dst{:}\mathsf{Reg})\ (src{:}\mathsf{MemSpec})$
$\mid \mathsf{MR}\ (dst{:}\mathsf{MemSpec})\ (src{:}\mathsf{Reg})$
$\mid \mathsf{RI}\ (dst{:}\mathsf{Reg})\ (src{:}\mathsf{Typeof}\ d)$
$\mid \mathsf{MI}\ (dst{:}\mathsf{MemSpec})\ (src{:}\mathsf{Typeof}\ d)$

$\mathsf{BinOp} = \mathsf{ADC} \mid \mathsf{ADD} \mid \mathsf{AND} \mid \mathsf{CMP} \mid \mathsf{OR} \mid \mathsf{SBB} \mid \mathsf{SUB} \mid \mathsf{XOR}$

$\mathsf{UnaryOp} = \mathsf{INC} \mid \mathsf{DEC} \mid \mathsf{NOT} \mid \mathsf{NEG} \mid \mathsf{POP}$

$\mathsf{BitOp} = \mathsf{BT} \mid \mathsf{BTC} \mid \mathsf{BTR} \mid \mathsf{BTS}$

$\mathsf{ShiftOp} = \mathsf{ROL} \mid \mathsf{ROR} \mid \mathsf{RCL} \mid \mathsf{RCR} \mid \mathsf{SHL} \mid \mathsf{SHR} \mid \mathsf{SAL} \mid \mathsf{SAR}$

$\mathsf{Count} = \mathsf{ShiftCL} \mid \mathsf{ShiftImm}\ (b{:}\mathsf{BYTE})$

$\mathsf{Condition} = \mathsf{O} \mid \mathsf{B} \mid \mathsf{Z} \mid \mathsf{BE} \mid \mathsf{S} \mid \mathsf{P} \mid \mathsf{L} \mid \mathsf{LE}$

$\mathsf{Instr} =$
$\mid \mathsf{UOP}\ (d{:}\mathsf{bool})\ (op{:}\mathsf{UnaryOp})\ (dst{:}\mathsf{RegMem})$
$\mid \mathsf{BOP}\ d\ (op{:}\mathsf{BinOp})\ (ds{:}\mathsf{DstSrc}\ d)$
$\mid \mathsf{BITOP}\ (op{:}\mathsf{BitOp})\ (dst{:}\mathsf{RegMem}\ \mathsf{false})$
$\mid \mathsf{TEST}\ d\ (dst{:}\mathsf{RegMem})\ (src{:}\mathsf{RegImm}\ d)$
$\mid \mathsf{MOV}\ d\ (ds{:}\mathsf{DstSrc}\ d)$
$\mid \mathsf{SHIFTOP}\ (d{:}\mathsf{bool})\ (op{:}\mathsf{ShiftOp})\ (dst{:}\mathsf{RegMem})\ (c{:}\mathsf{Count})$
$\mid \mathsf{MUL}\ (src{:}\mathsf{RegMem})$
$\mid \mathsf{LEA}\ (reg{:}\mathsf{Reg})\ (src{:}\mathsf{RegMem})$
$\mid \mathsf{JCC}\ (cc{:}\mathsf{Condition})\ (dir{:}\mathsf{bool})\ (tgt{:}\mathsf{DWORD})$
$\mid \mathsf{PUSH}\ (src{:}\mathsf{Src})$
$\mid \mathsf{POP}\ (dst{:}\mathsf{RegMem})$
$\mid \mathsf{CALL}\ (src{:}\mathsf{Src}) \mid \mathsf{JMP}\ (src{:}\mathsf{Src})$
$\mid \mathsf{RET}\ (size{:}\mathsf{WORD})$
$\mid \mathsf{CLC} \mid \mathsf{STC} \mid \mathsf{CMC} \mid \mathsf{HLT}$

Figure 1. Instruction datatype

[2] There is no particular reason for choosing x86 over x64 or ARM.

3. Assertion logic

3.1 Partial states

Assertions in separation logic describe a subset, or 'footprint', of the machine state. For high-level imperative programs with dynamic allocation this footprint consists of a subset of the heap. Indeed a common idiom is to prove that some code starts or finishes with an 'empty' heap.

Here, there is no such thing: we have the whole machine at our disposal, and we must carve out our own abstractions such as heaps or stacks, so the footprint is simply that part of the state that we care about right now. We also find it useful to use separation in describing the manipulation of registers and flags, and so define *partial* states as follows, noting the resemblance to the definition of total states in Section 2.

$$\begin{aligned}\Sigma = {} & (\mathsf{reg} \rightharpoonup \mathsf{DWORD}) \times \\ & (\mathsf{flag} \rightharpoonup \{\mathsf{true}, \mathsf{false}, \mathsf{undef}\}) \times \\ & (\mathsf{DWORD} \rightharpoonup (\mathsf{BYTE} \uplus \{\mathsf{unmapped}\}))\end{aligned}$$

There is a partial binary operation $\uplus$ on elements of Σ, defined when its operands have disjoint domains on all three tuple components and yielding a tuple with the union of each of the maps. This makes $(\Sigma, \uplus)$ a *separation algebra* [15]; i.e., a partial commutative monoid.

3.2 Assertion logic

An assertion is a predicate on partial states:

$$\mathsf{asn} \triangleq \mathcal{P}(\Sigma)$$

Since $(\Sigma, \uplus)$ is a separation algebra, its powerset asn forms a complete boolean BI algebra, i.e., a model of the assertion language of classical separation logic, where the connectives are defined in the standard way [15]:

$$\begin{aligned}\forall x{:}T.\ P(x) &\triangleq \textstyle\bigcap_{x:T} P(x) \qquad \exists x{:}T.\ P(x) \triangleq \textstyle\bigcup_{x:T} P(x) \\ P \Rightarrow Q &\triangleq \{\sigma \mid \sigma \in P \Rightarrow \sigma \in Q\} \qquad \mathsf{emp} \triangleq \{([],[],[])\} \\ P * Q &\triangleq \{\sigma \mid \exists \sigma_1, \sigma_2.\ \sigma = \sigma_1 \uplus \sigma_2 \wedge \sigma_1 \in P \wedge \sigma_2 \in Q\} \\ P \mathrel{-\!\!*} Q &\triangleq \{\sigma_2 \mid \forall \sigma_1.\ \forall \sigma = \sigma_1 \uplus \sigma_2.\ \sigma_1 \in P \Rightarrow \sigma \in Q\}\end{aligned}$$

The propositional connectives $(\wedge, \top)$ and $(\vee, \bot)$ are just binary and nullary special cases of $\forall$ and $\exists$ respectively. As usual, entailment is defined as $P \vdash Q \triangleq P \subseteq Q$, and we write $\vdash P$ for $\top \vdash P$ and $P \equiv Q$ whenever $P \vdash Q$ and $Q \vdash P$.

There is a notion of points-to [35] for registers and for flags:

$$r \mapsto v \triangleq \{([r \mapsto v], [], [])\} \qquad f \mapsto b \triangleq \{([], [f \mapsto b], [])\}$$

The meaning of the points-to assertion for memory, $i..j \mapsto v$, depends on the type of v; this is done using a *type class* [37] in our Coq implementation. For BYTE and DWORD types, points-to means that memory from address i to j contains that value. In these cases, j is uniquely determined to be $i + 1$ or $i + 4$ respectively. For syntactic assembly instructions ι, it means that the memory at $i..j$ decodes to ι. In other words, $\mathsf{decode}(i, m) = (j, \iota)$ where m is the memory component of the state. Instruction encoding is not unique, so more than one byte sequence in memory may decode to the same ι. We write $i \mapsto v$ to mean $\exists j.\ i..j \mapsto v$.

Discussion. Another option would have been to let the registers and flags behave like the 'stack' in traditional separation logic [35] and not split them over $*$. This is the approach taken by Shao et al. [14, 31] and Chlipala [16], but it leads to the side condition on the frame rules that the program may not modify any registers mentioned by the frame. In a setting where programs have multiple entry and exit points and may be self-modifying, it is not even clear what that side condition means or how to check it, so we instead make registers and flags split across $*$, following Myreen et al. [29].

4. Specification logic

4.1 Safety

We extend the single-step partial function $\mathsf{step} : \mathbb{S} \rightharpoonup \mathbb{S}$ to a function $\mathsf{run} : \mathbb{N} \times \mathbb{S} \rightharpoonup \mathbb{S}$, where $\mathsf{run}(k, s)$ is the state that results from successful execution of k instructions starting from state s.

Unlike the high-level languages typically modelled with Hoare logics, a CPU has no natural notion of finishing a computation. It will run forever until it either faults or loses power[3]. This means that we cannot apply the standard Hoare-logic approach of describing a computation by a precondition and a postcondition since there is no meaningful time to check the postcondition.

Instead, specifications revolve around *safety*. We characterise the safe machine configurations as the set of pairs $(k, P) : \mathbb{N} \times \mathsf{asn}$ such that the machine will run for at least k steps without faulting if started from a state in P:

$$\mathsf{safe} \triangleq \{(k, P) \mid \forall \sigma \in P.\ \forall s \sqsupseteq \sigma.\ \exists s' : \mathbb{S}.\ \mathsf{run}(k, s) = s'\}$$

The relation $s \sqsupseteq \sigma$ states that all the mappings in σ are also found in s. This is how we connect the partial states found in assertions to the total states executed by the machine.

Example 1. It is safe to sit in a tight loop forever. That is,

$$\forall k, i.\ (k, (\mathsf{EIP} \mapsto i * i \mapsto \mathsf{jmp}\ i)) \in \mathsf{safe}$$

The EIP register is the instruction pointer, and jmp i is an unconditional jump to address i. The proof goes by induction on k. $\diamond$

The number k plays the role of a *step index* [2]. We are ultimately always interested in proving computations safe for an arbitrary number of steps, but exposing an intermediate step index gives us a value on which we can do induction.

As a running example, we will attempt to specify the unconditional jump instruction. We can show that for all i, a, k, R,

$$\begin{aligned}&(k, Q * R) \in \mathsf{safe} \Rightarrow (k + 1, P * R) \in \mathsf{safe} && \text{where} \\ &P = (\mathsf{EIP} \mapsto i * i \mapsto \mathsf{jmp}\ a) && \text{and} \\ &Q = (\mathsf{EIP} \mapsto a * i \mapsto \mathsf{jmp}\ a)\end{aligned}$$

In words, if you need to show that $P * R$ is a safe configuration for $k + 1$ steps, it suffices to show that $Q * R$ is safe for k steps. When a specification follows this pattern, we can think of P as a precondition, Q as a postcondition, and R as a frame.

The specification does not say that $Q * R$ will ever hold. Rather, it requires that if $Q * R$ does hold, then we are in a safe configuration. This can be seen as a CPS version of Hoare logic, which is appropriate for machine code since nothing ever returns or finishes at this level [5, 31, 38].

We will refine this specification in later examples as we develop constructions at higher levels of abstraction.

4.2 Specification logic

Reasoning directly about membership of safe is awkward since the step index and frame are explicit and visible even though their use always follows the same pattern. The solution is to instead consider safe as a formula in a *specification logic*. We define a specification to be a set of of (k, P)-pairs that is closed under decreasing k and under starring arbitrary assertions onto P:

$$\mathsf{spec} \triangleq \{S \subseteq \mathbb{N} \times \mathsf{asn} \mid \forall (k, P) \in S, k' \leq k, R.\ (k', P * R) \in S\}$$

[3] Even when it faults, it will typically reboot and so keep running, but this behaviour is outside of our model.

Intuitively, a specification $S : \mathsf{spec}$ describes how many steps the machine has to execute before it no longer holds and what frames the execution will preserve. This idea comes from the work of Birkedal, Torp-Smith and Yang on higher-order frame rules [8, 10], and spec is essentially a step-indexed version of Krishnaswami's specification logic model [23].

The definition of spec is such that $\mathsf{safe} \in \mathsf{spec}$. Furthermore, spec is a *complete Heyting algebra* and thus a model of intuitionistic logic. This gives us a notion of entailment ($\vdash \triangleq \subseteq$) and the logical connectives ($\forall, \exists, \wedge, \vee, \top, \bot, \Rightarrow$) with the expected rules. The definitions of the connectives follow a standard Kripke model:

$$\forall x{:}T.\ S(x) \triangleq \bigcap_{x:T} S(x) \qquad \exists x{:}T.\ S(x) \triangleq \bigcup_{x:T} S(x)$$
$$S \Rightarrow S' \triangleq \{(k, P) \mid \forall k' \leq k.\ \forall R.\ (k', P * R) \in S \Rightarrow (k', P * R) \in S'\}$$

Again, the propositional connectives ($\wedge, \top$) and ($\vee, \bot$) are just binary and nullary special cases of $\forall$ and $\exists$ respectively.

Notice how the semantics of $\Rightarrow$ requires arbitrary frames to be preserved across the implication. This was not a choice we made – it is the only definition that makes $\Rightarrow$ be the right adjoint of $\wedge$, and it falls out of giving standard Kripke semantics.

We also get two new connectives: the *later* connective $\triangleright$ and the *frame* connective $\otimes$. We will define and discuss these in the next two subsections. They will enable us to state the specification of the jmp instruction from Section 4.1 more succinctly:

$$\triangleright\, \mathsf{safe} \otimes (\mathsf{EIP} \mapsto a * i \mapsto \mathsf{jmp}\ a) \vdash \mathsf{safe} \otimes (\mathsf{EIP} \mapsto i * i \mapsto \mathsf{jmp}\ a) \quad (1)$$

We can even factor out the duplicated part of the assertion and just write

$$\vdash (\triangleright\, \mathsf{safe} \otimes (\mathsf{EIP} \mapsto a) \Rightarrow \mathsf{safe} \otimes (\mathsf{EIP} \mapsto i)) \otimes i \mapsto \mathsf{jmp}\ a$$

or, informally, reading from right to left: 'given that i points to instruction jmp a, it is safe to execute with the instruction pointer set to i if it is later safe to execute with the instruction pointer set to a'.

4.3 Frame connective

Following the literature on higher-order frame rules [8, 10, 23], we define the *frame connective* $\otimes : \mathsf{spec} \times \mathsf{asn} \to \mathsf{spec}$ as

$$S \otimes R \triangleq \{(k, P) \mid (k, P * R) \in S\}$$

This is also known as the invariant extension connective [8] because an intuitive reading of $S \otimes R$ is that the computation described by S is allowed to additionally depend on and maintain the invariant R. Note that, by unpacking the definitions,

$$\vdash \mathsf{safe} \otimes P \quad \text{iff} \quad \forall k, R.\ (k, P * R) \in \mathsf{safe}$$

relating judgements in the specification logic with the safety of executions. Since we defined spec such that any S can be extended by any invariant, we can immediately prove the higher-order frame rule:

$$\frac{}{S \vdash S \otimes R}\ \textsc{Frame}$$

The frame connective distributes over all other connectives, including $\triangleright$ and itself. That means, for example, that

$$\frac{}{(S \Rightarrow S') \otimes R \ \equiv\ S \otimes R \Rightarrow S' \otimes R}\ \otimes\text{-}\Rightarrow$$

It also interacts with emp and $*$ as follows.

$$\frac{}{S \otimes \mathsf{emp} \ \equiv\ S}\ \otimes\text{-EMP}$$

$$\frac{}{S \otimes R_1 \otimes R_2 \ \equiv\ S \otimes (R_1 * R_2)}\ \otimes\text{-}*$$

Example 2. We can now start to see why FRAME should be thought of as a frame rule. Assume we have proved for some P and Q that

$$\vdash \mathsf{safe} \otimes Q \Rightarrow \mathsf{safe} \otimes P.$$

Then by FRAME, $\otimes$-$\Rightarrow$ and $\otimes$-$*$, we can derive

$$\vdash \mathsf{safe} \otimes (Q * R) \Rightarrow \mathsf{safe} \otimes (P * R).$$

Visually this looks like the standard frame rule, and it performs the same function: to extend both pre- and post-condition by an invariant. ◇

The formula $S \otimes R$ is covariant in S with respect to entailment, meaning that

$$\frac{S \vdash S'}{S \otimes R \vdash S' \otimes R}\ \otimes\text{-}\vdash$$

The variance in R is more complicated and will be discussed in Section 7.1.

Example 3. To illustrate informally how FRAME generalises the standard first-order frame rule, consider a program in a high-level functional programming language $f_1 : (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}$, whose specification is, for some particular P, Q and R,

$$\forall g.\ \{P * R\}\ g\ ()\ \{Q * R\} \Rightarrow \{P * R\}\ f_1(g)\ \{Q * R\}$$

That is, f_1 forwards the specification of g. Most likely, f_1 simply applies its argument to the unit value, but assume that it has been verified separately and we should not see its implementation.

If we have g_1 with specification $\{P\}\ g_1\ ()\ \{Q\}$, we cannot immediately apply $f_1(g_1)$ since the specification does not match what f_1 requires. However, we can apply the ordinary frame rule to deduce that g_1 also has the specification $\{P * R\}\ g_1\ ()\ \{Q * R\}$, and then we can call $f_1(g_1)$ if we are in a state satisfying $P * R$.

Now instead consider an f_2 with the specification

$$\forall g.\ \{P\}\ g\ ()\ \{Q\} \Rightarrow \{P\}\ f_2(g)\ \{Q\}$$

and a g_2 with specification $\{P * R\}\ g_2\ ()\ \{Q * R\}$. It is impossible with just the standard frame rule to call $f_2(g_2)$ since the specification of g_2 cannot be refined to match what is assumed by f_2. But with the higher-order frame rule, we can instead refine the specification of f_2 to be

$$\forall g.\ (\{P\}\ g\ ()\ \{Q\} \Rightarrow \{P\}\ f_2(g)\ \{Q\}) \otimes R \equiv$$
$$\forall g.\ \{P\}\ g\ ()\ \{Q\} \otimes R \Rightarrow \{P\}\ f_2(g)\ \{Q\} \otimes R \equiv$$
$$\forall g.\ \{P * R\}\ g\ ()\ \{Q * R\} \Rightarrow \{P * R\}\ f_2(g)\ \{Q * R\}$$

It is now compatible with our g_2.

Without the higher-order frame rule, we would have had to either re-verify the implementation of f_2 or generalise the original specification of f_2 to explicitly quantify over all possible frames that may be threaded through. The latter option is essentially what the definition of spec does, but this is invisible and implicit. ◇

Using $\otimes$ to give concise and modular specifications to higher-order functions is as important here as in any other separation logic, but that is not our main reason for including $\otimes$. We do it because it allows our logic to have a frame rule despite the program being unstructured and low-level. Chlipala [16] uses explicit second-order quantification in place of a frame rule, whilst Shao et al. [14, 31] have a frame rule that only applies to judgements in a very restrictive specification logic; in particular, it does not apply directly to specifications of function pointers.

4.4 Later connective

Just as we hide the explicit frames using $\otimes$, we hide the step indexes using the *later* connective, $\triangleright$. This is a trick pioneered by Nakano [30] that exploits the fact that we are never interested in the absolute number of steps but only that they are the same or differ

by exactly one between two specification formulas. We define

$$\triangleright S \triangleq \{(k, P) \mid \forall k' < k.\ (k', P) \in S\}$$

Because any $S : \mathsf{spec}$ is closed under decreasing steps, an equivalent definition is that $(0, P) \in \triangleright S$ for all P, and $(k+1, P) \in \triangleright S$ iff $(k, P) \in S$. The closure under decreasing steps is expressed logically as the rule

$$\frac{}{S \vdash \triangleright S}\ \triangleright\text{-WEAKEN}$$

As mentioned in Section 4.1, the purpose of step indexes is to serve as a handle for induction. We can phrase the induction principle on natural numbers using the following rule [3, 30], which is named for its similarity to a corresponding rule in Gödel-Löb logic [3].

$$\frac{\triangleright S \vdash S}{\vdash S}\ \text{LÖB}$$

The Löb rule is a reformulation of the strong induction principle for natural numbers: if $(\forall k' < k.\ P(k')) \Rightarrow P(k)$ for all k, then $P(n)$ holds for any n. It is a powerful rule in that it almost allows assuming the formula one wants to prove, except that the assumption may only be used after taking one step of computation.

Example 4. Recall the specification of a tight loop from Example 1. We can now express and prove that inside the specification logic in just two steps:

$$\dfrac{\dfrac{}{\triangleright\, \mathsf{safe} \otimes (\mathsf{EIP} \mapsto i * i \mapsto \mathsf{jmp}\ i)}\ (1)}{\vdash \mathsf{safe} \otimes (\mathsf{EIP} \mapsto i * i \mapsto \mathsf{jmp}\ i)}\ \text{LÖB} \qquad \diamond$$

The $\triangleright$ connective distributes over every other connective we have mentioned except for $\bot$ and existential quantification over empty types.

Discussion. Like we saw in the rule for jmp (Equation (1)), every step of computation allows us to relax our remaining proof obligation by adding a $\triangleright$. For example, we could prove that

$$\vdash (\triangleright\triangleright \mathsf{safe} \otimes (\mathsf{EIP} \mapsto j) \Rightarrow \mathsf{safe} \otimes (\mathsf{EIP} \mapsto i)) \otimes i..j \mapsto (\mathsf{nop}; \mathsf{nop}) \quad (2)$$

where nop is the no-operation instruction. There are *two* $\triangleright$'s on the 'postcondition' of (2) because it takes two steps of computation to get there. It turns out, however, that it is never useful to have more than one $\triangleright$ applied to a specification since the purpose of step indexes is to do induction, and induction will always give us the necessary assumptions on the immediate predecessor of the number of interest.

Furthermore, we have found that $\triangleright$ is not necessary in code that only moves forward. Löb induction only makes sense when verifying loops, and a loop requires some form of backward jump unless we consider highly-contrived self-modifying code. Therefore, in practice, we would state (2) without any $\triangleright$-connective at all.

4.5 Read-only frame

The instruction rules we have discussed so far are too weak for some purposes. Recall the rule for jmp:

$$\vdash (\triangleright\, \mathsf{safe} \otimes (\mathsf{EIP} \mapsto a) \Rightarrow \mathsf{safe} \otimes (\mathsf{EIP} \mapsto i)) \otimes i \mapsto \mathsf{jmp}\ a$$

Because the meaning of $i \mapsto \mathsf{jmp}\ a$ is only that the memory starting at i decodes to jmp a, the rule would be satisfied in a semantics where the jmp instruction not only performed the jump but also replaced its own machine code in memory with a different byte sequence that also decoded to jmp a. This would be a problem for programs whose code needs to stay unmodified; e.g., to verify that the checksum of the code remains the same.

Our solution is to make this specification more precise by employing the *read-only frame* connective, defined as

$$S \oslash R \triangleq \forall \sigma \in R.\ S \otimes \{\sigma\}.$$

A more precise specification for jmp is then

$$\vdash (\triangleright\, \mathsf{safe} \otimes (\mathsf{EIP} \mapsto a) \Rightarrow \mathsf{safe} \otimes (\mathsf{EIP} \mapsto i)) \oslash i \mapsto \mathsf{jmp}\ a$$

Intuitively, $S \oslash R$ requires S not only to preserve the truth of R but to leave unmodified the underlying state fragment that made R true. The state may be changed temporarily, just as R might be broken, as long as it is restored at the end of the computation described by S.

Like for $\otimes$, there is a frame rule:

$$\frac{}{S \vdash S \oslash R}\ \text{FRAME-RO}$$

The $\oslash$ connective does not distribute over every other connective like $\otimes$ does, but it does distribute over $\forall, \wedge, \top, \otimes, \oslash, \triangleright$. It only distributes in one direction over $\exists$ and $\Rightarrow$:

$$\overline{(\exists a.\ S(a) \oslash R) \vdash (\exists a.\ S(a)) \oslash R}$$

$$\overline{(S \Rightarrow S') \oslash R \vdash S \oslash R \Rightarrow S' \oslash R}$$

The formula $S \oslash R$ is covariant in S and contravariant in R with respect to entailment, meaning that

$$\frac{S \vdash S' \qquad R' \vdash R}{S \oslash R \vdash S' \oslash R'}\ \oslash\text{-}\vdash$$

Another convenient property is that existential quantifiers can be moved in and out of the frame:

$$\overline{S \oslash (\exists x.\ R(x)) \ \equiv\ \forall x.\ S \oslash R(x)}$$

These last two properties of $\oslash$ about variance and commuting with existentials do not generally hold for $\otimes$. The cases where they hold are discussed in Sections 7.1 and 7.2. We explore further properties of $\oslash$ in Section 7.3.

Discussion. This connective is reminiscent of fractional permissions [12] but more coarse-grained and light-weight. We mention connections to other notions of weak ownership [22] in Section 9.

Our definition of $\oslash$ may not be the only good one, but we have examined three other candidate definitions and found that the one given above had the most convenient properties for our purposes. The candidates relate to each other as follows.

$$\begin{aligned} \forall R' \vdash R * \top.\ S \otimes R' \quad &\vdash \\ \forall R' \vdash R.\ S \otimes R' \quad &\vdash \\ \forall \sigma \in (R * \top).\ S \otimes \{\sigma\} \quad &\equiv \\ \forall \sigma \in R.\ S \otimes \{\sigma\} & \end{aligned}$$

5. High-level assembly code

5.1 Basic blocks

Using safe and the connectives discussed so far, we can specify code with multiple entry points and exit points, jumps to code pointers, self-modification, and so on. In practice, though, most code is much simpler. For code that behaves like a *basic block*, with control flow always coming in at the top and going out at the bottom, we can describe its behaviour with a Hoare triple, defined in the specification logic as:

$$\begin{aligned} \{P\}\ c\ \{Q\} \triangleq \forall i, j.\ (&\mathsf{safe} \otimes (\mathsf{EIP} \mapsto j * Q) \Rightarrow \\ &\mathsf{safe} \otimes (\mathsf{EIP} \mapsto i * P)) \oslash i..j \mapsto c \end{aligned}$$

Example 5. The instruction mov r, v (move literal v to register r) can be specified as $\vdash \{r?\}\ \mathsf{mov}\ r, v\ \{r \mapsto v\}$, where $r?$ is

shorthand for $\exists v.\ r \mapsto v$. This is much more compact and readable than writing the specification in terms of safe. ◇

This triple satisfies the structural rules we expect from a Hoare triple in separation logic:

$$\frac{P \vdash P' \quad S \vdash \{P'\}\ c\ \{Q'\} \quad Q' \vdash Q}{S \vdash \{P\}\ c\ \{Q\}}$$

$$\frac{S \vdash \forall x.\ \{P(x)\}\ c\ \{Q\}}{S \vdash \{\exists x.\ P(x)\}\ c\ \{Q\}} \qquad \frac{S \vdash \{P\}\ c\ \{Q\}}{S \vdash \{P * R\}\ c\ \{Q * R\}}$$

The frame rule for the triple follows from FRAME and the fact that $\otimes$ distributes into the triple:

$$\frac{}{\{P\}\ c\ \{Q\} \otimes R\ \equiv\ \{P * R\}\ c\ \{Q * R\}}\ \otimes\text{-TRIPLE}$$

There is no rule of conjunction for the triple since this would be unsound in the presence of FRAME [8].

Discussion. This kind of triple is certainly not the only useful one. One could also adapt the position-indexed triples of Myreen and Gordon [29] to this setting, allowing use of the triple metaphor in specifying code with multiple entry and exit points. It would be a matter of taste whether this seemed more convenient to work with than reasoning directly in terms of safe.

The triple defined here can be thought of as encoding a very simple calling convention: inlining; i.e., concatenation of code. We envision defining triples for other conventions as needed and proving similar properties about them. See Section 5.5 for another example.

It is a valid question to ask why there is no $\triangleright$ on the postcondition part of the triple so it would read $\triangleright\ \mathsf{safe} \otimes (\mathsf{EIP} \mapsto j * Q)$. It would give stronger specifications for single instructions like in Example 5, but as discussed in Section 4.4, it would also be unnecessary since control flow always moves forward in a triple. We will also see in Section 5.3 that there are useful values of c that take no computation steps.

5.2 Rules for x86 instructions

With a variety of logical building blocks in place, we can give appealingly simple rules for x86 instructions. These split into instructions that do not touch the instruction pointer, for which we can use the Hoare-triple form, and control flow instructions, for which we describe their effect on the instruction pointer explicitly.

Example 6. The following rule for 'add register indirect with offset' is a typical instance. Here d is a literal DWORD offset, and addition of two 32-bit values produces a pair (c, v) where v is the 32-bit (truncated) result, and c is the carry into bit 32.

$$\begin{array}{ll}\vdash & \{r_1 \mapsto v_1 * \mathsf{OF}? * \mathsf{SF}? * \mathsf{ZF}? * \mathsf{CF}? * \mathsf{PF}?\} \\ & \mathsf{add}\ r_1, [r_2 + d] \\ & \{r_1 \mapsto v * \mathsf{OF} \mapsto \neg(\mathrm{msb}\ v_1 \oplus \mathrm{msb}\ v_2) \oplus \mathrm{msb}\ v \\ & \quad * \mathsf{SF} \mapsto \mathrm{msb}\ v * \mathsf{ZF} \mapsto (v = 0) * \mathsf{CF} \mapsto c * \mathsf{PF} \mapsto \mathrm{lsb}\ v\} \\ & \otimes (r_2 \mapsto w * w + d \mapsto v_2) \\ & \text{where } v_1 + v_2 = (c, v)\end{array}$$

The $\neg$ and $\oplus$ operators are boolean negation and xor respectively. The instruction affects flags OF, SF, ZF, CF and PF whose values initially are arbitrary ($F?$ is shorthand for $\exists f.\ F \mapsto f$, where f may be undef). Notice the framing of invariant registers and memory. ◇

Example 7. For the jump-if-zero instruction, we specify two 'postconditions', the first for when the branch is taken, and the second for when it isn't.

$$\begin{array}{ll}\vdash & (\triangleright\ \mathsf{safe} \otimes (b \wedge \mathsf{EIP} \mapsto a * \mathsf{ZF} \mapsto b) \wedge \\ & \quad \mathsf{safe} \otimes (\neg b \wedge \mathsf{EIP} \mapsto j * \mathsf{ZF} \mapsto b) \\ & \quad \Rightarrow \mathsf{safe} \otimes (\mathsf{EIP} \mapsto i * \mathsf{ZF} \mapsto b)) \\ & \oslash\ i..j \mapsto \mathsf{jz}\ a\end{array}$$

Note the use of the *later* connective when the (possibly backwards) branch is taken. ◇

Our approach is to give a very general specification to each instruction and then on top of that provide convenience definitions for common cases. In a sense, our rules are therefore just a logical reformulation of the operational semantics, which may seem a bit unimpressive but turns out to be a strong platform on which to build higher-level layers of abstraction.

5.3 Instruction encoding and assembly language

We have implemented an encoder for syntactic instructions, and it has the property that

$$i..j \mapsto \mathsf{encode}(i, \iota) \vdash i..j \mapsto \iota$$

That is, if the memory at $i..j$ contains the sequence of bytes $\mathsf{encode}(i, \iota)$, then that memory will decode to the instruction ι. The instruction decoder referred to here is the same one that is part of the operational semantics for the machine. The encode function takes i as parameter because the encoding of x86 instructions is not generally position-independent.

This encoder is the main ingredient in our *assembler*: a certified and executable Coq function that takes a *program* as input and produces a list of bytes as output. A program is a value in the following inductive definition.

$$p ::= (\iota) \mid \mathsf{skip} \mid p; p \mid l\mathsf{:} \mid \mathsf{LOCAL}\ l;\ p$$

That is, a program is essentially a list of instructions with label markers 'l:' interspersed. A label l may be declared local to program p with the LOCAL $l;\ p$ construction. A label is simply a memory address; i.e., a 32-bit word, and it can therefore be used as an argument to jump instructions. The following is a closed program that loops forever.

$$\mathsf{LOCAL}\ l;\ l\mathsf{:}\ \mathsf{jmp}\ l$$

The LOCAL constructor in our Coq implementation has type $(\mathsf{DWORD} \to \mathsf{program}) \to \mathsf{program}$, so writing LOCAL $l;\ p$ is just syntactic sugar for $\mathsf{LOCAL}(\lambda l.\ p(l))$. The benefit of modelling label scopes with function spaces is that Coq handles all aspects of label naming transparently, including the necessary capture-avoidance and α-conversion. The downside is that it is not viable to statically rule out ill-formed programs, such as programs that place the same label more than once.

The assembler function, assemble, is partial and maps an address and a program to a sequence of bytes. It is undefined if the program is ill-formed. Where defined, it has the correctness property that

$$i..j \mapsto \mathsf{assemble}(i, p) \vdash i..j \mapsto p$$

Here, $i..j \mapsto p$ is defined recursively as follows.

$$\begin{array}{ll} i..j \mapsto (\iota) & \triangleq i..j \mapsto \iota \\ i..j \mapsto \mathsf{skip} & \triangleq i = j \wedge \mathsf{emp} \\ i..j \mapsto p_1; p_2 & \triangleq \exists i'.\ i..i' \mapsto p_1 * i'..j \mapsto p_2 \\ i..j \mapsto \mathsf{LOCAL}\ l;\ p & \triangleq \exists l.\ i..j \mapsto p(l) \\ i..j \mapsto l\mathsf{:} & \triangleq i = j = l \wedge \mathsf{emp} \end{array}$$

Recall that the definition of triples $\{P\}\ c\ \{Q\}$ in Section 5.1 did not require c to have a particular type; the definition and its rules are valid for any c that can occur on the right of a points-to. Thus, we can put a program p in a triple, and it turns out that the following rules hold.

$$\frac{}{\vdash \{P\}\ \mathsf{skip}\ \{P\}} \qquad \frac{S \vdash \{P\}\ p_1\ \{Q\} \quad S \vdash \{Q\}\ p_2\ \{R\}}{S \vdash \{P\}\ p_1; p_2\ \{R\}}$$

$$\frac{S \vdash \{P\}\ \iota\ \{Q\}}{S \vdash \{P\}\ (\iota)\ \{Q\}} \qquad \frac{S \vdash \forall l.\ \{P\}\ p(l)\ \{Q\}}{S \vdash \{P\}\ \mathsf{LOCAL}\ l;\ p\ \{Q\}}$$

There is no useful rule for the case of l: in a triple.

Example 8. We cannot specify jmp with a triple in any useful way, but we can specify the special case of a tight loop shown above:

$$\vdash \{\mathsf{emp}\}\ \mathsf{LOCAL}\ l;\ l\colon \mathsf{jmp}\ l\ \{\bot\}$$

The proof is by first applying the triple rule for LOCAL, then unfolding the definition of the triple and applying the result of Example 4. ◇

5.4 Assembly macros

A useful assembly language has not only labels but also macros; i.e., parameterised definitions that expand to instruction sequences when invoked. We get macros almost for free since assembly programs are written and parsed inside Coq and can be intermixed with all the features of its term language. This includes let-bindings, fixpoint computations, custom syntax, coercions, overloading and other features of a modern dependently-typed programming language.

An example of a very useful macro is the following definition of $\mathsf{while}(p_1, t, b, p_2)$, where p_1 is a loop test, p_2 is a loop body, t encodes the combination of processor flags to be branched on, and b is a boolean that indicates whether the test should be inverted.

$$\begin{aligned}\mathsf{while}(p_1, t, b, p_2) \triangleq\ & \mathsf{LOCAL}\ l_1, l_2;\\ & \quad \mathsf{jmp}\ l_1;\\ & l_2\colon\ p_2;\\ & l_1\colon\ p_1;\\ & \quad \mathsf{jcc}\ t, b, l_2\end{aligned}$$

The jcc instruction is the general *conditional jump* on x86. We see here how LOCAL lets us declare labels that will be fresh for every invocation of the while macro. Real-world macro assemblers also have that functionality, although the scope is usually tied to the nearest named macro or global label. Our Coq notations for assembly syntax, including LOCAL, are chosen to be compatible with *MASM*, the Microsoft assembler.

Macros such as while give us the usual convenience of not having to write similar code many times. But even better, it lets us avoid writing similar proofs many times. If the body and test can be specified in terms of a triple, then the loop as a whole also has a triple specification:

$$\frac{\begin{array}{l}S \vdash \{P\}\ p_1\ \{\exists b'.\ I(b') * \mathsf{cond}(t, b')\}\\ S \vdash \{I(b) * \mathsf{cond}(t, b)\}\ p_2\ \{P\}\end{array}}{S \vdash \{P\}\ \mathsf{while}(p_1, t, b, p_2)\ \{I(\neg b) * \mathsf{cond}(t, \neg b)\}}\ \textsc{While}$$

Here, $\mathsf{cond}(t, b)$ translates t, of type Condition from Figure 1, to an assertion that tests the relevant flags. For example, $\mathsf{cond}(\mathsf{Z}, b) = \mathsf{ZF} \mapsto\!\!\triangleright\ b$, where ZF is the *zero flag*. There are two loop invariants, P and I, representing the state before and after executing the test p_1 since this may have side effects.

The proof of the WHILE rule involves $\triangleright$-operators and the LÖB rule, but these technicalities do not leak out into the rule statement.

With if and while macros and the sequence operator on programs, we have the building blocks to easily write and verify programs with structured control flow. These constructs also facilitate using our assembly language as the target of a verified compiler from a structured language, which is something we hope to investigate more in future work.

5.5 Procedure calls

The triple $\{P\}\ c\ \{Q\}$ encodes and abstracts the often-occurring programming pattern of structured control flow. Another crucial pattern to capture is procedure calls. We will here show the theory of a very simple calling convention [29]: store the return address in register EDX and jump to the procedure entry point. The following macro calls the procedure whose code is at address f.

$$\begin{aligned}\mathsf{call}\ f \triangleq\ & \mathsf{LOCAL}\ i_{\text{ret}};\\ & \quad \mathsf{mov}\ \mathsf{EDX}, i_{\text{ret}};\\ & \quad \mathsf{jmp}\ f;\\ & i_{\text{ret}}\colon\end{aligned}$$

The calling convention does not specify how to pass arguments or return values; this is instead part of individual procedure specifications. A more realistic calling convention would maintain a stack of arguments and return addresses to allow deep call hierarchies and reentrancy, but this would clutter our examples with arithmetic side conditions because the stack has to be finite [29].

The following definition describes the behaviour of a procedure starting at f with precondition P and postcondition Q.

$$\begin{aligned}f \mapsto\!\!\triangleright \{P\}\{Q\} \triangleq \forall i_{\text{ret}}.\ \mathsf{safe} \otimes (\mathsf{EIP} \mapsto\!\!\triangleright\ i_{\text{ret}} * \mathsf{EDX}? * Q) \Rightarrow\\ \mathsf{safe} \otimes (\mathsf{EIP} \mapsto\!\!\triangleright\ f * \mathsf{EDX} \mapsto\!\!\triangleright\ i_{\text{ret}} * P)\end{aligned}$$

Recall that EDX? is shorthand for $\exists v.\ \mathsf{EDX} \mapsto\!\!\triangleright\ v$. This definition satisfies the usual rules for a triple-like formula, including

$$\frac{}{f \mapsto\!\!\triangleright \{P\}\{Q\} \otimes R\ \equiv\ f \mapsto\!\!\triangleright \{P * R\}\{Q * R\}}\ \otimes\text{-}\textsc{Proc}$$

In contrast with the triple defined in Section 5.1, this definition of a procedure specification does not mention the code stored at f. The code should be mentioned separately from its behaviour such that the footprint of the code covers both the caller and the callee.

The rule for calling a procedure looks fairly standard:

$$\frac{}{\triangleright f \mapsto\!\!\triangleright \{P\}\{Q\} \vdash \{P\}\ \mathsf{call}\ f\ \{Q\} \otimes \mathsf{EDX}?}\ \textsc{Call}$$

It reveals that EDX is overwritten as part of the calling convention. The $\triangleright$ modality on the premise, together with LÖB, permits recursion [3].

Example 9. This is the first of three examples to illustrate independent verification of caller and callee. Consider the following definition of a program that calls some procedure at f twice:

$$p_{\text{caller}}(f) \triangleq \mathsf{call}\ f;\ \mathsf{call}\ f$$

If the intention with this program is to compose it with a procedure that satisfies

$$S_{\text{callee}}(f) \triangleq \forall a.\ f \mapsto\!\!\triangleright \{\mathsf{EAX} \mapsto\!\!\triangleright\ a\}\{\mathsf{EAX} \mapsto\!\!\triangleright\ a + 2\},$$

then we can specify the caller as

$$S_{\text{callee}}(f) \vdash \{\mathsf{EAX} \mapsto\!\!\triangleright\ a\}\ p_{\text{caller}}(f)\ \{\mathsf{EAX} \mapsto\!\!\triangleright\ a + 4\} \otimes \mathsf{EDX}?$$

We can prove this specification directly from the program sequencing rule and CALL. No $\triangleright$ connective is put on the assumption since no recursion is intended. ◇

If a procedure body p is structured and returns at its very end, we can prove its specification through the following rule.

$$\frac{S \vdash \{P\}\ p\ \{Q\} \oslash \mathsf{EDX}?}{S \vdash f \mapsto\!\!\triangleright \{P\}\{Q\} \oslash f \mapsto\!\!\triangleright (p;\ \mathsf{jmp}\ \mathsf{EDX})}\ \textsc{Body}$$

In words, this means that calling f behaves as (P, Q) when in memory where the program $(p;\ \mathsf{jmp}\ \mathsf{EDX})$ is at address f, assuming we can prove the given triple, which is allowed to access EDX as long as it restores its value in the end.

Example 10. The following program almost satisfies S_{callee} as defined in Example 9.

$$p_{\text{callee}} \triangleq \mathsf{inc}\ \mathsf{EAX};\ \mathsf{inc}\ \mathsf{EAX};\ \mathsf{jmp}\ \mathsf{EDX}$$

We say *almost* because the inc instruction affects the status flags of the CPU as a side effect. The caller is not interested in the flags, but they have to be in the specification of p_{callee} since they

do get affected. Let flags be the assertion that all flags are of some (existential) value. Then we can prove

$$\vdash S_{\text{callee}}(f) \otimes \mathsf{flags} \oslash f \mapsto p_{\text{callee}}.$$

The proof is by applying BODY, whose conclusion matches the above specification after rewriting by $\otimes$-PROC. $\diamond$

The next example demonstrates how to compose a caller and a callee, even if the callee has a larger footprint than what the caller assumes. This shows how to execute the informal reasoning from Example 3 in our logic.

Example 11. We can now compose the implementations of the caller from Example 9 and the callee from Example 10 to obtain the following closed program. We arbitrarily choose to place the callee in memory before the caller.

$$p_{\text{main}}(\mathit{entry}) \triangleq \mathsf{LOCAL}\ f;\ f{:}\ p_{\text{callee}};\ \mathit{entry}{:}\ p_{\text{caller}}(f)$$

We can give the following specification to this program, which says that the code between entry and j will increment EAX by 4 and step on EDX and the flags.

$$\begin{array}{l}\vdash (\mathsf{safe} \otimes (\mathsf{EIP} \mapsto j * \mathsf{EAX} \mapsto a + 4) \Rightarrow \\ \quad \mathsf{safe} \otimes (\mathsf{EIP} \mapsto \mathit{entry} * \mathsf{EAX} \mapsto a) \\ ) \otimes (\mathsf{EDX?} * \mathsf{flags}) \oslash (i..j \mapsto p_{\text{main}}(\mathit{entry}))\end{array}$$

The crucial step in proving this specification is to satisfy the caller's assumption, S_{callee}, with the callee specification, which is essentially $S_{\text{callee}} \otimes \mathsf{flags}$. The former entails the latter, but here we would need the entailment to go the other way. Instead, we exploit that FRAME is a higher-order frame rule [9] and lets us frame an assertion on to the left and right side of an entailment simultaneously. This is allowed by the rules $\otimes$-$\vdash$ and $\oslash$-$\vdash$ from Sections 4.3 and 4.5. Abbreviating

$$S_{\text{caller}}(f) \triangleq \forall a.\ \{\mathsf{EAX} \mapsto a\}\ p_{\text{caller}}(f)\ \{\mathsf{EAX} \mapsto a + 4\} \otimes \mathsf{EDX?},$$

we can derive

$$\dfrac{\dfrac{\dfrac{\dfrac{}{S_{\text{callee}}(f) \vdash S_{\text{caller}}(f)}\ \text{Ex. 9}}{S_{\text{callee}}(f) \otimes \mathsf{flags} \vdash S_{\text{caller}}(f) \otimes \mathsf{flags}}\ \otimes\text{-}\vdash}{S_{\text{callee}}(f) \otimes \mathsf{flags} \oslash f \mapsto p_{\text{callee}} \vdash S_{\text{caller}}(f) \otimes \mathsf{flags} \oslash f \mapsto p_{\text{callee}}}\ \oslash\text{-}\vdash$$

We know from Example 10 that $\vdash S_{\text{callee}}(f) \otimes \mathsf{flags} \oslash f \mapsto p_{\text{callee}}$, so by transitivity of $\vdash$ we conclude $\vdash S_{\text{caller}}(f) \otimes \mathsf{flags} \oslash f \mapsto p_{\text{callee}}$. From this, it is straightforward to derive our desired specification for p_{main}. $\diamond$

The preceding example showed how to use FRAME as a *second-order* [9], or, *hypothetical* [32] frame rule. The procedure involved was first-order at run-time, though. The following example involves a proper higher-order procedure; i.e., a procedure that takes a pointer to another procedure as argument.

Example 12. The simplest example of a higher-order procedure is 'apply', which in a functional programming language would be defined as

$$\mathsf{apply}(g, x) = g(x).$$

In our set-up, an apply procedure that takes its g argument in register EBX is implemented simply as

$$p_{\text{apply}} \triangleq \mathsf{jmp\ EBX}.$$

Its specification reflects how it forwards the behaviour (P, Q) of g:

$$\begin{array}{l}\vdash \big(g{\mapsto}\{P * \mathsf{EBX?}\}\{Q\} \Rightarrow \\ \quad f{\mapsto}\{P * \mathsf{EBX} \mapsto g\}\{Q\}\big) \oslash f \mapsto p_{\text{apply}}.\end{array} \quad \diamond$$

6. Practical verification

We have used our Coq development not only to build a machine model and to validate the logic developed in this paper; it is also an environment for building and verifying actual machine-code programs.

In this section we describe the Coq tactic support that we have developed for making machine code verification manageable, and present a slightly larger example of assembly language (seven instructions!) in order to give a flavour of the Coq proof of its correctness.

6.1 Example: memory allocation

We illustrate the use of the logic, rules, and Coq tactics with a slightly more challenging example: the specification of a memory allocator and its simplest possible realisation, the bumping of a pointer and checking it against a limit.

Its specification is as follows, parameterized by the number of bytes n to be allocated and an address fail to jump to on failure.

$$\begin{array}{l}\mathsf{allocSpec}(n, \mathit{fail}, \mathit{inv}, \mathit{code}) \triangleq \forall i, j. \\ \quad (\mathsf{safe} \otimes (\mathsf{EIP} \mapsto \mathit{fail} * \mathsf{EDI?}) \wedge \\ \quad\ \mathsf{safe} \otimes (\mathsf{EIP} \mapsto j * \exists a.\ (\mathsf{EDI} \mapsto a{+}n) * (a\ ..\ a{+}n \mapsto \_)) \Rightarrow \\ \quad\ \mathsf{safe} \otimes (\mathsf{EIP} \mapsto i * \mathsf{EDI?})) \\ \quad \otimes (\mathsf{ESI?} * \mathsf{flags} * \mathit{inv}) \oslash (i..j \mapsto \mathit{code})\end{array}$$

The specification is framed by an assertion that register ESI is used as scratch storage, flags are updated arbitrarily, and an internal invariant inv is maintained. The latter might be the well-formedness of some representation of free lists, or in our trivial allocator, simply a pair of pointers.

The calling convention is 'inline', in other words, the allocator is just a macro consisting of assembly in code. In Section 6.5, we will wrap a slightly less trivial calling convention around it.

Control either drops through, if successful, or branches to address fail, if memory cannot be allocated. On success, the allocator leaves an address a in EDI that is just beyond the n bytes of memory that were allocated; on failure, EDI is trashed.

Perhaps surprisingly, even a bump-and-check implementation consists of seven instructions:

$$\begin{array}{rl}\mathsf{allocImp}(\mathit{info}, n, \mathit{fail}) \triangleq & \mathsf{mov\ ESI}, \mathit{info}; \\ & \mathsf{mov\ EDI}, [\mathsf{ESI}]; \\ & \mathsf{add\ EDI}, n; \\ & \mathsf{jc}\ \mathit{fail}; \\ & \mathsf{cmp}\ [\mathsf{ESI} + 4],\ \mathsf{EDI}; \\ & \mathsf{jc}\ \mathit{fail}; \\ & \mathsf{mov}\ [\mathsf{ESI}],\ \mathsf{EDI}.\end{array}$$

The implementation invariant inv is the following:

$$\begin{array}{l}\mathsf{inv}(\mathit{info}) \triangleq \exists \mathit{base}, \mathit{limit}. \\ \quad \mathit{info} \mapsto \mathit{base} * (\mathit{info} + 4 \mapsto \mathit{limit}) * (\mathit{base}\ ..\ \mathit{limit} \mapsto \_).\end{array}$$

In other words, at address info there is a pair of pointers base and limit that bound a piece of mapped memory.

6.2 Applying instruction rules

During a proof, we typically keep the goal in the form

$$S_{\text{ctx}} \vdash (S \Rightarrow \mathsf{safe} \otimes P) \oslash R.$$

The specifications discussed in this paper are easy to put into that form by applying distributivity rules for $\otimes$ and decurrying nested implications, and we have implemented a tactic to do this automatically. Typically, R describes the code to be executed, and P describes the instruction pointer and the remaining state that will go into proving the precondition of the next instruction.

We may use the full range of specification-logic rules on this goal, but eventually we will want to apply the lemma appropriate

for the code that EIP is pointing to in P. We assume that the lemma has the same form as the goal and apply a lemma through the following rule.

$$\frac{\begin{array}{l} S'_{\text{ctx}} \vdash (S' \Rightarrow \mathsf{safe} \otimes P') \oslash R' \\ S_{\text{ctx}} \vdash S'_{\text{ctx}} \\ P \quad \vdash P' * R_{\text{P}} \\ R \quad \vdash R' * \top \\ S_{\text{ctx}} \vdash (S \Rightarrow S' \otimes R_{\text{P}}) \oslash R \end{array}}{S_{\text{ctx}} \vdash (S \Rightarrow \mathsf{safe} \otimes P) \oslash R} \textsc{SpecApply}$$

The top premise is the lemma to be applied, and the bottom premise is the remaining proof obligation that describes the symbolic state after having applied the lemma. If the lemma is an instruction rule, the three middle premises correspond to satisfying its preconditions at the level of specifications, data memory and code memory respectively. The latter two can be dealt with by our entailment checker, described in the next subsection.

6.3 Assertion entailment solving

Much of the activity in a formal separation logic proof is proving entailment between assertions. This happens every time a precondition needs to be discharged, and if it is not automated, the proofs will drown in the details of fragile manual context manipulation and rewriting modulo associativity and commutativity.

Typically, we are given a description of the current state P and a precondition P', and we must show $P \vdash P' * R$ for our own choice of frame R, which represents all the left-over state that was not consumed by the precondition and can therefore be framed out. Our approach to this automation is similar to other separation-logic tools [4, 16]: if P and P' consist only of $*$, emp and atomic assertions, we iterate through the conjuncts of P', attempting to unify each with a conjunct found in P and let the two cancel out.

Typically, P' is full of holes corresponding to universally-quantified variables that have yet to be instantiated. The holes are represented in Coq as *unification variables*, which are identifiers that will receive a value upon being unified with a subformula from P. Several subformulas of P may unify, but typically only one choice will permit the entailment as a whole to be solved. For example, we may be proving

$$\mathsf{EAX} \mapsto i * j \mapsto 2 * i \mapsto 1 \vdash \mathsf{EAX} \mapsto U_1 * U_1 \mapsto U_2 * \top,$$

where U_1 and U_2 are unification variables of type DWORD. If our algorithm should attempt to unify the atom $U_1 \mapsto U_2$ with the atom $j \mapsto 2$, it will succeed, but the remaining proof obligation will be

$$\mathsf{EAX} \mapsto i * i \mapsto 1 \vdash \mathsf{EAX} \mapsto j * \top$$

The algorithm succeeds even if it did not solve the goal entirely, leaving the rest to be proved interactively, but in this case there is no solution for the remaining part of the goal.

Rather than try to support backtracking, which does not combine well with interactive proof, we make the algorithm greedy but predictable: subformulas of P' are unified from left to right. In our current example, this would first fix the choice of U_1 to be i, and the second conjunct of P' would therefore become $i \mapsto U_2$, which rules out the bad unification choice from before.

There is of course no guarantee that this always works, but we have found that it virtually always works in practice as long as preconditions are written with this left-to-right order in mind. This happens naturally since it is also more readable for humans who read from left to right. If the algorithm should still fail, it remains possible to manually instantiate the unification variables.

The entailment solving algorithm is implemented with a hybrid approach, where the unification is done by Coq's built-in higher-order unification engine, while the cancellation of identical terms is done with *proof by reflection* [17], which has good performance.

If an entailment has existential quantifiers on the left-hand side, we can apply the rule

$$\frac{\forall x.\ (\mathcal{C}[P(x)] \vdash Q)}{\mathcal{C}[\exists x.\ P(x)] \vdash Q}$$

where $\mathcal{C}$ is formula with a hole that contains only $*$-connectives in the path from the root to the hole. This lets us effectively move the quantified variable into the Coq variable context.

If an entailment has existential quantifiers on the right-hand side, we will eventually need to instantiate them with witnesses. This can be done with the rule

$$\frac{\exists x.\ (P \vdash \mathcal{C}[Q(x)])}{P \vdash \mathcal{C}[\exists x.\ Q(x)]}$$

We immediately apply the rule, but we instantiate x with a *unification variable*, which in practice defers instantiation until a unification forces it to happen as described above.

We have extended the tactic for moving quantifiers into the context so it also works on specification-logic entailments. For example, given the goal

$$S' \vdash \forall x_1.\ S \Rightarrow \mathsf{safe} \otimes (\exists x_3.\ P(x_1, x_3)) \oslash (\exists x_2.\ R(x_2)),$$

the extended tactic will introduce x_1, x_2 and x_3 into the Coq variable context and leave the new goal

$$S' \vdash S \Rightarrow \mathsf{safe} \otimes P(x_1, x_3) \oslash R(x_2).$$

The rules allowing x_2 and x_3 to be pulled out are given in Sections 4.5 and 7.2 respectively.

6.4 Proving the allocator

Correctness consists of proving the following, for any $info$, n, $fail$.

$$\vdash \mathsf{allocSpec}(n, fail, \mathsf{inv}(info), \mathsf{allocImp}(info, n, fail)).$$

Here is a fragment of the Coq proof script that deals with the second instruction in the implementation. (We make use of `ssreflect` extensions to standard Coq tactic notation [20].)

```
(* mov EDI, [ESI] *)
rewrite {2}/inv. specintros => base limit.
specapply MOV_RM0_rule.
- by ssimpl.
```

For this instruction, almost everything is handled automatically. The initial `rewrite` simply unfolds the invariant inv to expose the existential quantifiers. The custom tactic `specintros` pulls the existentially-quantified variables from deep within the goal to introduce them into the Coq context. The tactic '`specapply` l' will first normalise both the goal and lemma l to have the form required by the SPECAPPLY rule from Section 6.2. It will then invoke SPECAPPLY with l as the first premise. In this case, l is `MOV_RM0_rule`, the rule for instructions of the form mov $r_1, [r_2]$. The second and fourth premises are trivial, leaving only the precondition of the mov rule as a subgoal. This can be discharged by our `ssimpl` tactic, which implements entailment checking as described in Section 6.3.

In fact none of the instructions needs more than four lines of proof, and we hope to reduce this further through the use of additional lemma and tactic support once we have more experience with proving.

6.5 Wrapping the allocator

Having verified a component, such as the allocator, it is reasonably straightforward to use to the logic to verify higher-level abstractions in a modular way. As an example, we show the wrapping of the allocator in a procedure for consing onto a list.

We start with the inductive 'list segment' assertion of separation logic (originally due to Burstall [13]):

$$\mathsf{listSeg}(p, e, vs) \triangleq \begin{cases} \exists q.\ (p \mapsto v) * (p{+}4 \mapsto q) * \mathsf{listSeg}(q, e, vs') & \text{if } vs = v :: vs' \\ p = e \wedge \mathsf{emp} & \text{otherwise} \end{cases}$$

Here, vs is a list of DWORDs and the assertion says that memory contains a linked list starting at p and ending at e with elements given by vs. A possible specification for our cons function is

$$\begin{aligned} &\mathsf{consSpec}(r_1, r_2, \mathit{info}, i, j, \mathit{code}) \triangleq \forall h, t, e, vs.\\ &(i \mapsto \{r_1 \mapsto h * r_2 \mapsto t * \mathsf{listSeg}(t, e, vs) * \mathsf{EDI}?\}\\ &\qquad \{r_1? * r_2? * ((\mathsf{EDI} \mapsto 0 * \mathsf{listSeg}(t, e, vs)) \vee\\ &\qquad\qquad (\exists q.\ \mathsf{EDI} \mapsto q * \mathsf{listSeg}(q, e, h :: vs))\}\\ &) \otimes (\mathsf{ESI}? * \mathsf{flags} * \mathsf{inv}(\mathit{info})) \oslash (i..j \mapsto \mathit{code}) \end{aligned}$$

specifying a procedure that is passed a value h in r_1 and a pointer to a list starting at t in r_2. On return, EDI is either zero, and the original linked list is preserved, or EDI points to a linked list segment ending at e with h added as the new head element. An implementation is given by

$$\begin{aligned} \mathsf{cons}(r_1, r_2, \mathit{info}) \triangleq\ & \mathsf{LOCAL}\ \mathit{fail};\ \mathsf{LOCAL}\ \mathit{succeed};\\ & \mathsf{allocImp}(\mathit{info}, 8, \mathit{fail});\\ & \mathsf{sub}\ \mathsf{EDI}, 8;\\ & \mathsf{mov}\ [\mathsf{EDI}], r_1;\\ & \mathsf{mov}\ [\mathsf{EDI}+4], r_2;\\ & \mathsf{jmp}\ \mathit{succeed};\\ & \mathit{fail}:\\ & \quad \mathsf{mov}\ \mathsf{EDI}, 0;\\ & \mathit{succeed}:\\ & \quad \mathsf{jmp}\ \mathsf{EDX} \end{aligned}$$

The proof that for any $r_1, r_2, \mathit{info}, i$ and j,

$$\vdash \mathsf{consSpec}(r_1, r_2, \mathit{info}, i, j, \mathsf{cons}(r_1, r_2, \mathit{info}))$$

is entirely modular, relying on the BODY rule and the previous result that allocImp meets allocSpec.

7. Properties of the frame connectives

We now return to the frame connective, $\otimes$, defined in Section 4.3. In previous literature on higher-order frame rules [8–10, 33], the R in $S \otimes R$ tends to be inert and does not interact with its environment until it has distributed inwards across all connectives and has been merged into the pre- and post-conditions of a triple. Only at that point will the rule of consequence and the existential rule for triples be used to interact with R.

Since we only see triples in certain special cases, as described in Section 5.1, we are interested in specification-level generalisations of the consequence and existential rules, just as FRAME is a specification-level generalisation of the frame rule for triples. The use of these generalised rules in practice is similar to their counterparts in ordinary Hoare logic.

7.1 Specification-level rule of consequence

The standard Hoare rule of consequence states that $\{P\}\ c\ \{Q\}$ is contravariant in P and covariant in Q with respect to entailment. Analogously, the generalisation we present here describes the variance of $S \otimes R$ in R. It turns out that $S \otimes R$ is not always covariant nor always contravariant in R; it can be either, depending on S. We encode this as two predicates on S:

$$\mathsf{frame}_+(S) \triangleq \forall P, Q.\ (P \vdash Q) \Rightarrow (S \otimes P \vdash S \otimes Q)$$
$$\mathsf{frame}_-(S) \triangleq \forall P, Q.\ (P \vdash Q) \Rightarrow (S \otimes Q \vdash S \otimes P)$$

These definitions directly give rise to our two specification-level rules of consequence:

$$\frac{\mathsf{frame}_+(S) \quad P \vdash Q}{S \otimes P \vdash S \otimes Q} \qquad \frac{\mathsf{frame}_-(S) \quad P \vdash Q}{S \otimes Q \vdash S \otimes P}$$

All we did so far was to switch the problem to proving $\mathsf{frame}_+(S)$ or $\mathsf{frame}_-(S)$ for particular S, but it turns out that there is a very schematic set of rules for this. Writing $f : (V_1, \ldots, V_n) \longrightarrow V$ to mean

$$\forall S_1, \ldots, S_n.\ \mathsf{frame}_{V_1}(S_1) \wedge \ldots \wedge \mathsf{frame}_{V_n}(S_n) \Rightarrow \mathsf{frame}_V(f(S_1, \ldots, S_n)),$$

we can tabulate the rules for various connectives concisely:

$$\begin{aligned} \mathsf{safe} &: - \\ \top, \bot &: + \text{ and } - \\ \triangleright, \otimes, \oslash, \forall, \exists &: + \longrightarrow + \text{ and } - \longrightarrow - \\ \wedge, \vee &: (+, +) \longrightarrow + \text{ and } (-, -) \longrightarrow - \\ \Rightarrow &: (-, +) \longrightarrow + \text{ and } (+, -) \longrightarrow - \end{aligned}$$

Notice that all the logical connectives preserve either covariance or contravariance of their operands (modulo the flip that happens for implication), but there is no way to combine the variances.

Example 13. For all P_1 and P_2, $\mathsf{frame}_-(\triangleright\mathsf{safe} \otimes P_1 \wedge \mathsf{safe} \otimes P_2)$.$\diamond$

Example 14. For all P, $\mathsf{frame}_+(\mathsf{safe} \otimes P \Rightarrow \bot)$. $\diamond$

There are no definitions analogous to $\mathsf{frame}_+(S)$ and $\mathsf{frame}_-(S)$ for the read-only frame connective since $S \oslash R$ is always contravariant in R. But as we will see in Section 7.3, the frame family of predicates plays an important role for $\oslash$ too.

It is no coincidence that these rules for variance have not been studied in the previous literature. There is no rule for frame_+ or frame_- on Hoare triples, and in a logic where the only atomic specifications are the triple and $\top, \bot$, then any S that satisfies $\mathsf{frame}_+(S)$ or $\mathsf{frame}_-(S)$ is equivalent to either $\top$ or $\bot$.

7.2 Specification-level existential rule

The *existential rule* in Hoare logic allows moving an existential quantifier from the precondition of a Hoare triple out into the logical variable context. Just as we have generalised the frame and consequence rules, we can generalise the existential rule to work with other specifications than triples. Using the same approach as in Section 7.1, we define

$$\mathsf{frame}_\exists(S) \triangleq \forall P.\ \big((\forall x.\ S \otimes P(x)) \vdash S \otimes (\exists x.\ P(x))\big)$$

We can then state the specification-level existential rule as

$$\frac{\mathsf{frame}_\exists(S) \quad S' \vdash \forall x.\ S \otimes P(x)}{S' \vdash S \otimes (\exists x.\ P(x))}$$

The following rules, using the notation introduced in Section 7.1, let us schematically prove $\mathsf{frame}_\exists$.

$$\begin{aligned} \mathsf{safe} &: \exists \\ \triangleright, \otimes, \oslash, \forall &: \exists \longrightarrow \exists \\ \wedge &: (\exists, \exists) \longrightarrow \exists \\ \Rightarrow &: (-, \exists) \longrightarrow \exists \end{aligned}$$

The natural converse of $\mathsf{frame}_\exists(S)$, with the entailment in the other direction, is equivalent to $\mathsf{frame}_-(S)$. This gives an intuitive justification of the rule for implication above.

Example 15. For all P, c, Q, we have $\mathsf{frame}_\exists(\{P\}\ c\ \{Q\})$. This is seen by unfolding the definition of the triple and applying the above rules. $\diamond$

7.3 Further properties of read-only frame connective

The read-only frame and the frame connectives are interchangeable in certain cases:

1. For singleton frames: $S \oslash \{\sigma\} \equiv S \otimes \{\sigma\}$.
2. If $\mathsf{frame}_{\exists}(S)$ then $S \oslash R \vdash S \otimes R$.
3. If $\mathsf{frame}_{\_}(S)$ then $S \otimes R \vdash S \oslash R$.

Whereas two adjacent frame connectives can always be merged and split by the $\otimes$-$*$ rule, this is not always possible for the read-only frame connective:

1. If $\mathsf{frame}_{\exists}(S)$ then $S \oslash (R * R') \vdash S \oslash R \oslash R'$.
2. Unconditionally, $S \oslash R \oslash R' \vdash S \oslash (R * R')$.

8. Related work

This paper builds on previous work on higher-order frame rules, assembly-language verification and guarded recursion.

Separation logic for assembly code. Our work shares many goals with the work of Myreen et al. [28, 29]. They have built a separation logic for subsets of ARM and x86 in the HOL4 proof assistant. Their logic emphasises total correctness, but since assembly-language programs do not terminate, total correctness does not mean guaranteed termination as it usually would. Instead, a post-condition Q means that execution will eventually reach a machine state in Q. This makes specifications much more intensional than in our case, preventing, for example, relocating and patching (or interpreting) code in memory in an externally-unobservable way unless this has been somehow explicitly allowed by the specification.

The logic of Myreen et al. lacks labels in assembly programs, relying instead on explicit instruction address arithmetic. Their entire specification logic takes place in a generalised Hoare triple with multiple pre- and postconditions and offset transformer functions. This is general enough to support jumps, function calls and self-modifying code, but it remains a triple and is thus restricted to what can be expressed with preconditions, postconditions and code blocks.

The CAP family of logics from Shao et al. are also Hoare logics for low-level code, all verified in Coq. The family includes XCAP [31], GCAP [14], SCAP and ISCAP [40]. Unfortunately, neither of them is a generalisation of any of the others, so each has its strengths and weaknesses. All except GCAP and SCAP have high-level heap manipulation commands such as allocation or function calls built into the machine semantics.

All except GCAP have the program residing in a map from labels to instruction sequences, which is a high level of abstraction and cannot support treating code as data. As Myreen [28] and hopefully this paper have shown, it is not difficult to treat code as data and support function pointers if the logic is fundamentally set up for it. In contrast, GCAP supports it with some awkwardness by attempting to impose the map-from-labels abstraction on top of what is actually happening in the machine.

Affeldt et al [1] have formalized in Coq a separation logic for first-order MIPS assembly code, extending a simpler logic due to Saabas and Uustalu [36], and applied it to verifying provable security of implementations of cryptographic primitives.

Chlipala's Bedrock project [16] is also a Coq framework for verifying low-level code with separation logic. Like our framework, Bedrock has 'while' and 'if' macros and associated proof principles for common patterns of structured code. Chlipala emphasises automated verification, and the program logic is therefore not very expressive. There is no frame rule, so frames are instead passed around explicitly in procedure specifications. Chlipala explains the problem with defining a frame rule for programs with unstructured jumps; here we have demonstrated how this may be solved.

None of the logics discussed above feature a higher-order frame rule.

Higher-order frame rules. The frame rule was extended by O'Hearn et al. [32] to the *hypothetical frame rule*, which allowed framing invariants onto a context of procedure specifications in addition to the triple under consideration. This allowed greater modularity in separate verification of caller and callee, but it still required programs to have structured control flow.

The higher-order frame rule was proposed by Birkedal et al. and used in a separation-logic type system for a programming language with higher-order functions and ground store [8–10]. It has later been extended to languages with higher-order store and used by Krishnaswami [23] and by Pottier [33]. In all cases, it has been for high-level functional programming languages, whereas we have applied it to machine code. We believe we are the first to complement the higher-order frame rule with a higher-order rule of consequence and a higher-order existential rule (Sections 7.1 and 7.2).

Typed assembly language. The work of Appel et al. on typed assembly language and foundational proof-carrying code has demonstrated that step indexing [2] is a viable technique for describing the behaviour of low-level programs. The 'Very Modal Model' paper [3] popularised Nakano's *later* operator, which we also use here, and demonstrated its applicability to assembly code.

The work on typed assembly language focuses on safety of reference types coming from high-level languages and does not attempt to verify code for full functional correctness as we do.

9. Future work

The logic described in this paper will form the foundation for broader research on language-based security in verified systems software.

Although the focus of this paper is on the general design of a separation logic for machine code, and is thus largely parametric in the underlying machine model, one's confidence in the real world validity of verifications in the logic is undeniably limited by one's confidence in the accuracy of that model. Our x86 model was hand-constructed from reading the Intel manuals and, although small programs extracted from Coq seem to run as expected on real hardware, has not been subject to any systematic testing or verification. Indeed, there is one aspect of the Intel specification that we knowingly do not currently model, namely the presence of a code cache: instructions written to memory are, on post-Pentium processors, not guaranteed to be picked up by subsequent execution until a jump or other synchronizing instruction has occurred. We plan to treat the code cache following the approach of Myreen [28], which we expect to be unproblematic. More generally, however, we would like to work with a more trustworthy machine model; these have previously been obtained by extensive testing [19, 26] and semi-automated extraction from the text of reference manuals [11].

An important feature currently missing from our machine model is I/O. By adding this we would incorporate *observations* beyond the simple notion of 'safe' execution, but we believe that our framework is generic enough that the safe specification can be generalized to safety properties involving observable input and output transitions. We have not so far given any serious thought to how one might also prove liveness properties in a comparably extensional way, though that is clearly an interesting subject for future work. It would also be useful to extend our logic to deal with binary relations, rather than unary predicates, on machine states. Such an extension would allow us to verify information flow and abstraction properties [7].

We have already begun to experiment with verified compilation, building a tiny imperative language, its compiler, program logic and proof of correctness, all within Coq. It is straightforward to mix machine code with higher-level languages, as our logic provides a common framework for specifying their interaction at a suitably high level. We plan to develop a number of domain-specific 'little languages' within the same framework.

Low-level code often makes sophisticated use of low-level data structures whose 'ownership' properties cannot easily be captured by the default model of separation described here. We might instead employ 'fictional separation logic' [22]; it is interesting to note that even our use of partial states Σ to describe the machine state $\mathbb{S}$ in a more fine-grained way is reminiscent of fictional separation.

Acknowledgements. We would like to thank Lars Birkedal and Kasper Svendsen for many discussions on higher-order frame rules and their applications.

References

[1] R. Affeldt, D. Nowak, and K. Yamada. Certifying assembly with formal security proofs: the case of BBS. *Sci. Comput. Prog.*, 77(10-11), 2012.

[2] A. W. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 2001.

[3] A. W. Appel, P.-A. Melliès, C. D. Richards, and J. Vouillon. A very modal model of a modern, major, general type system. In *Proceedings of POPL*, 2007.

[4] J. Bengtson, J. B. Jensen, and L. Birkedal. Charge! – a framework for higher-order separation logic in Coq. In *Proc. of ITP*, 2012.

[5] N. Benton. A typed, compositional logic for a stack-based abstract machine. In *APLAS*, volume 3780 of *LNCS*, 2005.

[6] N. Benton. Abstracting allocation: The new new thing. In *Computer Science Logic (CSL 2006)*, volume 4207 of *LNCS*, 2006.

[7] N. Benton and N. Tabareau. Compiling functional types to relational specifications for low level imperative code. In *TLDI*, 2009.

[8] L. Birkedal, N. Torp-Smith, and H. Yang. Semantics of separation-logic typing and higher-order frame rules. In *Proc. of LICS*, 2005.

[9] L. Birkedal, N. Torp-Smith, and H. Yang. Semantics of separation-logic typing and higher-order frame rules for Algol-like languages. *Logical Methods in Computer Science*, 2006.

[10] L. Birkedal and H. Yang. Relational parametricity and separation logic. *Logical Methods in Computer Science*, 2008.

[11] F. Blanqui, C. Helmstetter, V. Joloboff, J.-F. Monin, and X. Shi. Designing a CPU model: from a pseudo-formal document to fast code. In *3rd Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO 2011)*, 2011.

[12] R. Bornat, C. Calcagno, P. W. O'Hearn, and M. J. Parkinson. Permission accounting in separation logic. In *Proceedings of POPL*, 2005.

[13] R. M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7, 1972.

[14] H. Cai, Z. Shao, and A. Vaynberg. Certified self-modifying code. In *Proc. of PLDI*, 2007.

[15] C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *Proc. of LICS*, 2007.

[16] A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *Proc. of PLDI*, 2011.

[17] A. Chlipala. *Certified Programming with Dependent Types*. MIT Press, to appear.

[18] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proc. of Symposia in Applied Mathematics*, Providence, Rhode Island, 1967. AMS.

[19] A. C. J. Fox and M. O. Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *1st International Conference on Interactive Theorem Proving (ITP 2010)*, volume 6172 of *LNCS*, 2010.

[20] G. Gonthier, A. Mahboubi, and E. Tassi. A small scale reflection extension for the Coq system. Technical Report 6455, INRIA, 2011.

[21] C. Hawblitzel and E. Petrank. Automated verification of practical garbage collectors. In *POPL*, 2009.

[22] J. B. Jensen and L. Birkedal. Fictional separation logic. In *Proc. of ESOP*, volume 7211 of *LNCS*. Springer, 2012.

[23] N. R. Krishnaswami. *Verifying Higher-Order Imperative Programs with Higher-Order Separation Logic*. PhD thesis, Carnegie Mellon University, 2012.

[24] J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In *Mathematical Aspects of Computer Science*, volume 19 of *Proc. of Symposia in Applied Mathematics*. AMS, 1967.

[25] J. Strother Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5, 1989.

[26] G. Morrisett, G. Tan, J. Tassarotti, J.B. Tristan, and E. Gan. Rocksalt: Better, faster, stronger SFI for the x86. In *33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2012)*. ACM, 2012.

[27] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3), 1999.

[28] M. O. Myreen. Verified just-in-time compiler on x86. In *Proc. of POPL*, 2010.

[29] M. O. Myreen and M. J. C. Gordon. Hoare logic for realistically modelled machine code. In *Proc. of TACAS*, 2007.

[30] H. Nakano. A modality for recursion. In *Proc. of LICS*, 2000.

[31] Z. Ni and Z. Shao. Certified assembly programming with embedded code pointers. In *Proc. of POPL*, 2006.

[32] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *Proc. of POPL*, 2004.

[33] F. Pottier. Hiding local state in direct style: a higher-order anti-frame rule. In *Proc. of LICS*, 2008.

[34] J. C. Reynolds. An introduction to specification logic. In *Logics of Programs*, 1983.

[35] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. of LICS*, 2002.

[36] A. Saabas and T. Uustalu. A compositional natural semantics and Hoare logic for low-level languages. *Theor. Comput. Sci.*, 373(3), 2007.

[37] M. Sozeau and N. Oury. First-class type classes. In *Proc. of TPHOLs*, 2008.

[38] G. Tan and A. W. Appel. A compositional logic for control flow. In *7th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2006)*, 2006.

[39] A. M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, 1949.

[40] X. Jiang W. Wang, Z. Shao and Y. Guo. A simple model for certifying assembly programs with first-class function pointers. In *Proc. of TASE*, 2011.

[41] W. D. Young. A mechanically verified code generator. *Journal of Automated Reasoning*, 5, 1989.