Correctness of Source-Level Safety Policies

Ewen Denney* and Bernd Fischer†

*QSS / †RIACS
NASA Ames Research Center, Moffett Field, CA 94035, USA
{edenney,fisch}@email.arc.nasa.gov

Abstract. Program certification techniques formally show that programs satisfy certain safety policies. They rely on the correctness of the safety policy which has to be established externally. In this paper we investigate an approach to show the correctness of safety policies which are formulated as a set of Hoare-style inference rules on the source code level. We develop a framework which is generic with respect to safety policies and which allows us to establish that proving the safety of a program statically guarantees dynamic safety, i.e., that the program never violates the safety property during its execution. We demonstrate our framework by proving safety policies for memory access safety and memory read/write limitations to be sound and complete. Finally, we formulate a set of generic safety inference rules which serve as the blueprint for the implementation of a verification condition generator which can be parameterized with different safety policies, and identify conditions on appropriate safety policies.
Keywords. Program verification, Hoare logic, program safety, code certification, proof-carrying code

1 Introduction

Program certification techniques like proof-carrying code (PCC) [12] use formal reasoning techniques to show that programs satisfy certain *safety policies* as for example memory safety (i.e., that they do not access out-of-bounds memory), rather than full functional correctness.

In effect, these techniques shift the trust burden from the original program to the certification system: instead of having to trust an arbitrary program to be safe, users have to trust the certifier to be correct. However, this still requires a lot of trust since a certifier is itself a complex system involving many different components and steps. In the original PCC approach [12], a compiler first translates an untrusted source program into an annotated machine program, to which a verification condition generator (VCG) then applies a safety policy, formulated as a set of Hoare rules. This produces a set of proof obligations, which are processed by a theorem prover; the resulting proofs are finally scrutinized by a proof checker (cf. Figure 1). Fortunately, not all of these components have to be trusted—here, trust is required only in the safety policy, the VCG, and the proof checker but not in the much larger prover or the compiler itself.

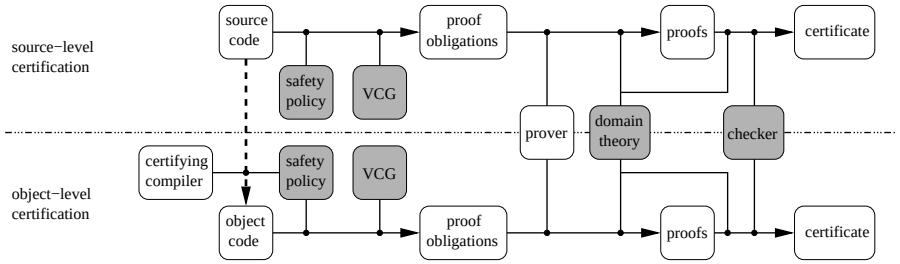

Fig. 1. System architecture for source-level certification and object-level certification; trusted components are shaded.

However, the fact that the safety policy still has to be trusted turns out to be the Achilles heel of the approach, both for theoretical and practical reasons. On the theoretical side, if the rules are unsound or do not exactly formalize the intuitive notion of safety, "all bets are off" [10], i.e., even a safety proof does not guarantee that the program is actually safe. On the practical side, since a safety policy can consist of a collection of fairly complex Hoare rules, it is as liable to error as any other component of the certifier. Moreover, the VCG and the proof checker can be reused essentially unchanged for different safety policies and can thus be hardened over time, but the Hoare rules change with each new safety policy. Recent work has thus concentrated on ways to guarantee the correctness of safety policies, using approaches like type-preserving compilation [10], foundational PCC [2, 6], and reduction to core safety policies [13].

However, all these approaches work on the object code level, and cannot directly be extended to safety policies which are formulated on the source code level. Here, we investigate an approach to showing the correctness of source-level safety policies. Our goal is to develop a generic framework (cf. Figure 2) which lets us establish that proving the safety of a source program using a safety policy formulated as a set of Hoare rules guarantees safe execution, i.e., the program never violates the safety property.[1] In other words, we want to establish that the static and dynamic notions of safety coincide. We thus explicitly distinguish between the static safety *policies* (which are logical characterizations) and the dynamic safety *properties* (which are operational characterizations). We also explicitly separate the formalization of the safety properties from the operational semantics: a program can be unsafe even if its execution does not raise an exception. We restrict ourselves to the analysis of individual programs only, and do not consider system-level safety aspects. However, our techniques could be used with system-level analysis techniques, e.g., to decide whether a system is still safe even if an exception can occur in one of the system components.

Our interest in source-level policies has a number of reasons. (i) Programmers make errors on the source code level, so showing safety on the source code level seems not only to be more natural, it also makes it easier to pinpoint

[1] Provided that the compiler preserves the property, of course. See below for a more detailed discussion of this.

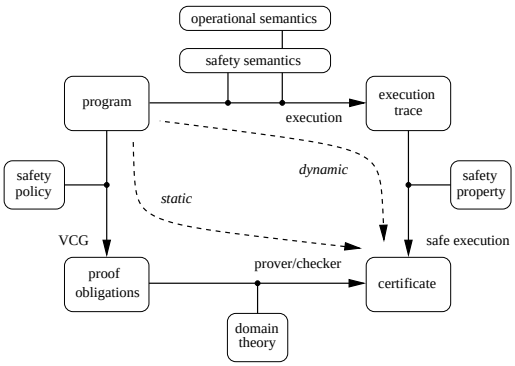

Fig. 2. Establishing correctness of source-level certification—general framework.

the errors. (*ii*) Some safety policies can be formulated more naturally (e.g., initialization-before-use) or only (e.g., loop variable restrictions) on the source code level.[2] In particular, high-level domain-specific policies such as *frame safety* [9] are inherently source level. (*iii*) Source-level certification is complementary to object-level approaches like PCC. In fact, to ensure that compilation does not compromise the demonstrated safety policy, source-level certification should be followed by object-level certification. However, explicit source-level certification provides a separation of concerns as different safety policies can be applied at different levels of abstraction. (*iv*) Formal software certification processes (e.g., DO-178B) usually also cover source code level, so certification support has to work on that level. (*v*) Finally, we are interested in the combination of certification and program synthesis [19], using certification (in a roundabout way) to increase confidence in our synthesis system, which generates source code and not object code.

The main contributions of this paper are as follows. First, we develop a general notion of safety property: we distinguish *stateless* properties, where safe execution can be defined in terms of the original operational semantics (i.e., execution traces), and *stateful* properties, where for book-keeping purposes the operational semantics must be augmented with a separate safety semantics. Second, we develop a *semantic* definition of safety, which lets us reason about the soundness and completeness of our safety policies. Third, we give a generic method of extending Hoare rules to incorporate an arbitrary safety property. In particular, our framework can serve as the basis for the implementation of a generic VCG.

In Section 2, we develop the basic theory of stateless safety properties, and then extend this in Section 3 to some examples of stateful safety. In Section 4 we present a general account of safety. Finally, Sections 5 and 6 discuss related work and draw some conclusions. Throughout this paper we assume a working familiarity with Hoare-style program correctness proofs (see [11], for example).

[2] This is related to the use of certification to enforce syntactic restrictions and coding standards.

2 Stateless Safety Properties

We introduce our framework using the deliberately simplistic language $\mathcal{L}_0$ of *while*-programs as shown in Figure 3; it uses the unspecified sets *Var* and *Const* of variables and literal constants.

$$
\begin{array}{lll}
\mathit{Cmd} ::= & \texttt{skip} & \mathit{Expr} ::= \mathit{Const} \\
\mid & \mathit{Var}\ \texttt{:=}\ \mathit{Expr} & \mid\ \mathit{Var} \\
\mid & \texttt{if}\ \mathit{Expr}\ \texttt{then}\ \mathit{Cmd} & \mid\ \mathit{Expr}\ \texttt{*}\ \mathit{Expr} \\
\mid & \texttt{while}\ \mathit{Expr}\ \texttt{do}\ \mathit{Cmd} & \mid\ \mathit{Expr}\ \texttt{/}\ \mathit{Expr} \\
\mid & \mathit{Cmd}\ \texttt{;}\ \mathit{Cmd} & \mid\ \mathit{Expr}\ \texttt{+}\ \mathit{Expr} \\
 & & \mid\ \mathit{Expr}\ \texttt{-}\ \mathit{Expr} \\
 & & \mid\ \mathit{Expr}\ \texttt{=}\ \mathit{Expr}
\end{array}
$$

Fig. 3. Syntax of *while*-language $\mathcal{L}_0$

Our initial safety property is *operator safety*, i.e., expression operators such as division are only applied to arguments within their respective domains. For $\mathcal{L}_0$, this boils down to the question of whether divisors are non-zero. However, even for this simple case we cannot naively define the safety of commands in terms of the safety of their subexpressions. Consider, for example, the commands

```
if false then x:=x+1/0      while true do skip; x:=x+1/0
```

which contain unsafe subexpressions but which we would nevertheless want to regard as safe (w.r.t. operator safety) because the division-by-zero exception will never be raised. Consider also the sequence `x:=y; w:=1/x` where safety of the subexpressions is not sufficient either because this does not incorporate the information that the division `1/x` is performed when `x` is bound to the value of `y`. Hence, we need an analysis of safety which takes into account the (operational) semantics of the programs.

2.1 Formulation of Safety Properties

A *safety property* is an operational characterization of the fact that "a program does not go wrong." We formalize safety properties as *judgements* of the form $\eta \vDash c\ \mathsf{safe}$, i.e., the command $c \in \mathit{Cmd}$ is safe under the environment $\eta \in \mathit{Env}$. As usual, we use environments $\eta : \mathit{Var} \rightharpoonup \mathit{Val}_\perp$ to record value bindings for the variables. Note that we use the bottom element $\perp$ only as an operational concept to denote and propagate the result of an undefined computation, but not to denote (un-) safety. In particular, a command can still be safe under an environment which contains a binding $x \mapsto \perp$; for example, `y:=x+1` is obviously still safe w.r.t. operator safety (i.e., division-by-zero free) simply because it does not contain any occurrence of the division operator. Conversely, unsafety does not necessarily manifest itself in a binding $x \mapsto \perp$.

$\eta \vDash c\ \mathsf{safe}_{\mathrm{op}}$
$\eta \vDash x\ \mathsf{safe}_{\mathrm{op}}$
$\eta \vDash e_1\ \mathrm{op}\ e_2\ \mathsf{safe}_{\mathrm{op}}$ iff $\eta \vDash e_1\ \mathsf{safe}_{\mathrm{op}}$ and $\eta \vDash e_2\ \mathsf{safe}_{\mathrm{op}}$ and $\mathrm{op} \in \{\mathtt{*}, \mathtt{+}, \mathtt{-}, \mathtt{=}\}$
$\eta \vDash e_1\ \mathtt{/}\ e_2\ \mathsf{safe}_{\mathrm{op}}$ iff $\eta \vDash e_1\ \mathsf{safe}_{\mathrm{op}}$ and $\eta \vDash e_2\ \mathsf{safe}_{\mathrm{op}}$ and $[\![e_2]\!]_\eta \neq 0$

Fig. 4. Operator safety for $\mathcal{L}_0$-expressions

We can then define the judgement $\mathsf{safe}_{\mathrm{op}}$ which formalizes operator safety for $\mathcal{L}_0$-expressions in the expected way, as shown in Figure 4. We use the notation $[\![e]\!]_\eta$ to denote evaluation of an expression $e \in \mathit{Expr}$ in an environment η.

Extending operator safety to commands requires an operational semantics for the commands; here, we assume the standard single-step operational semantics $\langle c, \eta\rangle \Rightarrow \langle c', \eta'\rangle$ for *while*-programs.[3][4] However, there are two different approaches to an extension. The first approach factors safety into two different judgements, $\mathsf{safestate}_{\mathrm{op}}$ and $\mathsf{safe}_{\mathrm{op}}$ (cf. Figure 5), where $\eta \vDash c\ \mathsf{safestate}_{\mathrm{op}}$ formalizes the intuition that the immediately next command is safe to execute (i.e., all of the expressions which it would evaluate immediately are safe) and the reduction relation restricts the application of $\mathsf{safestate}_{\mathrm{op}}$ to reachable commands and environments only. Hence, we have $\eta \vDash \mathtt{while\ true\ do\ skip;\ x:=1/0}\ \mathsf{safe}_{\mathrm{op}}$, as expected. This approach essentially mirrors the definition of what is called the safety policy in the syntactic FPCC-approach of Hamid et al. [6].

$\eta \vDash \mathtt{skip}\ \mathsf{safestate}_{\mathrm{op}}$
$\eta \vDash x\ \mathtt{:=}\ e\ \mathsf{safestate}_{\mathrm{op}}$ iff $\eta \vDash e\ \mathsf{safe}_{\mathrm{op}}$
$\eta \vDash \mathtt{if}\ b\ \mathtt{then}\ c\ \mathsf{safestate}_{\mathrm{op}}$ iff $\eta \vDash b\ \mathsf{safe}_{\mathrm{op}}$
$\eta \vDash \mathtt{while}\ b\ \mathtt{do}\ c\ \mathsf{safestate}_{\mathrm{op}}$ iff $\eta \vDash b\ \mathsf{safe}_{\mathrm{op}}$
$\eta \vDash c_1\,\mathtt{;}\,c_2\ \mathsf{safestate}_{\mathrm{op}}$ iff $\eta \vDash c_1\ \mathsf{safestate}_{\mathrm{op}}$
$\eta \vDash c\ \mathsf{safe}_{\mathrm{op}}$ iff $\forall \langle c, \eta\rangle \Rightarrow^* \langle c', \eta'\rangle \cdot \eta' \vDash c'\ \mathsf{safestate}_{\mathrm{op}}$

Fig. 5. Operator safety for $\mathcal{L}_0$-commands

The second approach directly integrates the formulation of the $\mathsf{safestate}_{\mathrm{op}}$-judgement into the operational semantics and has thus more of an abstract interpretation flavor (cf. Figure 6).

For this alternative definition $\widehat{\mathsf{safe}}_{\mathrm{op}}$ we first show by straightforward induction over commands that safety is preserved by reduction; in analogy to subject

[3] $\langle x := e, \eta\rangle \Rightarrow \langle \mathtt{skip}, \eta \oplus \{x \mapsto [\![e]\!]_\eta\}\rangle$,
$\langle \mathtt{skip}\ \mathtt{;}\ c_2, \eta\rangle \Rightarrow \langle c_2, \eta\rangle$,
$\langle c_1\ \mathtt{;}\ c_2, \eta\rangle \Rightarrow \langle c_1'\ \mathtt{;}\ c_2, \eta'\rangle$ if $\langle c_1, \eta\rangle \Rightarrow \langle c_1', \eta'\rangle$,
$\langle \mathtt{if}\ b\ \mathtt{then}\ c, \eta\rangle \Rightarrow \langle c, \eta\rangle$ if $[\![b]\!]_\eta = \mathit{true}$,
$\langle \mathtt{if}\ b\ \mathtt{then}\ c, \eta\rangle \Rightarrow \langle \mathtt{skip}, \eta\rangle$ if $[\![b]\!]_\eta = \mathit{false}$,
$\langle \mathtt{while}\ b\ \mathtt{do}\ c, \eta\rangle \Rightarrow \langle \mathtt{if}\ b\ \mathtt{then}\ (c; \mathtt{while}\ b\ \mathtt{do}\ c), \eta\rangle$

[4] We also use $\langle c, \eta\rangle \Downarrow \eta'$ to denote the result of a terminating evaluation of c, i.e., $\langle c, \eta\rangle \Downarrow \eta'$ iff $\langle c, \eta\rangle \Rightarrow^* \langle \mathtt{skip}, \eta'\rangle$.

$$
\begin{array}{ll}
\eta \vDash \texttt{skip}\ \widehat{\mathsf{safe}}_{\mathrm{op}} & \\
\eta \vDash x\ \texttt{:=}\ e\ \widehat{\mathsf{safe}}_{\mathrm{op}} & \text{iff } \eta \vDash e\ \mathsf{safe}_{\mathrm{op}} \\
\eta \vDash \texttt{if}\, b\, \texttt{then}\, c\ \widehat{\mathsf{safe}}_{\mathrm{op}} & \text{iff } \eta \vDash b\ \mathsf{safe}_{\mathrm{op}} \text{ and } [\![b]\!]_\eta = \mathit{true} \text{ implies} \\
 & \quad \eta \vDash c\ \widehat{\mathsf{safe}}_{\mathrm{op}} \\
\eta \vDash \texttt{while}\, b\, \texttt{do}\, c\ \widehat{\mathsf{safe}}_{\mathrm{op}} & \text{iff } \eta \vDash b\ \mathsf{safe}_{\mathrm{op}} \text{ and } [\![b]\!]_\eta = \mathit{true} \text{ implies } (\eta \vDash c\ \widehat{\mathsf{safe}}_{\mathrm{op}} \text{ and} \\
 & \quad \langle c, \eta\rangle \Downarrow \eta' \text{ implies } \eta' \vDash \texttt{while}\, b\, \texttt{do}\, c\ \widehat{\mathsf{safe}}_{\mathrm{op}}) \\
\eta \vDash c_1\,;c_2\ \widehat{\mathsf{safe}}_{\mathrm{op}} & \text{iff } \eta \vDash c_1\ \widehat{\mathsf{safe}}_{\mathrm{op}} \text{ and } \langle c_1, \eta\rangle \Downarrow \eta' \text{ implies } \eta' \vDash c_2\ \widehat{\mathsf{safe}}_{\mathrm{op}}.
\end{array}
$$

Fig. 6. Operator safety for $\mathcal{L}_0$-commands (structural definition)

reduction[5] we call this property *safety reduction*. Note that safety reduction holds trivially for $\mathsf{safe}_{\mathrm{op}}$ as defined in Figure 5.

Lemma 1. *(Safety Reduction)* $\eta \vDash c\ \widehat{\mathsf{safe}}_{op}$ *and* $\langle c, \eta\rangle \Rightarrow \langle c', \eta'\rangle$ *implies* $\eta' \vDash c'\ \widehat{\mathsf{safe}}_{op}$.

We can then show that both definitions are in fact equivalent. This is quite useful because the operational definition ($\widehat{\mathsf{safe}}$) is what we intuitively want but most proofs use the inductive definition ($\mathsf{safestate}$).

Lemma 2. *For all* η, c: $\eta \vDash c\ \widehat{\mathsf{safe}}_{op}$ *iff* $\eta \vDash c\ \mathsf{safe}_{op}$.

Proof: Use Lemma 1, and the fact that $\eta \vDash c\ \widehat{\mathsf{safe}}_{\mathrm{op}}$ implies $\eta \vDash c\ \mathsf{safestate}_{\mathrm{op}}$. ∎

Both Lemma 1 and Lemma 2 are independent of the particular safety judgement and hold as long as command safety is derived from expression safety in the way described in Figure 6.

In the following we discuss arbitrary safety properties, which can be *any* mathematical relation between environments and expressions. We reserve the use of *safety judgement* for the semantic clauses defining the property. For commands, we define a safety property to be any relation, $\_ \vDash \_\,\mathsf{safe} \subseteq \mathit{Env} \times \mathit{Cmd}$, which is defined from expression safety, according to Figure 6 (cf. Definition 5 for the stateful case).

2.2 Formulation of Safety Policies

A *safety policy* is a set of proof rules and auxiliary definitions which are designed to show that safe programs satisfy the safety property of interest. The intention is that a safety policy enforces a particular safety property (see Section 2.1). For source-level safety properties, the proof rules can be formalized concisely using the usual Hoare triples $P\ \{c\}\ Q$. We also use the notation $\vdash^{\mathtt{safe}} P\ \{c\}\ Q$ to

[5] A system is said to satisfy *subject reduction* [11] if types are preserved by term reduction, i.e., if $M : \tau$ and $M \Rightarrow M'$ implies $M' : \tau$.

denote derivability of Hoare triples, given a set of Hoare rules. Figure 7 shows the Hoare rules for operator safety. The rules are a slight modification of the standard ones; the (*assign*) axiom requires safety of the right-hand side expression, and the (*if*) and (*while*) rules require the additional hypothesis that the guard is safe under the precondition P. Figure 8 shows the definition of the auxiliary predicate $safe_{\text{op}}$ used in the rules; note that $safe_{\text{op}}$ is not a judgement but a function which maps expressions into formulae.

$$(skip)\quad \frac{}{Q\ \{\texttt{skip}\}\ Q}$$

$$(assign)\quad \frac{}{Q[e/x] \wedge safe_{\text{op}}(e)\ \{x\ \texttt{:=}\ e\}\ Q}$$

$$(if)\quad \frac{P \Rightarrow safe_{\text{op}}(b) \quad b \wedge P\ \{c\}\ Q \quad \neg b \wedge P \Rightarrow Q}{P\ \{\texttt{if}\, b\, \texttt{then}\, c\}\ Q}$$

$$(while)\quad \frac{P \Rightarrow safe_{\text{op}}(b) \quad b \wedge P\ \{c\}\ P}{P\ \{\texttt{while}\, b\, \texttt{do}\, c\}\ \neg b \wedge P}$$

$$(comp)\quad \frac{P\ \{c_1\}\ R \quad R\ \{c_2\}\ Q}{P\ \{c_1\ \texttt{;}\ c_2\}\ Q}$$

$$(cons)\quad \frac{P \Rightarrow P' \quad P'\ \{c\}\ Q' \quad Q' \Rightarrow Q}{P\ \{c\}\ Q}$$

Fig. 7. Hoare rules for $\mathcal{L}_0$ operator safety

$$safe_{\text{op}}(e) = \begin{cases} true & \text{if } e \in Var \text{ or } e \in Const \\ safe_{\text{op}}(e_1) \wedge safe_{\text{op}}(e_2) & \text{if } e \equiv e_1 \text{ op } e_2,\ \text{op} \in \{\texttt{*}, \texttt{+}, \texttt{-}, \texttt{=}\} \\ safe_{\text{op}}(e_1) \wedge safe_{\text{op}}(e_2) \wedge e_2 \neq 0 & \text{if } e \equiv e_1 \texttt{/} e_2 \end{cases}$$

Fig. 8. Safety formula for $\mathcal{L}_0$ operator safety

The standard Hoare rules are well-understood, unlike extensions to deal with safety. Our aim is to show how such extensions can be made automatically, while ensuring soundness and completeness. We first need to modify the standard interpretation of Hoare triples (i.e., $\eta \vDash P\ \{c\}\ Q$ iff $\eta \vDash P$ and $\langle c, \eta\rangle \Downarrow \eta'$ together imply $\eta' \vDash Q$) to take a safety judgement into account.

Definition 1. $\vDash^{\texttt{safe}} P\ \{c\}\ Q$ *holds iff for all* $\eta \in Env$*, if* $\eta \vDash P$*, then* $\eta \vDash c$ safe*, and if* $\langle c, \eta\rangle \Downarrow \eta'$*, then* $\eta' \vDash Q$*.*

Note that the proof rules inherit an underlying logic from a system given separately; in particular, they do not say anything about the definedness of the formulae P and Q used in the Hoare triples (e.g., $\vDash^{\texttt{safe}} true\ \{x := 0\}\ 1/x \neq 100$ holds). Hence, logical definedness is unconnected to the safety policy.

2.3 Soundness and Completeness of Safety Policies

The crucial task is now to show that the proof rules of the safety policy are sound and complete w.r.t. the safety property of interest. Since we have defined semantic safety of a command with respect to an environment we need to show a theorem of the form $\eta \vDash c$ safe *iff* $\vdash^{\texttt{safe}} P\ \{c\}\ \mathit{true}$, for some P such that $\eta \vDash P$. The role of the proof obligation P is to collect all the safety information for c in η.

For the *only if* direction of the proof (i.e., completeness), we need the notion of *expressivity* [11] which postulates the existence of formulae which characterise particular sets of environments. More precisely, we *assume* the existence of weakest preconditions *wpc* for all statements. Formally, a (first-order) language $\mathcal{L}$ is called *expressive* if, for all commands $c \in \mathit{Cmd}$ and postconditions Q, there exists a formula $\mathit{wpc}(c, Q)$ such that $\eta \vDash \mathit{wpc}(c, Q)$ iff $\langle c, \eta\rangle \Downarrow \eta'$ implies $\eta' \vDash Q$. This is a nontrivial assumption as there is no reason why an arbitrary semantic condition should be expressible by a (first-order) formula. However, the assumption is required for proof purposes only and in practice *wpc*s can often be computed automatically. As usual, `while`-loops pose the real problem, and here loop invariants have to be given explicitly.

Unfortunately, this standard definition of expressivity is not strong enough to show safety in all cases. Consider the example

```
i:=0;
while true do
  x:=1/(a-i); i:=i+1
```

which is safe in environments where `a` is negative but where the weakest precondition of the non-terminating loop is *true*, telling us nothing about its safety. Indeed, examples can be given which have *no* first-order wspc. We thus introduce the notion of *weakest safety precondition* (*wspc*) to characterize safe environments.

Definition 2. *(Expressivity for commands) A command $c \in \mathit{Cmd}$ is called* expressible *w.r.t. a safety judgement* safe *if, for all postconditions Q, there exists a formula $\mathit{wspc}(c, Q)$ such that*

$$\eta \vDash \mathit{wspc}(c, Q)\ \textit{iff}\ (\eta \vDash c\ \textsf{safe}\ \textit{and}\ \langle c, \eta\rangle \Downarrow \eta'\ \textit{implies}\ \eta' \vDash Q).$$

A language $\mathcal{L}$ is called expressive for commands w.r.t. a safety judgement safe *if all commands are expressible.*

Now a consequence of the definition of *wspc* is that all intermediate commands are safe, by safety reduction. However, since there is no useful notion of safe environment, it is not sufficient to simply consider the environments in which c reduces to a safe environment, or for which all intermediate environments are safe. We also need to extend expressivity to the expression level; here it assumes the existence of safety formulae, $\mathit{safe}(e)$, compatible with the safety judgement safe.

Definition 3. *(Expressivity for expressions) An expression $e \in Expr$ is called* expressible *w.r.t. a safety judgement* safe *if there exists a formula $safe(e)$ such that $\eta \vDash e\ \mathsf{safe}$ iff $\eta \vDash safe(e)$.*

By abuse of notation we will also call a given safety predicate $safe(\_)$ expressive for a safety judgement safe if it satisfies the condition of Definition 3. It is then easy to show by straightforward induction over expressions that $safe_{\mathrm{op}}$ is expressive for $\mathsf{safe}_{\mathrm{op}}$.

Lemma 3. *For all $e \in Expr$, $\eta \vDash e\ \mathsf{safe}_{op}$ iff $\eta \vDash safe_{op}(e)$.*

We can now characterize the weakest safety preconditions *wspc* (w.r.t. operator safety) for each command of $\mathcal{L}_0$. Lemma 4 thus gives a recursive (but due to the `while`-case unfortunately not well-founded) definition of *wspc*.

Lemma 4. *Assuming all formulae exist, the following equivalences are sound:*

1. $wspc(\mathtt{skip}, Q) \iff wpc(\mathtt{skip}, Q)$
2. $wspc(x := e, Q) \iff safe_{op}(e) \wedge wpc(x := e, Q)$
3. $wspc(\mathtt{if}\ b\ \mathtt{then}\ c, Q) \iff safe_{op}(b) \wedge (b \Rightarrow wspc(c, Q)) \wedge (\neg b \Rightarrow Q)$
4. $wspc(\mathtt{while}\ b\ \mathtt{do}\ c, Q) \iff safe_{op}(b) \wedge (b \Rightarrow wspc(c, wspc(\mathtt{while}\ b\ \mathtt{do}\ c, Q))) \wedge (\neg b \Rightarrow Q)$
5. $wspc(c_1 ; c_2, Q) \iff wspc(c_1, wspc(c_2, Q)))$

The preceding lemma does not give a constructive definition of *wspc*, because of the recursion in the `while`-case.

Lemma 5. *(wspc properties) For all formulas P and Q, and commands, c:*

1. $\vDash^{\mathsf{safe}} wspc(c, Q)\ \{c\}\ Q$.
2. $\vDash^{\mathsf{safe}} P\ \{c\}\ Q$ *implies* $P \Rightarrow wspc(c, Q)$.

Proof: 1. By definition of *wspc*. 2. The implication is clearly true in the model. Provability follows from completeness of the underlying logic. ∎

We can now extend the definition of safety formulae to commands via a reduction to *wspc*. We define $safe_{\mathrm{op}}(c) = wspc(c, true)$, which also yields $\vDash^{\mathsf{safe}} safe_{\mathrm{op}}(c)\ \{c\}\ true$, for all $c \in Cmd$, as a special case of Lemma 5. Moreover, we clearly have $\eta \vDash c\ \mathsf{safe}_{\mathrm{op}}$ iff $\eta \vDash safe_{\mathrm{op}}(c)$, so can factor *wspc* into a functional component expressed in terms of the standard precondition *wpc* and a safety component $safe_{\mathrm{op}}(c)$.

Proposition 1. $wspc(c, Q) \iff wpc(c, Q) \wedge safe_{op}(c)$.

Note that we choose not to define *wspc* this way i.e., by giving a direct definition of $safe_{\mathrm{op}}(c)$. The reason is that checking safety requires a similar recursive descent over the structure of a command, similar to computing the *wpc*, so it is more natural to combine them into a single definition. Similarly, it is not possible to give a neat definition of *wspc* from *wpc* and safety of expressions, for the reasons given in Section 2.

Theorem 1. *Suppose c is expressible. Then, $\vDash^{\mathtt{safe}} P\ \{c\}\ Q$ iff $\vdash^{\mathtt{safe}} P\ \{c\}\ Q$.*

Proof: Soundness is by induction over the derivation. For completeness, the proof structure follows that of the standard (relative) completeness proof for Hoare logic, using expressivity to get, in our case, the weakest safety preconditions which are needed to make the proof go through. The most interesting cases are for conditionals and `while`-loops.
(`if`) Let R denote $wspc(\mathtt{if}\, b\, \mathtt{then}\, c, Q)$. Then:

$$\dfrac{\dfrac{\dfrac{}{R \Rightarrow safe(b)}\,(1) \quad \dfrac{\dfrac{}{b \wedge R \Rightarrow wspc(c,Q)}\,(2) \quad \dfrac{}{wspc(c,Q)\ \{c\}\ Q}\,(3)}{b \wedge R\ \{c\}\ Q}}{R\ \{\mathtt{if}\, b\, \mathtt{then}\, c\}\ Q} \quad \dfrac{}{\neg b \wedge R \Rightarrow Q}\,(4)}{P\ \{\mathtt{if}\, b\, \mathtt{then}\, c\}\ Q} \quad \dfrac{}{P \Rightarrow R}\,(5)$$

The first, second, and fourth hypotheses follow from Lemma 4, the third and fifth follow from Lemma 5 (parts 1 and 2, respectively).
(`while`) Suppose $\vDash^{\mathtt{safe}} P\ \{\mathtt{while}\, b\, \mathtt{do}\, c\}\ Q$. Let R denote $wspc(\mathtt{while}\, b\, \mathtt{do}\, c, Q)$. Then:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{R \Rightarrow safe_{\mathrm{op}}(b)}\,(1) \quad \dfrac{\dfrac{}{b \wedge R \Rightarrow wspc(c,R)}\,(2) \quad \dfrac{}{wspc(c,R)\ \{c\}\ R}\,(3)}{b \wedge R\ \{c\}\ R}}{R\ \{\mathtt{while}\, b\, \mathtt{do}\, c\}\ \neg b \wedge R} \quad \dfrac{}{\neg b \wedge R \Rightarrow Q}\,(4)}{R\ \{\mathtt{while}\, b\, \mathtt{do}\, c\}\ Q} \quad \dfrac{}{P \Rightarrow R}\,(5)}{P\ \{\mathtt{while}\, b\, \mathtt{do}\, c\}\ Q}$$

The first, second and fourth hypothesis follow from Lemma 4, the third follows from the inductive hypothesis on c and Lemma 5(1); and the fifth follows from Lemma 5(2). ∎

Theorem 2. *Assume expressivity. Then, $\eta \vDash c\ \mathsf{safe}_{op}$ iff $\vdash^{\mathtt{safe}} \phi\ \{c\}$ true for some ϕ such that $\eta \vDash \phi$.*

Proof: We show the left-to-right implication. We know that $\vDash^{\mathtt{safe}} safe_{\mathrm{op}}(c)\ \{c\}$ *true* by Lemma 5. Hence, by Theorem 1, we have that $\vdash^{\mathtt{safe}} safe_{\mathrm{op}}(c)\ \{c\}$ *true*, and since $\eta \vDash c\ \mathsf{safe}_{\mathrm{op}}$ by assumption, expressivity gives us $\eta \vDash safe_{\mathrm{op}}(c)$. ∎

At this point it might look like we have built a formidable machinery to prove some less than formidable properties. However, subtle variations of the Hoare rules are possible, and finding the "right" rules (much less proving that they are right) is difficult without a formal framework like the one we have developed here. Consider, for example, the following variant of the *if*-rule

$$(\mathit{if}\,')\ \frac{safe_{\mathrm{op}}(b) \wedge b \wedge P\ \{c\}\ Q \quad safe_{\mathrm{op}}(b) \wedge \neg b \wedge P \Rightarrow Q}{P\ \{\mathtt{if}\, b\, \mathtt{then}\, c\}\ Q}$$

in which the safety formula is "inlined" into the two hypotheses and not separated into a third hypothesis (cf. Figure 7). However, this rule variant allows

safety information to be used to determine the control flow, which makes it potentially unsound. It allows us to derive the triple

$$\mathit{true}\ \{\texttt{if 1/x} \neq \texttt{1 then if x} \neq \texttt{0 then y:=3}\}\ x = 1 \vee y = 3$$

which on the surface seems reasonable: either x is one and nothing can be concluded about y, or x is non-zero and y is assigned, or x is zero, the outer guard is undefined, and hence, the statement causes an exception and does not terminate properly. However, it is exactly this third alternative which causes the trouble: if division by zero does *not* cause an exception but returns a defined value (e.g., *NaN*, "not a number"), we can no longer conclude at the inner guard that the safety formula on the outer guard holds.

We note in passing that the rules in this paper are different from those in [19]. However, we believe that the rules shown here are easier to implement and apply in practice.

3 Stateful Safety Properties

For operator safety, the property itself was defined in terms of the original environments only. Most safety properties, however, are not that simple and require additional information: memory safety requires information about the size of arrays or the number of variable accesses, domain-specific policies such as frame safety require additional typing information which is not part of the operational semantics of the language, and so on. We now extend our framework to deal with such safety properties.

Our basic idea is to introduce a distinct auxiliary (or *shadow*) variable $\overline{x} \in \overline{\mathit{Var}}$ for each variable $x \in \mathit{Var}$, which records the necessary safety information associated with x. We also introduce shadow environments $\overline{\eta} : \overline{\mathit{Var}} \rightharpoonup \overline{\mathit{Val}}$, where the shadow domain $\overline{\mathit{Val}}$ depends on the safety property of interest, and extend the operational semantics to include the effects the different commands have on the values of the shadow variables. We then modify the Hoare rules to ensure that $\overline{x}$ actually "shadows" x, i.e., that the information recorded in $\overline{x}$ is always current.

We already adopted part of this methodology in [19]; one motivation for the present work is to formally justify it. The methodology itself is quite flexible and allows us to encode different safety properties, using different shadow domains. We illustrate our approach first for memory safety (more precisely, array bounds checks), and then show how two other, less typical safety policies can be encoded.

3.1 Memory Safety

For memory safety, we need to extend our language $\mathcal{L}_0$ by simple arrays; here, we restrict ourselves to one-dimensional arrays with a fixed lower bound of zero to simplify the presentation. Figure 9 shows the syntax of the extended language $\mathcal{L}_1$. As usual, we add array updates to the commands and array selects to the

expressions. However, we also require explicit array declarations of the form `var` x`[`n`]`, which declares an n-element array x.[6]

$$
\begin{array}{lll}
\mathit{Cmd} ::= \ldots & & \mathit{Expr} ::= \ldots \\
\quad \mid \ \mathit{Var}\texttt{[}\mathit{Expr}\texttt{]} \ \texttt{:=} \ \mathit{Expr} & & \quad \mid \ \mathit{Var}\texttt{[}\mathit{Expr}\texttt{]} \\
\quad \mid \ \mathit{Decl} & & \\
\mathit{Decl} ::= \texttt{var}\ \mathit{Var} & & \\
\quad \mid \ \texttt{var}\ \mathit{Var}\texttt{[}\mathit{Const}\texttt{]} & &
\end{array}
$$

Fig. 9. Syntax of extended *while*-language $\mathcal{L}_1$

For memory safety, the shadow environment needs to record the size of each array; we thus have $\overline{\eta} : \overline{\mathit{Var}} \rightharpoonup I\!N$. Eventually, the shadow variables get their values from the declarations. This differs from the usual approach where the array bounds are represented by an extra function $\mathit{high}(x)$ on the logical level.

Since we now have two environments, we have to slightly extend some parts of our machinery. This includes interpretations, the operational semantics, and the safety judgements. For interpretations, the only difference is in the case of variables, which need to be taken from the correct environment:

$$
\begin{array}{l}
[\![x]\!]_{\eta,\overline{\eta}} \ = \eta(x) \\
[\![x_{\mathrm{hi}}]\!]_{\eta,\overline{\eta}} = \overline{\eta}(x_{\mathrm{hi}})
\end{array}
$$

In the operational semantics, the only case interesting for memory safety is the array declaration; all other constructs leave the shadow environment unchanged.[7]

$$
\begin{array}{ll}
\langle \texttt{var}\ x, \eta, \overline{\eta}\rangle & \Rightarrow \langle \texttt{skip}, \eta, \overline{\eta}\rangle \\
\langle \texttt{var}\ x\texttt{[}n\texttt{]}, \eta, \overline{\eta}\rangle & \Rightarrow \langle \texttt{skip}, \eta, \overline{\eta} \oplus \{x_{\mathrm{hi}} \mapsto [\![n]\!]_{\eta,\overline{\eta}}\rangle \\
\langle x\texttt{[}e_1\texttt{]} \ \texttt{:=} \ e_2, \eta, \overline{\eta}\rangle & \Rightarrow \langle \texttt{skip}, \eta \oplus \{x \mapsto (x \oplus \{[\![e_1]\!]_{\eta,\overline{\eta}} \mapsto [\![e_2]\!]_{\eta,\overline{\eta}}\})\}, \overline{\eta}\rangle \\
\langle c, \eta, \overline{\eta}\rangle & \Rightarrow \langle c', \eta', \overline{\eta}\rangle, \ \text{if } \langle c, \eta\rangle \Rightarrow \langle c', \eta'\rangle
\end{array}
$$

As in the stateless case, we can then define the safety judgement for memory safety. Figure 10 shows both judgements for expressions and commands.

Again following the schema developed for the stateless case, we then formulate the Hoare rules of the safety policy, as shown in Figure 11; we have omitted the rules (*skip*), (*comp*), and (*cons*) which remain unchanged. In the rules (*assign*), (*if*), and (*while*), the safety predicate is changed. The (*update*)-rule is an appropriately modified version of McCarthy's original rule.

The lemmas and theorems of the previous section hold in a suitably modified form. The main change is to modify the expansions of *wspc*. The key cases are

[6] For consistency, we also add scalar declarations `var` x.

[7] We also need to specify how array selection and updates are modeled; however, this is a consequence of extending the language and is independent of any certification issues. Here, we model arrays as maps from naturals to values; hence: $[\![x[e]]\!]_{\eta,\overline{\eta}} = (\eta(x))([\![e]\!]_{\eta,\overline{\eta}})$

$$
\begin{array}{ll}
\eta, \overline{\eta} \vDash c \ \mathsf{safe}_{\mathrm{mem}} & \\
\eta, \overline{\eta} \vDash x \ \mathsf{safe}_{\mathrm{mem}} & \\
\eta, \overline{\eta} \vDash x\,\texttt{[}e\texttt{]} \ \mathsf{safe}_{\mathrm{mem}} & \text{iff } \ 0 \leq [\![e]\!]_{\eta,\overline{\eta}} < \overline{\eta}(x_{\mathrm{hi}}) \text{ and } \eta, \overline{\eta} \vDash e \ \mathsf{safe}_{\mathrm{mem}} \\
\eta, \overline{\eta} \vDash e_1 \text{ op } e_2 \ \mathsf{safe}_{\mathrm{mem}} & \text{iff } \ \eta, \overline{\eta} \vDash e_1 \ \mathsf{safe}_{\mathrm{mem}} \text{ and } \eta, \overline{\eta} \vDash e_2 \ \mathsf{safe}_{\mathrm{mem}} \\
\\
\eta, \overline{\eta} \vDash \texttt{var } x \ \mathsf{safestate}_{\mathrm{mem}} & \\
\eta, \overline{\eta} \vDash \texttt{var } x\texttt{[}n\texttt{]} \ \mathsf{safestate}_{\mathrm{mem}} & \\
\eta, \overline{\eta} \vDash \texttt{skip} \ \mathsf{safestate}_{\mathrm{mem}} & \\
\eta, \overline{\eta} \vDash e_1 \ \texttt{:=} \ e_2 \ \mathsf{safestate}_{\mathrm{mem}} & \text{iff } \ \eta, \overline{\eta} \vDash e_1 \ \mathsf{safe}_{\mathrm{mem}} \text{ and } \eta, \overline{\eta} \vDash e_2 \ \mathsf{safe}_{\mathrm{mem}} \\
\eta, \overline{\eta} \vDash \texttt{if } b \texttt{ then } c \ \mathsf{safestate}_{\mathrm{mem}} & \text{iff } \ \eta, \overline{\eta} \vDash b \ \mathsf{safe}_{\mathrm{mem}} \\
\eta, \overline{\eta} \vDash \texttt{while } b \texttt{ do } c \ \mathsf{safestate}_{\mathrm{mem}} & \text{iff } \ \eta, \overline{\eta} \vDash b \ \mathsf{safe}_{\mathrm{mem}} \\
\eta, \overline{\eta} \vDash c_1\texttt{;}c_2 \ \mathsf{safestate}_{\mathrm{mem}} & \text{iff } \ \eta, \overline{\eta} \vDash c_1 \ \mathsf{safestate}_{\mathrm{mem}}
\end{array}
$$

$$
\eta, \overline{\eta} \vDash c \ \mathsf{safe}_{\mathrm{mem}} \text{ iff } \forall \langle c, \eta, \overline{\eta} \rangle \Rightarrow^* \langle c', \eta', \overline{\eta}; \rangle \cdot \eta', \overline{\eta}; \vDash c' \ \mathsf{safestate}_{\mathrm{mem}}
$$

Fig. 10. $\mathcal{L}_1$ memory safety

$$
\begin{array}{ll}
(\mathit{decl}) & \dfrac{}{Q\ \{\texttt{var } x\}\ Q} \\
(\mathit{adecl}) & \dfrac{}{Q[n/x_{\mathrm{hi}}]\ \{\texttt{var } x\texttt{[}n\texttt{]}\}\ Q} \\
(\mathit{assign}) & \dfrac{}{Q[e/x] \wedge \mathit{safe}_{\mathrm{mem}}(e)\ \{x \ \texttt{:=}\ e\}\ Q} \\
(\mathit{update}) & \dfrac{}{Q[\mathit{update}(x, e_1, e_2)/x] \wedge \mathit{safe}_{\mathrm{mem}}(x\texttt{[}e_1\texttt{]}) \wedge \mathit{safe}_{\mathrm{mem}}(e_2)\ \{x\texttt{[}e_1\texttt{]} \ \texttt{:=}\ e_2\}\ Q} \\
(\mathit{if}) & \dfrac{P \Rightarrow \mathit{safe}_{\mathrm{mem}}(b) \quad b \wedge P\ \{c\}\ Q \quad \neg b \wedge P \Rightarrow Q}{P\ \{\texttt{if } b \texttt{ then } c\}\ Q} \\
(\mathit{while}) & \dfrac{P \Rightarrow \mathit{safe}_{\mathrm{mem}}(b) \quad b \wedge P\ \{c\}\ P}{P\ \{\texttt{while } b \texttt{ do } c\}\ \neg b \wedge P}
\end{array}
$$

Fig. 11. Hoare rules for $\mathcal{L}_1$ memory safety

$$
\begin{array}{lll}
\mathit{wspc}(\texttt{var } x[n], Q) & \iff & Q[0/x_{\mathrm{hi}}] \\
\mathit{wspc}(x\texttt{[}e_1\texttt{]} \ \texttt{:=}\ e_2, Q) & \iff & Q[\mathit{update}(x, e_1, e_2)/x] \wedge \mathit{safe}_{\mathrm{mem}}(x[e_1]) \wedge \mathit{safe}_{\mathrm{mem}}(e_2)
\end{array}
$$

3.2 Memory Write Limits

Next, we consider a safety policy which limits the number of times values can be written into each memory location. Obviously, this is undecidable in general, but with appropriate annotations (i.e., loop invariants) it can still be very helpful. Such a policy can then be used to ensure that the physical limitations of non-volatile memory, as for example used in spacecraft, are not exceeded. For example, locations in flash memory can only be written to a finite number of times before wearing out.

$$safe_{\text{mem}}(e) = \begin{cases} true & \text{if } e \in Var \text{ or } e \in Const \\ safe_{\text{mem}}(e_1) \wedge 0 \leq e_1 < x_{\text{hi}} & \text{if } e \equiv x\texttt{[}e_1\texttt{]} \\ safe_{\text{mem}}(e_1) \wedge safe_{\text{mem}}(e_2) & \text{if } e \equiv e_1 \text{ mem } e_2,\ \text{op} \in \{\texttt{*}, \texttt{/}, \texttt{+}, \texttt{-}, \texttt{=}\} \end{cases}$$

Fig. 12. Safety formula for $\mathcal{L}_1$ memory safety

We formalize this using shadow variables x_{wl} which are initialized with zero when x is declared and incremented each time it is assigned to. As in the case of memory safety, the abstract environments map the variables to naturals, $\overline{\eta} : \overline{Var} \rightharpoonup I\!N$. However, unlike in the case of memory safety, we now need (i) shadow variables for scalars as well, and (ii) a separate shadow variable for each element of an array. While the first point is straightforward to deal with, the second seems at first more complicated. However, by introducing a complete shadow array, we get around all these problems. In the operational semantics we then see a nice symmetry between the operations on the original value environment and on the shadow environment:

$$\begin{array}{ll}
\langle \texttt{var}\ x, \eta, \overline{\eta}\rangle & \Rightarrow \langle \texttt{skip}, \eta, \overline{\eta} \oplus \{x_{\text{wl}} \mapsto 0\}\rangle \\
\langle \texttt{var}\ x\texttt{[}n\texttt{]}, \eta, \overline{\eta}\rangle & \Rightarrow \langle \texttt{skip}, \eta, \overline{\eta} \oplus \{x_{\text{wl}} \mapsto \lambda i \cdot 0\}\rangle \\
\langle x\ \texttt{:=}\ e, \eta, \overline{\eta}\rangle & \Rightarrow \langle \texttt{skip}, \eta \oplus \{x \mapsto [\![e]\!]_\eta\}, \overline{\eta} \oplus \{x_{\text{wl}} \mapsto \overline{\eta}(x_{\text{wl}}) + 1\}\rangle \\
\langle x\texttt{[}e_1\texttt{]}\ \texttt{:=}\ e_2, \eta, \overline{\eta}\rangle & \Rightarrow \langle \texttt{skip}, \\
& \quad \eta \oplus \{x \mapsto (x \oplus \{[\![e_1]\!]_{\eta,\overline{\eta}} \mapsto [\![e_2]\!]_{\eta,\overline{\eta}}\})\}, \\
& \quad \overline{\eta} \oplus \{x_{\text{wl}} \mapsto (x_{\text{wl}} \oplus \{[\![e_1]\!]_{\eta,\overline{\eta}} \mapsto x_{\text{wl}}([\![e_1]\!]_{\eta,\overline{\eta}}) + 1\})\} \\
& \quad \rangle \\
\langle c, \eta, \overline{\eta}\rangle & \Rightarrow \langle c', \eta', \overline{\eta}\rangle,\ \text{if } \langle c, \eta\rangle \Rightarrow \langle c', \eta'\rangle
\end{array}$$

The safety judgement $\mathsf{safe}_{\text{wl}}$ obviously only needs to look at assignments; it just checks that the assignment counts are still below a fixed upper limit MAXWR. Since safety reduction holds trivially, we formulate $\mathsf{safe}_{\text{wl}}$ directly and not via safestate.

$$\begin{array}{lll}
\eta, \overline{\eta} \vDash x\ \texttt{:=}\ e\ \mathsf{safe}_{\text{wl}} & \text{iff} & \overline{\eta}(x_{\text{wl}}) < \text{MAXWR} \\
\eta, \overline{\eta} \vDash x\texttt{[}e_1\texttt{]}\ \texttt{:=}\ e_2\ \mathsf{safe}_{\text{wl}} & \text{iff} & (\overline{\eta}(x_{\text{wl}}))([\![e_1]\!]_{\eta,\overline{\eta}}) < \text{MAXWR}
\end{array}$$

Finally, we formulate the Hoare rules (cf. Figure 13); again, the only interesting cases are declarations and assignments. We thus omit an explicit definition of the safety formula and inline it instead. Note that we extend the logic for arrays by the construct $init(x, n, k)$ which denotes the array x of size n where every element is set to k. For this, we need the axiom $i < n \Rightarrow (init(x, n, k))(i) = k$ in the domain theory of the underlying logic (not shown here).

Again, we can show that the system is sound and complete with respect to the corresponding semantics. The proofs follow the outline in Section 2.

$$(decl)\quad \overline{Q[0/x_{\mathrm{wl}}]\ \{\texttt{var}\ x\}\ Q}$$

$$(adecl)\quad \overline{Q[init(x_{\mathrm{wl}}, n, 0)/x_{\mathrm{wl}}]\ \{\texttt{var}\, x\texttt{[}n\texttt{]}\}\ Q}$$

$$(assign)\quad \overline{Q[e/x, (x_{\mathrm{wl}}+1)/x_{\mathrm{wl}}] \wedge x_{\mathrm{wl}} < \textsc{Maxwr}\ \{x\ \texttt{:=}\ e\}\ Q}$$

$$(update)\quad \overline{Q\begin{bmatrix} update(x, e_1, e_2)/x, \\ update(x_{\mathrm{wl}}, e_1, x_{\mathrm{wl}}[e_1]{+}1)/x_{\mathrm{wl}} \end{bmatrix} \wedge x_{\mathrm{wl}}\texttt{[}e_1\texttt{]} < \textsc{Maxwr}\ \{x\texttt{[}e_1\texttt{]}\ \texttt{:=}\ e_2\}\ Q}$$

Fig. 13. Hoare rules for $\mathcal{L}_1$ write limits

3.3 Memory Read Limits

The final safety policy we consider in this paper limits the number of times memory locations can be read. Intuitively, this is the dual of the write limit policy considered above; formally, however, it is quite different. The reason for the difference (and the source of additional complexity) is that the updates of the shadow environment are now much less localized: the evaluation of each expression can potentially change the shadow environment. This problem is not restricted to read limits but occurs whenever expression evaluation can have side effects, either in the original environment, or in the shadow environment. We thus extend the evaluation notation for expressions to take the environments into account, i.e., $\langle e, \eta, \overline{\eta}\rangle \Downarrow \langle v, \eta', \overline{\eta}'\rangle$.

To simplify our notation we define a shadow environment update function $upd : Env \times \overline{Env} \times Expr \to \overline{Env}$ which examines the expression and adds the correct number of occurrences to the shadow environment; the notation $y \in_n e$ denotes that there are n occurrences of the variable y in e:

$$\begin{aligned} upd(\eta, \overline{\eta}, e) = \overline{\eta} \oplus \{&x_{\mathrm{rl}} \mapsto \overline{\eta}(x_{\mathrm{rl}}) + n \mid x \in_n e\} \\ \oplus \{&x_{\mathrm{rl}} \mapsto x_{\mathrm{rl}} \oplus \{[\![e']\!]_{\eta,\overline{\eta}} \mapsto x_{\mathrm{rl}}([\![e']\!]_{\eta,\overline{\eta}}) + n\} \mid x\texttt{[}e'\texttt{]} \in_n e\} \end{aligned}$$

We can then formulate the operational semantics concisely; the omitted cases follow easily.

$$\begin{aligned}
&\langle \texttt{var}\ x, \eta, \overline{\eta}\rangle && \Rightarrow \langle \texttt{skip}, \eta, \overline{\eta} \oplus \{x_{\mathrm{rl}} \mapsto 0\}\rangle \\
&\langle \texttt{var}\, x\texttt{[}n\texttt{]}, \eta, \overline{\eta}\rangle && \Rightarrow \langle \texttt{skip}, \eta, \overline{\eta} \oplus \{x_{\mathrm{rl}} \mapsto \lambda i \cdot 0\}\rangle \\
&\langle x\ \texttt{:=}\ e, \eta, \overline{\eta}\rangle && \Rightarrow \langle \texttt{skip}, \eta \oplus \{x \mapsto [\![e]\!]_{\eta,\overline{\eta}}\}, upd(\eta, \overline{\eta}, e)\rangle \\
&\langle x\texttt{[}e_1\texttt{]}\ \texttt{:=}\ e_2, \eta, \overline{\eta}\rangle && \Rightarrow \langle \texttt{skip}, \\
&&& \quad \eta \oplus \{x \mapsto (x \oplus \{[\![e_1]\!]_{\eta,\overline{\eta}} \mapsto [\![e_2]\!]_{\eta,\overline{\eta}}\})\}, \\
&&& \quad upd(\eta, upd(\eta, \overline{\eta}, e_1), e_2) \\
&&& \rangle \\
&\langle \texttt{if}\, b\, \texttt{then}\, c, \eta, \overline{\eta}\rangle && \Rightarrow \langle c, \eta, upd(\eta, \overline{\eta}, b)\rangle \text{ if } [\![b]\!]_{\eta,\overline{\eta}} = true \\
&\langle \texttt{if}\, b\, \texttt{then}\, c, \eta, \overline{\eta}\rangle && \Rightarrow \langle \texttt{skip}, \eta, upd(\eta, \overline{\eta}, b)\rangle \text{ if } [\![b]\!]_{\eta,\overline{\eta}} = false
\end{aligned}$$

In effect, we can give the semantics in terms of the basic underlying semantics and the update function on the shadow environments: if $\langle c, \eta\rangle \Rightarrow \langle c', \eta'\rangle$, then

$\langle c, \eta, \overline{\eta}\rangle \Rightarrow \langle c', \eta', \mathit{upd}(\eta, \overline{\eta}', e_1, \ldots, e_n)\rangle$ for immediate subexpressions $e_1, \ldots, e_n$ of c (extending *upd* to lists of expressions in the obvious way). We can also apply the same idea to the Hoare rules. Instead of an update function which is applied to the shadow environment we need an update substitution $\mathsf{Sub}(e)$ which is applied to the precondition; it is defined in the same way as the update function:

$$\mathsf{Sub}(e) = [x_{\mathrm{rl}} + n/x_{\mathrm{rl}} \mid x \in_n e] \cup [\mathit{update}(x_{\mathrm{rl}}, e', x_{\mathrm{rl}}\texttt{[}e'\texttt{]} + n)/x_{\mathrm{rl}} \mid x\texttt{[}e'\texttt{]} \in_n e]$$

We then define the safety formula $\mathit{safe}_{\mathrm{rl}}(e)$ in the same way: it checks that the occurrences in e do not exceed the limit MAXRL:

$$\mathit{safe}_{\mathrm{rl}}(e) = \bigwedge_{x \in_n e} x_{\mathrm{rl}} + n \leq \textsc{Maxrl} \quad \wedge \bigwedge_{x[e'] \in e} (\mathit{fold}_{x[e'] \in_n e}(\mathit{upd}(n), x_{\mathrm{rl}}))[e'] \leq \textsc{Maxrl}$$

where the folded update of x_{rl} by all *literal* occurrences $x[e']$ in e is defined using:

$$\mathit{upd}(n)(x[e'], x_{\mathrm{rl}}) = \mathit{update}(x_{\mathrm{rl}}, e', x_{\mathrm{rl}}[e'] + n).$$

The safety judgements are similar to those for write limits. The only change is that since expression evaluation can affect the shadow environment, we need to add the the safety condition outside the substitution. With this, we have all the pieces in place to formulate the Hoare rules. We only give a single rule for the `if`-statement; the other rules follow the same schema.

$$(\mathit{if})\ \frac{b \wedge P\ \{c\}\ Q \quad \neg b \wedge P \Rightarrow Q}{\mathsf{Sub}^b(P) \wedge \mathit{safe}_{\mathrm{rl}}(b)\ \{\texttt{if}\, b\, \texttt{then}\, c\}\ Q}$$

4 Automatic Derivation of Safety Policies

We now generalize the idea from Section 3.3 and derive a general way of formulating safety extensions to an operational semantics and Hoare logic, respectively, such that the results of the previous sections are preserved. The main idea is to develop a notion of *compositional* safety property, which then allows us to augment the Hoare rules in a similarly compositional manner.

We have seen that abstract environments describe how programs compute the abstract properties we are interested in for a given safety property. In order to reason about such properties in a safety policy, we need a notion of expressivity to relate environments to the logic.

Definition 4. *We say that a command $c \in \mathit{Cmd}$ is* operationally expressive, *if whenever $\langle c, \eta, \bar{\eta}\rangle \Rightarrow \langle c', \eta', \bar{\eta}'\rangle$, then for all $x \in (\eta' \cup \bar{\eta}')$, there exists an expression e, such that $[\![e]\!]_{\eta,\bar{\eta}} = [\![x]\!]_{\eta',\bar{\eta}'}$.* ∎

This formalizes the idea that whatever change a command makes to the environments can be expressed in terms of substitutions. Clearly, the expression can only contain variables from the original environments.

We use the notation $\mathsf{Sub}_\theta^{e_1,\ldots,e_n}(P)$ to denote the substitution, applied to P, which expresses the change in environments effected by command type θ with immediate subexpressions $e_1, \ldots, e_n$. We are implicitly assuming particularly simple changes to the environment which can always be expressed this way, but this accounts for all our examples. For example, $\mathsf{Sub}_{\mathtt{assign}}^{x,e}(P)$ is simply $P[e/x]$ for the assignment $x := e$.

In general, each command has its own notion of safety. However, we want to exclude pathological examples of safety properties, so we consider, now, what sort of properties are acceptable. For atomic commands, we allow an arbitrary condition on the environments and the component expressions. For example, the safety of the assignment $x := e$ can be any condition on x and e. We can express this as a predicate $P \subseteq \mathit{Env} \times \overline{\mathit{Env}} \times \mathit{Expr} \times \mathit{Expr}$. For compound commands, the key idea is that the basic data of a safety property consists of arbitrary predicates, Cond, on the immediately accessible subexpressions for each command. We will write $\eta, \overline{\eta} \vDash \mathsf{Cond}(e_1, \ldots, e_n)$ to mean $\langle \eta, \overline{\eta}, e_1, \ldots, e_n \rangle \in \mathsf{Cond}$.

Definition 5. *A safety property on commands is* compositional*, if there exist predicates* Cond_θ*, with the following properties:*

- $\eta, \overline{\eta} \vDash \mathtt{var}\ x\ \mathsf{safe}$ *iff* $\eta, \overline{\eta} \vDash \mathsf{Cond}_{\mathtt{decl}}(x)$
- $\eta, \overline{\eta} \vDash \mathtt{var}\ x[n]\ \mathsf{safe}$ *iff* $\eta, \overline{\eta} \vDash \mathsf{Cond}_{\mathtt{adecl}}(x, n)$
- $\eta, \bar{\eta} \vDash x := e\ \mathsf{safe}$ *iff* $\eta, \bar{\eta} \vDash \mathsf{Cond}_{\mathtt{assign}}(x, e)$
- $\eta, \bar{\eta} \vDash x[e_1] := e_2\ \mathsf{safe}$ *iff* $\eta, \bar{\eta} \vDash \mathsf{Cond}_{\mathtt{update}}(x, e_1, e_2)$
- $\eta, \overline{\eta} \vDash \mathtt{skip}\ \mathsf{safe}$
- $\eta, \overline{\eta} \vDash \mathtt{if}\ b\ \mathtt{then}\ c\ \mathsf{safe}$ *iff* $\mathsf{Cond}_{\mathtt{if}}(b)$ *and* $\langle b, \eta, \overline{\eta} \rangle \Downarrow \langle \mathit{true}, \eta', \overline{\eta}' \rangle$ *implies* $\langle \eta', \overline{\eta}' \rangle \vDash c\ \mathsf{safe}$
- $\eta, \overline{\eta} \vDash \mathtt{while}\ b\ \mathtt{do}\ c\ \mathsf{safe}$ *iff* $\eta, \overline{\eta} \vDash \mathsf{Cond}_{\mathtt{while}}(b)$ *and* $\langle b, \eta, \overline{\eta} \rangle \Downarrow \langle \mathit{true}, \eta', \overline{\eta}' \rangle$ *implies* $(\langle \eta', \overline{\eta}' \rangle \vDash c\ \mathsf{safe}$ *and* $\langle c, \eta', \overline{\eta}' \rangle \Downarrow \langle \eta'', \overline{\eta}'' \rangle$ *implies* $\langle \eta'', \overline{\eta}'' \rangle \vDash \mathtt{while}\ b\ \mathtt{do}\ c\ \mathsf{safe})$.

For sequential composition, the safety of $c_1; c_2$ is defined as before. Although this looks fairly similar to Figure 6 it generalizes it by allowing arbitrary conditions on the expressions. Stateless safety follows as the special case where $\eta, \overline{\eta} \vDash \mathsf{Cond}_\theta(e_1, \ldots, e_n)$ iff $\eta, \overline{\eta} \vDash e_i\ \mathsf{safe}$, for each i.

This notion of compositionality maintains the correspondence between safe and $\mathsf{safestate}$, while allowing that safety of a command is arbitrarily expressed in terms of the safety of its subcommands. Now it should come as no surprise that we require the condition predicates to be expressible.

Definition 6. *We say that the n-ary predicate, P, is expressible when there exists formulas* ϕ *such that*

$$\langle e_1, \ldots, e_n \rangle \in P \text{ iff } \eta, \overline{\eta} \vDash \phi(e_1, \ldots, e_n).$$

Finally, we are in a position to state a general completeness theorem, which generalizes the theory of stateless safety developed in Section 2. We omit the details of the proof here and just state the theorem; the proof structure is the same as for the stateless case, making use of expressivity where appropriate.

Theorem 3. *Given (*i*) a set, $\overline{Val}$ (the shadow domain), (*ii*) an operational semantics, $\langle c, \eta, \bar{\eta}\rangle \Rightarrow \langle c, \eta', \bar{\eta}'\rangle$, and (*iii*) a compositional safety property, such that expressivity (operational, predicate, commands and expressions) holds, the following system is sound and complete with respect to the safety property:*

$$(decl)\quad \frac{}{\mathsf{Sub}^{x}_{\mathtt{decl}}(Q) \wedge \mathsf{Cond}_{\mathtt{decl}}(x)\ \{\mathtt{var}\ x\}\ Q}$$

$$(adecl)\quad \frac{}{\mathsf{Sub}^{x,n}_{\mathtt{adecl}}(Q) \wedge \mathsf{Cond}_{\mathtt{decl}}(x,n)\ \{\mathtt{var}\, x\mathtt{[}n\mathtt{]}\}\ Q}$$

$$(assign)\quad \frac{}{\mathsf{Sub}^{x,e}_{\mathtt{assign}}(Q) \wedge \mathsf{Cond}_{\mathtt{assign}}(x,e)\ \{x\ \mathtt{:=}\ e\}\ Q}$$

$$(update)\quad \frac{}{\mathsf{Sub}^{x,e_1,e_2}_{\mathtt{update}}(Q) \wedge \mathsf{Cond}_{\mathtt{update}}(x,e_1,e_2)\ \{x\mathtt{[}e_1\mathtt{]}\ \mathtt{:=}\ e_2\}\ Q}$$

$$(if)\quad \frac{b \wedge P\ \{c\}\ Q \quad \neg b \wedge P \Rightarrow Q}{\mathsf{Sub}^{b}_{\mathtt{if}}(P) \wedge \mathsf{Cond}_{\mathtt{if}}(b)\ \{\mathtt{if}\, b\, \mathtt{then}\, c\}\ Q}$$

$$(while)\quad \frac{b \wedge P\ \{c\}\ P}{\mathsf{Sub}^{b}_{\mathtt{while}}(P) \wedge \mathsf{Cond}_{\mathtt{while}}(b)\ \{\mathtt{while}\, b\, \mathtt{do}\, c\}\ Q}$$

(with the rules (skip) and (cons) as before). ∎

5 Related Work

A number of different techniques have been applied to program certification. The following list is certainly not exhaustive; we focus on static techniques and leave out dynamic techniques like runtime monitoring [5].

Certification tools based on static analysis are already commercially available, e.g., PolySpace [14], which uses abstract interpretation and constraint solving techniques to identify possible runtime errors. However, such tools usually have fixed built-in notions of safety and suffer from a high number of false positives.

Other approaches use expressive type systems to enforce safety policies. Rittri [16] and Kennedy [7] have extended type inference techniques to ensure the consistent use of physical dimensions in functional programs. However, both approaches exploit certain algebraic properties of dimensions and it is unclear how general they are. Xi and Pfenning [20] have used dependent types to show array bounds safety, again for functional programs. Using similar ideas, Walker [18] has developed a type system to express and enforce a number of security policies. Shankar et al. [17] have used type qualifiers [3] to detect vulnerabilities due to C's format strings. Their `tainted` and `untainted` qualifiers take the same role as the values in our shadow domains. In general, type-based approaches tend to scale better, although it is unclear when a specific expressive type inference

algorithm becomes intractable in practice. Unlike the shadow variables, however, inferred types are static, i.e., the abstract value associated with a program cannot change during execution. Moreover, structured collections like arrays are usually modeled using a single type to keep inference tractable; this makes the analysis necessarily less precise. Experiments are thus required to compare the effects and trade-offs of the different approaches in practice.

Traditionally, program verification concentrates on showing full functional equivalence between specifications and programs. This is true especially for integrated development/proof environments as for example the KIV system [15]. However, Hoare-style verification has also been used in property-oriented certification as we investigate it here. Extended static checking (ESC) [8, 4] can be thought of as an "inference-based debugger": it uses Hoare rules, supported by program annotations, to detect a variety of potential errors, including division-by-zero and array-bounds violations. The more annotations the program contains, the more errors ESC can detect. Similarly, the SPARK Examiner [1] is a tool which uses Hoare rules to show exception freedom of Ada programs; this corresponds to a safety policy which combines more elaborate versions of operator safety (i.e., division-by-zero and overflow) and memory safety (i.e., array-bounds violations and overflow).[8] However, none of the systems deal with the question of correctness of their respective safety policies. Also, they typically only deal with one specific policy, whereas our framework is general.

6 Conclusions and Future Work

In this paper we have formalized a selection of safety properties using Hoare logic, and shown that they are sound and complete with respect to a semantic notion of safety. We have developed a generic method of doing this for arbitrary safety properties, thus showing how a safety policy can be automatically derived from a safety property and an operational semantics. The principal difficulty has been finding a general definition of safety property which enables this automatic derivation.

The rules we have presented show that safety rules can be quite complicated, even when dealing with a single policy at a time. The semantic framework developed in this paper serves as a structuring mechanism to deal with such complexity. The modularization of safety policies is a difficult problem but the present theory should serve as a starting point.

We are currently using this theory as the basis for the implementation of a VCG which is parametric with respect to a safety policy, and we are looking at a wide range of safety properties. Direct application of the theory should lead to a modular implementation.

The simple *while*-language studied here is sufficient for this because our aim is to certify synthesized code, and so we can control the language subset under

[8] Note that overflows can result from arithmetic operations as well as from inconsistent use of derived types (i.e., subtypes) and thus influence both operator safety and memory safety.

consideration. Moreover, since we can generate loop invariants along with the synthesized code our safety logic need not be concerned with this.

On a theoretical side, we believe that the logical nature of Definition 5 points to some interesting connections to the theory of computation, and we are currently investigating this.

References

[1] P. Amey and R. Chapman. "Industrial Strength Exception Freedom". In: *2002 SIGAda Intl. Conf. on Ada*, pp. 1–9. ACM, 2002.

[2] A. Appel. "Foundational proof-carrying code". In: *LICS-16*, pp. 247–258. IEEE, 2001.

[3] J. S. Foster, M. Fähndrich, and A. Aiken. "A Theory of Type Qualifiers". In: *PLDI*, pp. 192–203. ACM, 1999.

[4] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. "Extended static checking for Java". In: *PLDI*, pp. 234–245. ACM, 2002.

[5] K. Havelund and G. Roşu. "Monitoring Java Programs with Java PathExplorer". In: *First Workshop on Runtime Verification*, *ENTCS* **55(2)**. Elsevier, 2001.

[6] N. A. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. "A Syntactic Approach to Foundational Proof-Carrying Code". In: *LICS-17*, pp. 89–100. IEEE, 2002.

[7] A. Kennedy. *Programming Languages and Dimensions*. PhD thesis, University of Cambridge, April 1996.

[8] K. R. M. Leino and G. Nelson. "An extended static checker for Modula-3". In: *7th Intl. Conf. Compiler Construction*, *LNCS* **1383**, pp. 302–305. Springer, 1998.

[9] M. Lowry, T. Pressburger, and G. Rosu. "Certifying Domain-Specific Policies". In: *16th Intl. Conf. Automated Software Engineering*, pp. 118–125. IEEE, 2001.

[10] C. League, Z. Shao, and V. Trifonov. "Precision in Practice: A Type-Preserving Java Compiler". In: *12th Intl. Conf. Compiler Construction*, *LNCS* **2622**, pp. 106–120. Springer, April 2003.

[11] J. C. Mitchell. *Foundations for Programming Languages*. The MIT Press, 1996.

[12] G. C. Necula and P. Lee. "The Design and Implementation of a Certifying Compiler". In: *PLDI*, pp. 333–344. ACM, 1998.

[13] G. C. Necula and R. R. Schneck. "A Gradual Approach to a More Trustworthy, yet Scalable, Proof-Carrying Code". In: *CADE-18*, *LNCS* **2392**, pp. 47–62. Springer, 2002.

[14] PolySpace Technologies, 2002. `http://www.polyspace.com`.

[15] W. Reif. "The KIV Approach to Software Verification". In: *KORSO: Methods, Languages and Tools for the Construction of Correct Software*, *LNCS* **1009**, pp. 339–370. Springer, 1995.

[16] M. Rittri. "Dimension Inference Under Polymorphic Recursion". In: *7th Conf. Functional Prog. Languages Computer Architecture*, pp. 147–159. ACM, 1995.

[17] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. "Detecting Format String Vulnerabilities with Type Qualifiers". In: *10th Usenix Security Symposium*, 2001.

[18] D. Walker. "A Type System for Expressive Security Policies". In: *POPL-27*, pp. 254–267. ACM, 2000.

[19] M. Whalen, J. Schumann, and B. Fischer. "Synthesizing Certified Code". In: *FME*, *LNCS* **2391**, pp. 431–450. Springer, 2002.

[20] H. Xi and F. Pfenning. "Eliminating Array Bound Checking Through Dependent Types". In: *PLDI*, pp. 249–257. ACM, 1998.