Gradual Program Verification

Johannes Bader[1], Jonathan Aldrich[2], and Éric Tanter[3]

[1] Microsoft Corporation, Redmond, USA,
jobader@microsoft.com
[2] Institute for Software Research, Carnegie Mellon University, Pittsburgh, USA
jonathan.aldrich@cs.cmu.edu
[3] PLEIAD Lab, Computer Science Dept (DCC), University of Chile, Santiago, Chile
etanter@dcc.uchile.cl

Abstract. Both static and dynamic program verification approaches have significant disadvantages when considered in isolation. Inspired by research on gradual typing, we propose *gradual verification* to seamlessly and flexibly combine static and dynamic verification. Drawing on general principles from abstract interpretation, and in particular on the recent Abstracting Gradual Typing methodology of Garcia *et al.*, we systematically derive a gradual verification system from a static one. This approach yields, by construction, a gradual verification system that is compatible with the original static system, but overcomes its rigidity by resorting to dynamic verification when desired. As with gradual typing, the programmer can control the trade-off between static and dynamic checking by tuning the (im)precision of pre- and postconditions. The formal semantics of the gradual verification system and the proofs of its properties, including the gradual guarantees of Siek *et al.*, have been fully mechanized in the Coq proof assistant.

1 Introduction

Program verification techniques have the potential to improve the correctness of programs, by exploiting pre- and postconditions specified in formulas drawn from a given logic, such as Hoare logic [8]. Unfortunately, traditional approaches to verification have a number of shortcomings, as illustrated next.

Example 1.
```
int withdraw(int balance, int amount)
    requires (balance ≥ amount) ensures (result ≥ 0) {
    return balance - amount; // returns the new balance
}

int balance := 100;
balance := withdraw(balance, 30);
balance := withdraw(balance, 40);
```

In this case, we reason about a variable `balance` representing some bank account. The contract (pre- and postconditions) of `withdraw` specifies that it may only be called if the balance is high enough to withdraw the given amount, ensuring that no negative balance is reached. There are a number of ways to verify Example 1. We briefly discuss static and dynamic verification, including hybrid approaches. We then introduce *gradual verification* as an approach that has the potential to overcome a number of their shortcomings.

Static verification. Formal methods like Hoare logic are used to establish *statically* that a program is *valid*, *i.e.* satisfies its specification. In Example 1, the static verifier proves both that `withdraw` itself complies with its contract and that the three statements below are valid, *e.g.* that the precondition of `withdraw` is satisfied prior to both calls.

A lack of detailed contracts may prevent the verifier from establishing that a program is valid. In Example 1, verification of the second call to `withdraw` in fact fails: after the first call, the verifier knows from the postcondition that (`balance` $\geq$ `0`), which is insufficient to justify that (`balance` $\geq$ `40`) as required for the second call. Deriving such knowledge would require a stronger postcondition such as `result = old(balance) - amount`. However, this is not the postcondition that was provided by the programmer, perhaps intentionally (*e.g.* if the intent was to focus on some weaker correctness properties) or perhaps due to limited expressiveness of the underlying logic (notation such as `old`(x) may not exist). In general, a verification tool might also fail to prove program properties due to undecidability of the underlying logic or practical limitations of the specific tool implementation.

Hoare logic has been extended to more powerful logics like separation logic [15] and implicit dynamic frames [20]. Yet, the requirement of rigorous annotation of contracts remains an issue in these settings. Due to space limitations and to capture the core ideas of gradual verification, this paper focuses on a simple Hoare logic. We have formalized an extension to implicit dynamic frames and implemented a prototype, which can both be found at http://olydis.github.io/GradVer/impl/HTML5wp/

Dynamic verification. An alternative approach is to use *dynamic* verification to ensure that a program adheres to its specification at runtime, by turning the contract into *runtime checks*. A contract violation causes a runtime exception to be thrown, effectively preventing the program from entering a state that contradicts its specification. In Example 1, a dynamic verification approach would not raise any error because the `balance` is in fact sufficient for both calls to succeed. Note that because contracts are checked at runtime, one can even use arbitrary programs as contracts, and not just formulas drawn from a logic [6].

Meyer's Design by Contract methodology [12] integrated writing contracts in this way as an integral part of the design process, with the Eiffel language automatically performing dynamic contract verification [11]. Dynamic verification has also notably been used to check JML specifications [3], and has been extended to the case of separation logic by Nguyen *et al.* [14]. Note that unlike the static approach, the dynamic approach only requires programmers to encode the properties they care about as pre- and postconditions, and does not require extra work for the sake of avoiding false negatives. However, the additional checks impose runtime overhead that may not always be acceptable. Furthermore, violations of the specification are no longer detected ahead of time.

Hybrid approaches. Recognizing that static and dynamic checking have complementary advantages, some approaches to combine them have emerged. In particular, with the Java Modeling Language (JML) [2] and Code Contracts [5], it is possible to use the same specifications for either static or dynamic verification. Additionally, Nguyen *et al.* explored a hybrid approach to reduce the overhead of their approach to runtime checking for separation logic, by exploiting static information [14].

Although useful, these techniques do not support a smooth continuum between static and dynamic verification. With the JML approach, engineers enable static *or* dynamic verification; the two checking regimes do not interact. Nguyen *et al.* use the static checker to optimize runtime checks, but do not try to report static verification failures because it is difficult to distinguish failures due to contradictions in the specification (which the developer should be warned about) from failures due to leaving out parts of the specification (which could have been intentional underspecification, and thus should not produce a warning). Their runtime checking approach also requires the specification of heap footprints to match in pre- and post-conditions, which like many static checking approaches forces programmers to do extra specification work to avoid false negatives.

Gradual verification. Because this tension between static and dynamic verification is reminiscent of the tension between static and dynamic type checking, we propose to draw on research on *gradual typing* [18,17,7] to develop a flexible approach to program verification, called *gradual verification.* Gradual typing supports both static and dynamic checking and the entire spectrum in between, driven by the precision of programmer annotations [19]. Similarly, gradual verification introduces a notion of *imprecise contracts*, supporting a continuum between static and dynamic verification. A static checker can analyze a gradually-specified program and warn the programmer of inconsistencies between specifications and code, including contracts that are intended to be fully precise but are not strong enough, as well as contracts that contradict one another despite possible imprecision in each. On the other hand, the static checker will not produce warnings that arise from a contract that is intentionally imprecise; in these cases, runtime checking is used instead. Programmers can rely on a *gradual guarantee* stating that reducing the precision of specifications never breaks the verifiability (and reduceability) of a program. This guarantee, originally formulated by Siek *et al.* in the context of gradual types [19], ensures that programmers can choose their desired level of precision without artificial constraints imposed by the verification technology.

It is worth noting that the similarly named work “The Gradual Verifier” [1] focuses on *measuring the progress of static verification.* Their verification technique “GraVy” is neither sound nor complete and does not comply with the gradual guarantee.

Gradual verification is not only useful in cases of missing information (*e.g.* when reusing a library that is not annotated) but also to overcome limitations of the static verification system as motivated by Example 1. Furthermore, programmers can gradually evolve and refine static annotations. As they do so, they are rewarded by progressively *increased static correctness guarantees* and progressively *decreased runtime checking*, supporting a pay-as-you-go cost model.

Specifically, we support imprecision by introducing an unknown formula `?` that acts like a wildcard during static verification. Semantically, the static checker will optimistically accept a formula containing `?` as long as there exists some interpretation of `?` that makes the formula valid. As we learn more information about the program state at runtime, the dynamic checker ensures that some valid instantiation still exists. Crucially, the unknown formula can be combined with static formulas, forming *imprecise* formulas. For instance, going back to Example 1, we can write the imprecise postcondition $(\texttt{result} \geq 0) \wedge \texttt{?}$ in order to enable gradual reasoning, resulting in an optimistic interpretation of `?` as $(\texttt{result} \geq 40)$ when statically proving the precondition of the second call. At runtime, this interpretation is checked, to ensure soundness.

Note that the postcondition we suggest is only *partially unknown*, preserving the *static knowledge* $(\texttt{result} \geq 0)$. This not only allows us to prove certain goals (*e.g.* $(\texttt{result} \neq \texttt{-10})$) without requiring any dynamic checks, but also to statically reject programs that provably contradict this knowledge (*e.g.* if a subsequent call had `balance = -10` as precondition).

Contributions. This paper is the first development of the ideas of gradual typing in the context of program logics for verification. More precisely, we first introduce a simple statically-verified language called SVL, along with its associated program logic. We then adapt the *Abstracting Gradual Typing* methodology (AGT) [7] to the verification setting and show in section 3 how the static semantics of a gradually-verified language GVL can be derived from SVL using principles of abstract interpretation. Section 4 develops GVL's dynamic semantics. Here, we deviate from the AGT approach and instead propose injecting a minimal amount of runtime assertion checks, yielding a pay-as-you-go cost model. Finally, Section 5 briefly discusses GVL_{IDF}, an extension of our approach to heap-allocated objects and an extended logic with implicit dynamic frames [20].

The formal semantics of GVL and the proofs of its properties have been fully mechanized in the Coq proof assistant and can be found at `http://olydis.github.io/GradVer/impl/HTML5wp/`. The site also includes a report with the formal treatment of the extended logic, as well as an interactive online prototype of GVL_{IDF}. Due to limited space, some figures contain only selected parts of definitions. Complete definitions can be found online as well.

Limitations. Our approach for dynamic semantics requires assertions to be evaluable at runtime, naturally limiting the logic usable for annotations. The AGT methodology is not restricted that way, so it may be a good starting point for a dynamic semantics in presence of assertions drawn from a higher-order logic.

$$
\begin{array}{rl@{\qquad}rl}
\mathit{program} ::= & \overline{\mathit{procedure}}\ s & s \in \textsc{Stmt} ::= & \texttt{skip} \mid s_1\texttt{;}\ s_2 \mid T\ x \mid x\ \texttt{:=}\ e \\
\mathit{procedure} ::= & T\ m(\overline{T\ x})\ \mathit{contract}\ \texttt{\{}\ s\ \texttt{\}} & & \mid x\ \texttt{:=}\ m(x) \mid \texttt{assert}\ \phi \\
\mathit{contract} ::= & \texttt{requires}\ \phi\ \texttt{ensures}\ \phi & e \in \textsc{Expr} ::= & v \mid x \mid (e\ \oplus\ e) \\
T ::= & \texttt{int} & x \in \textsc{Var} ::= & \texttt{result} \mid \mathit{ident} \mid \texttt{old}(\mathit{ident}) \\
\oplus ::= & \texttt{+} \mid \texttt{-} \mid \ldots & v \in \textsc{Val} ::= & n \\
\odot ::= & \texttt{=} \mid \neq \mid \texttt{<} \mid \ldots & \phi \in \textsc{Formula} ::= & \texttt{true} \mid (e\ \odot\ e) \mid \phi\ \land\ \phi
\end{array}
$$

$$\text{and syntactic sugar}\quad \texttt{return}\ e \overset{\text{def}}{=} \texttt{result :=}\ e \quad\text{and}\quad T\ x\ \texttt{:=}\ e \overset{\text{def}}{=} T\ x\texttt{;}\ x\ \texttt{:=}\ e$$

Fig. 1. SVL: Syntax

However, it is also more abstract, with no obvious implementation strategy or approach to estimate runtime overhead.

2 SVL: Statically Verified Language

In the following sections, we describe a simple statically verified language called SVL. We formalize its syntax, semantics and soundness.

2.1 Syntax

Figure 1 shows the syntax of SVL. Programs consist of a collection of procedures and a designated statement resembling the entry point ("main procedure"). We include the empty statement, statement sequences, variable declarations, variable assignments, procedure calls, and assertions. All statements are in A-normal form, which is not essential semantically but does simplify the formalism. Procedures have contracts consisting of a pre- and postcondition, which are formulas. Formulas can be the constant `true`, binary relations between expressions, and a conjunction $\land$. Expressions can occur within a formula or variable assignment, and consist of variables, constants and arithmetic operations. [4]

For the remainder of this work we only consider well-formed programs: variables are declared before use, procedure names are unique and contracts only contain variables that are in scope. More specifically, a precondition may only contain the procedure's parameters x_i, a postcondition may only contain the special variable `result` and the dummy variables $\texttt{old}(x_i)$.

To simplify the presentation of semantics, we will give rules for procedures that have exactly one parameter.

2.2 Dynamic Semantics

We now describe the dynamic semantics of SVL. SVL has a small-step semantics $\cdot \longrightarrow \cdot : \textsc{State} \rightharpoonup \textsc{State}$ (see Fig. 2) that describes discrete transitions between

[4] We chose a minimal syntax to keep presentation of our approach simple, yet it is directly applicable to further statements, richer type systems or formulas that are arbitrary boolean expressions which can be checked at runtime.

program states. Program states that are not in the domain of this partial function are said to be *stuck*, which happens if an assertion does not hold or a contract is violated before/after a call. In Section 2.3, we define a static verification system whose soundness result implies that valid SVL programs do not get stuck.

Program states. Program states consist of a stack, *i.e.* $\textsc{State} = \textsc{Stack}$ where:

$$S \in \textsc{Stack} ::= E \cdot S \mid \mathsf{nil} \qquad \text{where} \qquad E \in \textsc{StackFrame} = \textsc{Env} \times \textsc{Stmt}$$

A stack frame consists of a local variable environment $\rho \in \textsc{Env} = \textsc{Var} \rightharpoonup \textsc{Val}$ and a continuation statement.

Evaluation. An expression e is evaluated according to a big-step evaluation relation $\rho \vdash e \Downarrow v$, yielding value v using local variable environment $\rho \in \textsc{Env}$ of the top-most stack-frame. The definition is standard: variables are looked up in ρ, and the resulting values are combined according to standard arithmetic rules. Example: $[\mathtt{x} \mapsto 3] \vdash \mathtt{x + 5} \Downarrow 8$

The evaluation of a formula in a given environment is specified by the predicate $\cdot \vDash \cdot \subseteq \textsc{Env} \times \textsc{Formula}$. We assume standard evaluation semantics for standard concepts like equality. We also say that a formula *describes* a certain (infinite) set of environments (exactly the environments under which it holds), yielding natural definitions for formula satisfiability and implication.

Definition 1 (Denotational Formula Semantics).
Let $[\![\cdot]\!] : \textsc{Formula} \rightarrow \mathcal{P}(\textsc{Env})$ *be defined as* $[\![\phi]\!] \overset{\mathsf{def}}{=} \{\ \rho \in \textsc{Env} \mid \rho \vDash \phi\ \}$

Definition 2 (Formula Satisfiability). *A formula ϕ is satisfiable if and only if $[\![\phi]\!] \neq \emptyset$. Let* $\textsc{SatFormula} \subset \textsc{Formula}$ *be the set of satisfiable formulas.*

Definition 3 (Formula Implication). $\phi_1 \Rightarrow \phi_2$ *if and only if* $[\![\phi_1]\!] \subseteq [\![\phi_2]\!]$

Reduction rules. We define a standard small-step reduction semantics for statements (Fig. 2). SsAssert ensures that assertions are stuck if the asserted formula is not satisfied. SsCall sets up a new stack frame and makes sure that the procedure's precondition is satisfied by the newly set up context. Similarly, SsCallExit ensures that the postcondition is satisfied before returning control to the call site. Note our use of auxiliary functions $\mathsf{procedure}$ and mpost in order to retrieve a procedure's definition or postcondition using that procedure's name. Formally, we assume all rules and definitions are implicitly parameterized with the "executing" program $p \in \textsc{Program}$ from which to extract this information. When required for disambiguation, we explicitly annotate reduction arrows with the executing program p, as in $\longrightarrow_p$.

Note that SsCall also initializes $\mathtt{old}(x')$, which allows assertions and most importantly the postcondition to reference the parameter's initial value. In reality, no additional memory is required to maintain this value since it is readily available as $\rho(x)$, i.e. it lives in the stack frame of the call site. For a program to be well-formed, it may not write to $\mathtt{old}(x')$ in order to enable this reasoning.

$$\frac{\rho \vDash \phi}{\langle \rho, \texttt{assert}\ \ \phi\rangle \cdot S \longrightarrow \langle \rho, \texttt{skip}\rangle \cdot S}\ \textsc{SsAssert}$$

$$\frac{\rho \vdash e \Downarrow v \qquad \rho' = \rho[x \mapsto v]}{\langle \rho, x\ \texttt{:=}\ e\rangle \cdot S \longrightarrow \langle \rho', \texttt{skip}\rangle \cdot S}\ \textsc{SsAssign}$$

$$\frac{\begin{array}{c}\mathsf{procedure}(m) = T_r\ m\texttt{(}T\ x'\texttt{)}\ \texttt{requires}\ \phi_p\ \texttt{ensures}\ \phi_q\ \texttt{\{}\ r\ \texttt{\}} \\ \rho \vdash x \Downarrow v \qquad \rho' = [x' \mapsto v, \texttt{old(}x'\texttt{)} \mapsto v] \qquad \rho' \vDash \phi_p\end{array}}{\langle \rho, y\ \texttt{:=}\ m\texttt{(}x\texttt{)}\texttt{;}\ s\rangle \cdot S \longrightarrow \langle \rho', r\rangle \cdot \langle \rho, y\ \texttt{:=}\ m\texttt{(}x\texttt{)}\texttt{;}\ s\rangle \cdot S}\ \textsc{SsCall}$$

$$\frac{\mathsf{post}(m) = \phi_q \qquad \rho' \vDash \phi_q}{\langle \rho', \texttt{skip}\rangle \cdot \langle \rho, y\ \texttt{:=}\ m\texttt{(}x\texttt{)}\texttt{;}\ s\rangle \cdot S \longrightarrow \langle \rho[y \mapsto \rho'(\texttt{result})], s\rangle \cdot S}\ \textsc{SsCallExit}$$

Fig. 2. SVL: Small-step semantics (selected rules)

$$\begin{array}{llll}
\mathsf{WLP}(\texttt{skip}, \phi) & = \phi & \mathsf{WLP}(s_1\texttt{;}\ s_2, \phi) & = \mathsf{WLP}(s_1, \mathsf{WLP}(s_2, \phi)) \\
\mathsf{WLP}(x\ \texttt{:=}\ e, \phi) & = \phi[e/x] & \mathsf{WLP}(\texttt{assert}\ \ \phi_a, \phi) & = \phi_a\ \wedge\ \phi
\end{array}$$

$$\begin{aligned}
\mathsf{WLP}(y\ \texttt{:=}\ m\texttt{(}x\texttt{)}, \phi) = \max_{\Rightarrow}\ \{\ \phi' \mid\ & y \notin \mathsf{FV}(\phi')\ \wedge \\
& \phi' \Rightarrow \mathsf{mpre}(m)[x/\mathsf{mparam}(m)]\ \wedge \\
& (\phi'\ \wedge\ \mathsf{mpost}(m)[x, y/\texttt{old(}\mathsf{mparam}(m)\texttt{)}, \texttt{result}]) \Rightarrow \phi\ \}
\end{aligned}$$

Fig. 3. SVL: Weakest precondition (selected rules)

2.3 Static Verification

We define the static verification of SVL contracts through a weakest liberal precondition calculus [4]. This syntax-directed approach (compared to, say, Hoare logic, which has an existential in its sequence rule) will be useful for the dynamic semantics of our gradual language (will be pointed out again later).

Definition 4 (Valid Procedure).
A procedure with contract `requires` ϕ_p `ensures` ϕ_q*, parameter* x *and body* s *is considered valid if* $\phi_p \Rightarrow \mathsf{WLP}(s, \phi_q)[x/\texttt{old(}x\texttt{)}]$ *holds.*

We define $\mathsf{WLP} : \textsc{Stmt} \times \textsc{Formula} \rightharpoonup \textsc{Formula}$ as shown in Figure 3. WLP is standard for the most part. The rule for calls computes a maximal formula ϕ' (*i.e.* minimum information content) that is sufficient to imply both the procedure's precondition and the overall postcondition ϕ with the help of the procedure's postcondition.

Definition 5 (Valid Program). *A program with entry point statement* s *is considered valid if* `true` $\Rightarrow \mathsf{WLP}(s, \texttt{true})$ *holds and all procedures are valid.* [5]

[5] Note that one can demand more than `true` to hold at the final state by simply ending the program with an assertion statement.

$$\begin{aligned}
\mathsf{sWLP}(s \cdot \mathsf{nil}, \phi) &= \mathsf{WLP}(s, \phi) \cdot \mathsf{nil} \\
\mathsf{sWLP}(s \cdot (y \texttt{ := } m\texttt{(}x\texttt{)}\texttt{; } s') \cdot \overline{s}, \phi) &= \mathsf{WLP}(s, \mathsf{mpost}(m)) \cdot \mathsf{sWLP}((y \texttt{ := } m\texttt{(}x\texttt{)}\texttt{; } s') \cdot \overline{s}, \phi)
\end{aligned}$$

Fig. 4. Weakest precondition across call boundaries

Example 2 (Static Checker of SVL). We demonstrate the resulting behavior of SVL's static checker using example 1, but with varying contracts:

`requires (balance ≥ amount)`
`ensures (result = old(balance) - old(amount))`

`withdraw` is valid since the WLP of the body, given the postcondition, is `(balance - amount = old(balance) - old(amount))`. Substitution gives `(balance - amount = balance - amount)` which is trivially implied by the precondition. The overall program is also valid since the main procedure's WLP is `(100 ≥ 70)` which is implied by `true`.

`requires (balance ≥ amount) ensures (result ≥ 0)` (as in example 1)

`withdraw` is valid since the body's WLP is `(balance - amount ≥ 0)` which matches the precondition. However, the program is not valid: The WLP of the second call is `(balance ≥ 40)` which is not implied by the postcondition of the first call. As a result, the WLP of the entire main procedure is undefined.

`requires (balance ≥ 0) ensures (result ≥ 0)`

Validating `withdraw` fails since the body's WLP (same as above) is not implied by the precondition.

2.4 Soundness

Verified programs should respect contracts and assertions. We have formulated the runtime semantics of SVL such that they get stuck if contracts or assertions are violated. As a result, *soundness* means that valid programs do not get stuck. In particular, we can use a syntactic progress/preservation argument [22].

If the WLP of a program is satisfied by the current state, then execution will not get stuck (progress) and after each step of execution, the WLP of the remaining program is again satisfied by the new state (preservation). We use a progress and preservation formulation of soundness not just because it allows us to reason about non-terminating programs, but mainly because this will allow us to naturally deal with the runtime checking needs of gradual verification.

To simplify reasoning about states with multiple stack frames, we extend the definition of WLP to accept a stack of statements and return a stack of preconditions, as shown in Figure 4. Note that WLP as defined previously can only reason about procedure calls atomically since an element of STMT cannot encode intermediate states of an ongoing procedure call. In contrast sWLP works across call boundaries by accepting a stack of statements and recursively picking up the postconditions of procedures which are currently being executed.

While before we defined what makes procedures as a whole valid, we can now validate arbitrary intermediate program states, e.g. we can say that

$$\mathsf{sWLP}\left(\begin{array}{l}\texttt{return balance - amount}\\ \qquad\qquad\cdot \\ \quad\texttt{b2 := withdraw(b1, a)}\\ \qquad\qquad\cdot\\ \qquad\quad\mathsf{nil}\end{array}, \texttt{(b2} \neq \texttt{-1)} \land \texttt{(a = 4)}\right) = \begin{array}{c}\texttt{(balance - amount} \geq \texttt{0)}\\ \cdot\\ \texttt{(b1} \geq \texttt{a)} \land \texttt{(a = 4)}\\ \cdot\\ \mathsf{nil}\end{array}$$

where `withdraw` is defined as in example 1. If $\overline{s}$ are the continuations of some arbitrary program state $\pi \in \textsc{State}$, then $\mathsf{sWLP}(\overline{s}, \texttt{true})$ is the precondition for $\overline{s}$. If $\mathsf{sWLP}(\overline{s}, \texttt{true})$ holds in the variable environments $\overline{\rho}$ of π, respectively, then soundness guarantees that the program does not get stuck. In the following, we extend the notion of validity to arbitrary intermediate program states in order to formalize progress and preservation. Validity of states is an invariant that relates the static and dynamic semantics of valid SVL programs.

Definition 6 (Valid state). *We call the state* $\langle \rho_n, s_n \rangle \cdot ... \cdot \langle \rho_1, s_1 \rangle \cdot \mathsf{nil} \in \textsc{State}$ valid *if* $\rho_i \vDash \mathsf{sWLP}_i(s_n \cdot ... \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$ *for all* $1 \leq i \leq n$. *(*$\mathsf{sWLP}_i(\overline{s}, \phi)$ *is the i-th component of* $\mathsf{sWLP}(\overline{s}, \phi)$*)*

Validity of the initial program state follows from validity of the program (Def. 5).

Proposition 1 (SVL: Progress). *If* $\pi \in \textsc{State}$ *is a valid state and* $\pi \notin \{\langle \rho, \texttt{skip} \rangle \cdot \mathsf{nil} \mid \rho \in \textsc{Env}\}$ *then* $\pi \longrightarrow \pi'$ *for some* $\pi' \in \textsc{State}$.

Proposition 2 (SVL: Preservation). *If* π *is a valid state and* $\pi \longrightarrow \pi'$ *for some* $\pi' \in \textsc{State}$ *then* π' *is a valid state.*

3 GVL: Static Semantics

Having defined SVL, we can now derive its gradual counterpart GVL, which supports gradual program verification thanks to *imprecise* contracts. We follow the abstract interpretation perspective on gradual typing [7], AGT for short. In this sense, we introduce *gradual formulas* as formulas that can include the *unknown formula*, denoted `?`:

$\widetilde{\phi} ::= \phi \mid \phi \land \texttt{?}$ and standalone formula `?` as syntactic sugar for $\texttt{true} \land \texttt{?}$

We define $\widetilde{\textsc{Formula}}$ as the set of all gradual formulas. The syntax of GVL is unchanged save for the use of gradual formulas in contracts: *contract* $::=$ $\texttt{requires}\ \widetilde{\phi}\ \texttt{ensures}\ \widetilde{\phi}$. In Sections 3.2 to 3.4 we *lift* the predicates and functions SVL uses for verification from the static domain to the gradual domain, yielding a gradual verification logic for GVL.

3.1 Interpretation of Gradual Formulas

We call ϕ in $\phi \land \texttt{?}$ *static part* of the imprecise formula and define a helper function $\mathsf{static} : \widetilde{\textsc{Formula}} \rightarrow \textsc{Formula}$ that extracts the static part of a gradual formula, i.e. $\mathsf{static}(\phi) = \phi$ and $\mathsf{static}(\phi \land \texttt{?}) = \phi$. Following the AGT approach, a gradual formula is given meaning by concretization to *the set of static formulas* that it represents.

Definition 7 (Concretization of gradual formulas).
$\gamma : \widetilde{\text{FORMULA}} \rightharpoonup \mathcal{P}^{\text{FORMULA}}$ *is defined as:*

$$\begin{array}{ll}\gamma(\phi) = \{\ \phi\ \} & \\ \gamma(\phi\ \wedge\ ?) = \{\ \phi' \in \text{SATFORMULA} \mid \phi' \Rightarrow \phi\ \} & \textit{if } \phi \in \text{SATFORMULA} \\ \gamma(\phi\ \wedge\ ?) \quad \textit{undefined otherwise} & \end{array}$$

A fully-precise formula concretizes to the singleton set. Importantly, we only concretize imprecise formulas to precise formulas that are *satisfiable.* Note that the concretization of any gradual formula always implies the static part of that formula. The notion of *precision* between formulas, reminiscent of the notion of precision between gradual types [19], is naturally induced by concretization [7]:

Definition 8 (Precision). *$\widetilde{\phi}_1$ is more precise than $\widetilde{\phi}_2$, written $\widetilde{\phi}_1 \sqsubseteq \widetilde{\phi}_2$ if and only if $\gamma(\widetilde{\phi}_1) \subseteq \gamma(\widetilde{\phi}_2)$.*

3.2 Lifting Predicates

The semantics of SVL makes use of predicates that operate on formulas, namely formula implication and formula evaluation. As GVL must operate on gradual formulas, these predicates are lifted in order to deal with gradual formulas in a consistent way. We propose the following definitions of consistent formula evaluation and implication.

Definition 9 (Consistent Formula Evaluation).
Let $\cdot \mathrel{\widetilde{\vDash}} \cdot \subseteq \text{ENV} \times \widetilde{\text{FORMULA}}$ *be defined as* $\rho \mathrel{\widetilde{\vDash}} \widetilde{\phi} \iff \rho \vDash \mathsf{static}(\widetilde{\phi})$

Definition 10 (Consistent Formula Implication).
Let $\cdot \mathrel{\widetilde{\Rightarrow}} \cdot \subseteq \widetilde{\text{FORMULA}} \times \widetilde{\text{FORMULA}}$ *be defined inductively as*

$$\frac{\phi_1 \Rightarrow \mathsf{static}(\widetilde{\phi}_2)}{\phi_1 \mathrel{\widetilde{\Rightarrow}} \widetilde{\phi}_2}\ \widetilde{\text{IMPLSTATIC}} \qquad \frac{\begin{array}{c}\phi \in \text{SATFORMULA} \\ \phi \Rightarrow \phi_1 \qquad \phi \Rightarrow \mathsf{static}(\widetilde{\phi}_2)\end{array}}{\phi_1\ \wedge\ ? \mathrel{\widetilde{\Rightarrow}} \widetilde{\phi}_2}\ \widetilde{\text{IMPLGRAD}}$$

In rule $\widetilde{\text{IMPLGRAD}}$, ϕ represents a *plausible* formula represented by $\phi_1\ \wedge\ ?$.

Abstract interpretation. Garcia *et al.* [7] define consistent liftings of predicates as their *existential* liftings:

Definition 11 (Consistent Predicate Lifting). *The consistent lifting $\widetilde{P} \subseteq \widetilde{\text{FORMULA}} \times \widetilde{\text{FORMULA}}$ of a predicate $P \subseteq \text{FORMULA} \times \text{FORMULA}$ is defined as:*

$$\widetilde{P}(\widetilde{\phi}_1, \widetilde{\phi}_2) \overset{\mathsf{def}}{\iff} \exists \phi_1 \in \gamma(\widetilde{\phi}_1), \phi_2 \in \gamma(\widetilde{\phi}_2).\ P(\phi_1, \phi_2)$$

Our definitions above are proper predicate liftings.

Lemma 1 (Consistent Formula Evaluation and Implication).
$\cdot \mathrel{\widetilde{\vDash}} \cdot$ (Def. 9) is a consistent lifting of $\cdot \vDash \cdot$ and $\cdot \mathrel{\widetilde{\Rightarrow}} \cdot$ (Def. 10) is a consistent lifting of $\cdot \Rightarrow \cdot$.

3.3 Lifting Functions

Deriving gradual semantics from SVL also involves lifting *functions* that operate on formulas, most importantly WLP (Definition 3). Figure 5 gives the definition of $\widetilde{\mathsf{WLP}} : \textsc{Stmt} \times \widetilde{\textsc{Formula}} \rightharpoonup \widetilde{\textsc{Formula}}$, the consistent lifting of WLP. For

$$\widetilde{\mathsf{WLP}}(\mathtt{skip}, \widetilde{\phi}) = \widetilde{\phi} \qquad \widetilde{\mathsf{WLP}}(s_1\,\mathtt{;}\ s_2, \widetilde{\phi}) = \widetilde{\mathsf{WLP}}(s_1, \widetilde{\mathsf{WLP}}(s_2, \widetilde{\phi}))$$

$$\widetilde{\mathsf{WLP}}(x\ \mathtt{:=}\ e, \widetilde{\phi}) = \widetilde{\phi}[e/x] \qquad \widetilde{\mathsf{WLP}}(\mathtt{assert}\ \phi_a, \widetilde{\phi}) = \phi_a \ \wedge\ \widetilde{\phi}$$

$$\widetilde{\mathsf{WLP}}(y\ \mathtt{:=}\ m\mathtt{(}x\mathtt{)}, \widetilde{\phi}) = \begin{cases} \phi' & \text{if } \widetilde{\phi}, \mathsf{mpre}(m), \mathsf{mpost}(m) \in \textsc{Formula} \\ \phi' \ \wedge\ \mathtt{?} & \text{otherwise} \end{cases}$$

$$\text{where } \phi' = \underset{\Rightarrow}{\max}\ \{\ \phi'' \mid y \notin \mathsf{FV}(\phi'')\ \wedge\ (\phi'' \mathrel{\widetilde{\Rightarrow}} \mathsf{mpre}(m)[x/\mathsf{mparam}(m)]) \ \wedge$$
$$(\phi''\ \wedge\ \mathsf{mpost}(m)[x, y/\mathtt{old}\mathtt{(}\mathsf{mparam}(m)\mathtt{)}, \mathtt{result}]) \mathrel{\widetilde{\Rightarrow}} \widetilde{\phi}\ \}$$

Fig. 5. GVL: Weakest precondition (selected rules)

most statements $\widetilde{\mathsf{WLP}}$ is defined almost identical to WLP, however, calls are more complex. Note that for calls, $\widetilde{\mathsf{WLP}}$ not only has to deal with the fact that $\widetilde{\phi}$ is a gradual formula, but also that procedure m may now have imprecise contracts. In a sense, the function is lifted w.r.t. three formula parameters, two of them referenced through the procedure's name. To accomplish this, we first determine the static part ϕ' of the result which is analogous to the WLP, but resorting to lifted predicates. Next, we determine whether it would be sufficient to return ϕ' unmodified, or whether it is plausible that the precondition must be stronger. If all three influencing formulas are precise $\widetilde{\mathsf{WLP}}$ should coincide with WLP, so ϕ' is returned. Otherwise, ϕ' might have been chosen too weak, which is counteracted by making it imprecise.

Abstract interpretation. Again, AGT [7] formalizes the notion of consistent functions using an *abstraction* function α that maps a set of static formulas back to the most precise gradual formula that represents this set, such that $\langle \gamma, \alpha \rangle$ forms a Galois connection.

Definition 12 (Abstraction of formulas). *Let* $\alpha : \mathcal{P}^{\textsc{SatFormula}} \rightharpoonup \widetilde{\textsc{Formula}}$ *be defined as* $\alpha(\overline{\phi}) = \underset{\sqsubseteq}{\min}\ \{\ \widetilde{\phi} \in \widetilde{\textsc{Formula}} \mid \overline{\phi} \subseteq \gamma(\widetilde{\phi})\ \}$

α is partial since $\underset{\sqsubseteq}{\min}$ does not necessarily exist, *e.g.* $\alpha(\{\mathtt{(x} \neq \mathtt{x)}, \mathtt{(x = x)}\})$ is undefined. Using concretization for gradual parameters and abstraction for return values one can consistently lift (partial) functions:

Definition 13 (Consistent Function Lifting). *Given a partial function* $f : \textsc{Formula} \rightharpoonup \textsc{Formula}$*, its consistent lifting* $\widetilde{f} : \widetilde{\textsc{Formula}} \rightharpoonup \widetilde{\textsc{Formula}}$ *is defined as* $\widetilde{f}(\widetilde{\phi}) = \alpha(\{\ f(\phi) \mid \phi \in \gamma(\widetilde{\phi})\ \})$

Lemma 2 (Consistent WLP). $\widetilde{\mathsf{WLP}}$ *is a consistent lifting of* WLP.

3.4 Lifting the Verification Judgment

Gradual verification in GVL must deal with imprecise contracts. The static verifier of SVL uses WLP and implication to determine whether contracts and the overall program are valid (Def. 4, 5). Because contracts in GVL may be imprecise, we have to resort to $\widetilde{\mathsf{WLP}}$ (Fig. 5) and consistent implication (Def. 10).

Example 3 (Static Checker of GVL). Static semantics of SVL and GVL coincide for precise contracts, so example 2 applies to GVL without modification. We extend the example with imprecise contracts:

`requires (balance` $\geq$ `amount) ensures (result` $\geq$ `0)` $\land$ `?`

Note the similarity to the precise contract in example 1 which causes GVL's static checker to reject the main procedure. However, with the imprecise postcondition we now have `(balance` $\geq$ `0)` $\land$ `?` $\widetilde{\Rightarrow}$ `(balance` $\geq$ `40)`.
As a result, the static checker optimistically accepts the program. At the same time, it is not *guaranteed* that the precondition is satisfied at runtime without additional checks. We expect GVL's runtime semantics (Section 4) to add such checks as appropriate. These runtime checks should succeed for the main procedure of example 1, however they should fail if we modify the main program as follows, withdrawing more money than available:

```
int b := 100; b := withdraw(b, 30); b := withdraw(b, 80);
```

Static checking succeeds since `(b` $\geq$ `0)` $\land$ `?` $\widetilde{\Rightarrow}$ `(b` $\geq$ `80)`, but `b`'s value at runtime will not satisfy the formula. Note that the presence of imprecision does not necessarily imply success of static checking:

```
int b := 100; b := withdraw(b, 30); assert (b < 0);
```

It is *implausible* that this program is valid since `(b` $\geq$ `0)` $\land$ `?` $\widetilde{\Rightarrow}$ `(b < 0)` does not hold. However, further weakening `withdraw`'s postcondition to `?` would again result in static acceptance but dynamic rejection.

`requires ? ensures (result = old(balance) - old(amount))` $\land$ `?`

This contract demonstrates that imprecision must not necessarily result in runtime checks. The body's $\widetilde{\mathsf{WLP}}$ is `?`, which is implied by the annotated precondition `?` without having to be optimistic (i.e. resort to the plausibility interpretation). We expect that an efficient runtime semantics, like the one we discuss in Section 4.3, adds no runtime overhead through checks here.

4 GVL: Dynamic Semantics

Accepting a gradually-verified program means that it is *plausible* that the program remains valid during each step of its execution, precisely as it is guaranteed by soundness of SVL. To prevent a GVL program from stepping from a valid state into an invalid state, we extend the dynamic semantics of SVL with (desirably minimal) runtime checks. As soon as the validity of the execution is no longer plausible, these checks cause the program to step into a dedicated error state. Example 3 motivates this behavior.

Soundness. We revise the soundness definition of SVL to the gradual setting which will guide the upcoming efforts to achieve soundness via runtime assertion checks. Validity of states (Def. 6) relies on sWLP (Fig. 4) which itself consumes postconditions of procedures. Hence, GVL uses a consistent lifting of sWLP which we define analogous to Fig. 4, but based on $\widetilde{\mathsf{WLP}}$. Save for using $\mathsf{s}\widetilde{\mathsf{WLP}}$ instead of sWLP, valid states of GVL are defined just like those of SVL.

We expect there to be error derivations $\pi \mathrel{\widetilde{\longrightarrow}} \textbf{error}$ whenever it becomes implausible that the remaining program can be run safely. Note that we do not extend $\textsc{State}$, but instead define $\cdot \mathrel{\widetilde{\longrightarrow}} \cdot \subseteq \textsc{State} \times (\textsc{State} \cup \{\textbf{error}\})$. As a result, we can leave Prop. 2 (Preservation) untouched.

In Section 4.1 we derive a naive runtime semantics driven by the soundness criteria of GVL. We then examine the properties of the resulting gradually verified language. In Section 4.3 we discuss optimizing this approach by combining $\widetilde{\mathsf{WLP}}$ with strongest preconditions $\widetilde{\mathsf{SP}}$ in order to determine statically-guaranteed information that can be used to minimize the runtime checks ahead of time.

4.1 Naive semantics

We start with a trivially correct but expensive strategy of adding runtime assertions to each execution step, checking whether the new state would be valid (preservation), right before actually transitioning into that state (progress). [6]

Let $\rho'_{1..m}, \rho_{1..n} \in \textsc{Env},\ s'_{1..m}, s_{1..n} \in \textsc{Stmt}$

If $\langle \rho'_m, s'_m \rangle \cdot \ldots \cdot \langle \rho'_1, s'_1 \rangle \cdot \mathsf{nil} \longrightarrow \langle \rho_n, s_n \rangle \cdot \ldots \cdot \langle \rho_1, s_1 \rangle \cdot \mathsf{nil}$ holds[7], then

$$\langle \rho'_m, s'_m \rangle \cdot \ldots \cdot \langle \rho'_1, s'_1 \rangle \cdot \mathsf{nil} \mathrel{\widetilde{\longrightarrow}} \begin{cases} \langle \rho_n, s_n \rangle \cdot \ldots \cdot \langle \rho_1, s_1 \rangle \cdot \mathsf{nil} & \text{if } (\rho_n \mathrel{\widetilde{\vDash}} \widetilde{\phi}_n) \land \ldots \land (\rho_1 \mathrel{\widetilde{\vDash}} \widetilde{\phi}_1) \\ \quad \text{where } \widetilde{\phi}_n \cdot \ldots \cdot \widetilde{\phi}_1 \cdot \mathsf{nil} = \mathsf{s}\widetilde{\mathsf{WLP}}(s_n \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \mathtt{true}) & \\ \textbf{error} & \text{otherwise} \end{cases}$$

Before showing how to implement the above semantics, we confirm its soundness: Progress of GVL follows from progress of SVL. The same is true for preservation: in the first reduction case, validity of the resulting state follows from preservation of SVL.

[6] Note the difference between runtime assertions and the `assert` statement. The former checks assertions at runtime, transitioning into a dedicated exceptional state on failure. The latter is a construct of a statically verified language, and is hence implementable as a no-operation.

[7] $\textsc{SsCall}$ and $\textsc{SsCallExit}$ as defined in Fig. 2 are not defined for gradual formulas. Thus, we adjust those rules to use consistent evaluation $\mathrel{\widetilde{\vDash}}$ instead of $\vDash$. Since $\mathrel{\widetilde{\vDash}}$ coincides with $\vDash$ for precise formulas, this is a conservative extension of SVL.

While we can draw the implementation of $\cdot \widetilde{\longrightarrow} \cdot$ from SVL, implementing the case condition $(\rho_n \mathrel{\widetilde{\vDash}} \widetilde{\phi}_n) \wedge ... \wedge (\rho_1 \mathrel{\widetilde{\vDash}} \widetilde{\phi}_1)$ results in overhead. As a first step, we can heavily simplify this check using inductive properties of our language: Stack frames besides the topmost one are not changed by a single reduction, *i.e.* $\rho_{n-1}, ..., \rho_1, s_{n-1}, ..., s_1$ stay untouched. It follows that $\widetilde{\phi}_i$ for $1 \leq i < n$ remains unchanged since changes in s_n do not affect lower components of $\widetilde{\mathsf{sWLP}}$ (see Fig. 4). As a result, it is sufficient to check $\rho_n \mathrel{\widetilde{\vDash}} \widetilde{\mathsf{sWLP}}_n(s_n \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$.

Recall how we argued that a weakest precondition approach is more suited for the dynamic semantics of GVL than Hoare logic (section 2.3). Due to the syntax-directed sequence rule, all potentially occurring $\widetilde{\mathsf{sWLP}}_n$ are partial results of statically *precomputed* preconditions. Contrast this with a consistent sequence rule of Hoare logic: If both premises of the rule are lifted independently, $\{?\}$`skip; skip`$\{?\}$ could be derived statically, say, by instantiating the existential with `(x = 3)`. However, the partial result $\{$`(x = 3)`$\}$`skip`$\{?\}$ has no (guaranteed) relationship with the next program state since the existential was chosen too strong. Any attempt to fix the gradual sequence rule by imposing additional restrictions on the existential must necessarily involve a weakest precondition calculus, applied to the suffix of the sequence.

4.2 Properties of GVL

Before discussing practical aspects of GVL, we turn to its formal properties: GVL is a *sound*, *gradual* language. The following three properties are formalized and proven in Coq.

Soundness. Our notion of soundness for GVL coincides with that of SVL, save for the possibility of runtime errors. Indeed, it is up to the dynamic semantics of GVL to make up for the imprecisions that weaken the statics of GVL.

Lemma 3 (Soundness of GVL). *GVL is sound:*

Progress *If* $\pi \in \textsc{State}$ *is a valid state and* $\pi \notin \{\langle \rho, \mathtt{skip}\rangle \cdot \mathsf{nil} \mid \rho \in \textsc{Env}\}$ *then* $\pi \widetilde{\longrightarrow} \pi'$ *for some* $\pi' \in \textsc{State}$ *or* $\pi \widetilde{\longrightarrow}$ **error***.*

Preservation *If* π *is a valid state and* $\pi \widetilde{\longrightarrow} \pi'$ *for some* $\pi' \in \textsc{State}$ *then* π' *is a valid state.*

We call the state $\langle \rho_n, s_n\rangle \cdot ... \cdot \langle \rho_1, s_1\rangle \cdot \mathsf{nil}$ valid *if* $\rho_i \mathrel{\widetilde{\vDash}} \widetilde{\mathsf{sWLP}}_i(s_n \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$ *for all* $1 \leq i \leq n$*.*

Conservative extension. GVL is a conservative extension of SVL, meaning that both languages coincide on fully-precise programs. This property is true *by construction.* In fact, the definition of concretization and consistent lifting captures this property, which thus percolates to the entire verification logic. In order for the dynamic semantics to be a conservative extension, GVL must progress whenever SVL does, yielding the same continuation. This is the case since the reduction rules of GVL coincide with those of SVL for fully-precise annotations (the runtime checks succeed due to preservation of SVL, so we do not step to **error**).

Gradual guarantees. Siek *et al.* formalize a number of criteria for gradually-typed languages [19], which we can adapt to the setting of program verification. In particular, the *gradual guarantee* captures the smooth continuum between static and dynamic verification. More precisely, it states that typeability (here, verifiability) and reducibility are *monotone* with respect to precision. We say a program p_1 is more precise than program p_2 ($p_1 \sqsubseteq p_2$) if p_1 and p_2 are equivalent except in terms of contracts and if p_1's contracts are more precise than p_2's contracts. A contract `requires` ϕ_p^1 `ensures` ϕ_q^1 is more precise than contract `requires` ϕ_p^2 `ensures` ϕ_q^2 if $\phi_p^1 \sqsubseteq \phi_p^2$ and $\phi_q^1 \sqsubseteq \phi_q^2$.

In particular, the static gradual guarantee for verification states that a valid gradual program is still valid when we reduce the precision of contracts.

Proposition 3 (Static gradual guarantee of verification).
Let $p_1, p_2 \in \textsc{Program}$ such that $p_1 \sqsubseteq p_2$. If p_1 is valid then p_2 is valid.

The dynamic gradual guarantee states that a valid program that takes a step still takes the same step if we reduce the precision of contracts.

Proposition 4 (Dynamic gradual guarantee of verification).
Let $p_1, p_2 \in \textsc{Program}$ such that $p_1 \sqsubseteq p_2$ and $\pi \in \textsc{State}$.
If $\pi \mathrel{\widetilde{\longrightarrow}_{p_1}} \pi'$ for some $\pi' \in \textsc{State}$ then $\pi \mathrel{\widetilde{\longrightarrow}_{p_2}} \pi'$.

This also means that if a gradual program fails at runtime, then making its contracts more precise will *not* eliminate the error. In fact, doing so may only make the error manifest *earlier* during runtime or even manifest *statically.* This is a fundamental property of gradual verification: a runtime verification error reveals a fundamental mismatch between the gradual program and the underlying verification discipline.

4.3 Practical aspects

Residual checks. Compared to SVL, the naive semantics adds a runtime assertion to every single reduction. Assuming that the cost of checking an assertion is proportional to the formula size, *i.e.* proportional to the size of the WLP of the remaining statement, this is highly unsatisfying. The situation is even worse if the entire GVL program has fully-precise annotations, because then the checks are performed even though they are not necessary for safety.

We can reduce formula sizes given static information, expecting formulas to vanish (reduce to `true`) in the presence of fully-precise contracts and gradually grow with the amount of imprecision introduced, yielding a pay-as-you-go cost model. Example 4 illustrates this relationship.

Example 4 (Residual checks). Consider the following variation of `withdraw`:

```
int withdraw(int balance, int amount)
      requires ? ensures result ≥ 0 {
   // WLP: (balance - amount ≥ 0) ∧ (amount > 0)
   assert amount > 0;
```

```
    // WLP: balance - amount ≥ 0
    balance = balance - amount;
    // WLP: balance ≥ 0
    return balance;
    // WLP: result ≥ 0
}
```

Precomputed preconditions are annotated. Following the naive semantics (Section 4.1), these are to be checked *before* entering the corresponding state, to ensure soundness. However, many of these checks are redundant. When entering the procedure (*i.e.* stepping to the state prior to the assertion statement), we must first check $\phi_1 =$ `(balance - amount ≥ 0) ∧ (amount > 0)`. According to the naive semantics, in order to execute the assertion statement, we would then check $\phi_2 =$ `(balance - amount ≥ 0)`. Fortunately, it is derivable statically that this check must definitely succeed: The strongest postcondition of `assert amount > 0` given ϕ_1 is again ϕ_1. Since $\phi_1 \Rightarrow \phi_2$ there is no point in checking ϕ_2 at runtime. Similar reasoning applies to both remaining statements, making all remaining checks redundant. Only the initial check remains, which is itself directly dependent on the imprecision of the precondition. Preconditions `(balance - amount ≥ 0) ∧ ?` or `(amount > 0) ∧ ?` would allow dropping the corresponding term of the formula, the conjunction of both (with or without a `?`) allows dropping the entire check.

The above example motivates the formalization of a strongest postcondition function $\widetilde{\mathsf{SP}}$ and function $\widetilde{\mathsf{diff}}$ which takes a formula $\widetilde{\phi}_a$ to check, a formula $\widetilde{\phi}_k$ known to be true and calculates a residual formula $\widetilde{\mathsf{diff}}(\widetilde{\phi}_a, \widetilde{\phi}_k)$ sufficient to check in order to guarantee that $\widetilde{\phi}_a$ holds.

Definition 14 (Strongest postcondition). *Let* $\mathsf{SP} : \textsc{Stmt} \times \textsc{Formula} \rightharpoonup \textsc{Formula}$ *be defined as:* $\mathsf{SP}(s, \phi) = \min_{\Rightarrow} \{ \phi' \in \textsc{Formula} \mid \phi \Rightarrow \mathsf{WLP}(s, \phi') \}$
Furthermore let $\widetilde{\mathsf{SP}} : \textsc{Stmt} \times \widetilde{\textsc{Formula}} \rightharpoonup \widetilde{\textsc{Formula}}$ *be the consistent lifting (Def. 13) of* SP.

Definition 15 (Reducing formulas).
Let $\mathsf{diff} : \textsc{Formula} \times \textsc{Formula} \rightharpoonup \textsc{Formula}$ *be defined as:*

$$\mathsf{diff}(\phi_a, \phi_k) = \max_{\Rightarrow} \{ \phi \in \textsc{Formula} \mid (\phi \wedge \phi_k \Rightarrow \phi_a) \wedge (\phi \wedge \phi_k \in \textsc{SatFormula}) \}$$

Furthermore let $\widetilde{\mathsf{diff}} : \widetilde{\textsc{Formula}} \times \widetilde{\textsc{Formula}} \rightharpoonup \widetilde{\textsc{Formula}}$ *be defined as:*

$$\widetilde{\mathsf{diff}}(\phi_a, \widetilde{\phi}_k) = \mathsf{diff}(\phi_a, \mathsf{static}(\widetilde{\phi}_k)) \qquad \widetilde{\mathsf{diff}}(\phi_a \wedge ?, \widetilde{\phi}_k) = \mathsf{diff}(\phi_a, \mathsf{static}(\widetilde{\phi}_k)) \wedge ?$$

Both SP and diff can be implemented approximately by only approximating min/max. Likewise, an implementation of $\widetilde{\mathsf{SP}}$ may err towards imprecision. As a result, the residual formulas may be larger than necessary. [8]

[8] Even a worst case implementation of $\widetilde{\mathsf{SP}}(s, \widetilde{\phi})$ as ? will only result in no reduction of the checks, but not affect soundness of the verification logic.

$$\frac{\langle \rho'_n, (s;\ s_n)\rangle \cdot \ldots \longrightarrow \langle \rho_n, s_n\rangle \cdot \ldots}{\langle \rho'_n, (s;\ s_n)\rangle \cdot \ldots \ \widetilde{\longrightarrow}\ \langle \rho_n, s_n\rangle \cdot \ldots}\ \widetilde{\text{SsLocal}}$$

$$\frac{\begin{array}{c}\langle \rho_{n-1}, (y := m(x);\ s_{n-1})\rangle \cdot \ldots \longrightarrow \langle \rho_n, \mathsf{mbody}(m)\rangle \cdot \ldots \\ \rho_n \mathrel{\widetilde{\models}} \widetilde{\mathsf{diff}}(\widetilde{\mathsf{WLP}}(\mathsf{mbody}(m), \mathsf{mpost}(m)), \mathsf{mpre}(m))\end{array}}{\langle \rho_{n-1}, (y := m(x);\ s_{n-1})\rangle \cdot \ldots \ \widetilde{\longrightarrow}\ \langle \rho_n, \mathsf{mbody}(m)\rangle \cdot \ldots}\ \widetilde{\text{SsCall}}$$

$$\frac{\begin{array}{l}\quad\langle \rho'_{n+1}, \mathtt{skip}\rangle \cdot \langle \rho'_n, (y := m(x);\ s_n)\rangle \cdot \ldots \longrightarrow \langle \rho_n, s_n\rangle \cdot \ldots \\ \rho_n \mathrel{\widetilde{\models}} \widetilde{\mathsf{diff}}(\widetilde{\mathsf{sWLP}}_n(s_n \cdot \ldots, \mathtt{true}), \\ \qquad\qquad \widetilde{\mathsf{SP}}(y := m(x), \widetilde{\mathsf{sWLP}}_n((y := m(x);\ s_n) \cdot \ldots, \mathtt{true})))\end{array}}{\langle \rho'_{n+1}, \mathtt{skip}\rangle \cdot \langle \rho'_n, (y := m(x);\ s_n)\rangle \cdot \ldots \ \widetilde{\longrightarrow}\ \langle \rho_n, s_n\rangle \cdot \ldots}\ \widetilde{\text{SsCallExit}}$$

Fig. 6. Dynamic semantics with reduced checks.

Definition 16 (Approximate algorithm for $\widetilde{\mathsf{diff}}$).

```
$\widetilde{\text{FORMULA}}$ $\widetilde{\mathsf{diff}}$ ($\widetilde{\text{FORMULA}}$ $\widetilde{\phi}_a$, $\widetilde{\text{FORMULA}}$ $\widetilde{\phi}_b$)
  let $\widetilde{\phi}_1 \wedge \widetilde{\phi}_2 \wedge \ldots \wedge \widetilde{\phi}_n$ := $\widetilde{\phi}_a$ (such that all $\widetilde{\phi}_i$ are atomic)
  for i from 1 to n
    if $[\![\widetilde{\phi}_b]\!] \cap [\![\widetilde{\phi}_1 \wedge \ldots \wedge \widetilde{\phi}_{i-1} \wedge \widetilde{\phi}_{i+1} \ldots \wedge \widetilde{\phi}_n]\!] \subseteq [\![\widetilde{\phi}_a]\!]$
      $\widetilde{\phi}_i$ := true
  return $\widetilde{\phi}_1 \wedge \ldots \wedge \widetilde{\phi}_n$        // one may drop the true terms
```

Figure 6 shows the dynamic semantics using residual checks (omitting error reductions). Runtime checks of reductions operating on a single stack frame vanish completely as there exists no source of imprecision in that subset of GVL. This property can be formalized as: *For all $s \in$ Stmt that do not contain a call,* $\widetilde{\mathsf{diff}}(\widetilde{\mathsf{sWLP}}_n(s_n \cdot \ldots, \mathtt{true}), \widetilde{\mathsf{SP}}(s, \widetilde{\mathsf{sWLP}}_n((s;\ s_n) \cdot \ldots, \mathtt{true}))) \in \{\mathtt{true}, ?\}$ The check in $\widetilde{\text{SsCall}}$ vanishes if $\mathsf{mpre}(m)$ is precise. The check in $\widetilde{\text{SsCallExit}}$ vanishes if $\mathsf{mpost}(m)$ is precise. Recall that Example 4 demonstrates this effect: We concluded that only the initial check is necessary and derived that it also vanishes if the precondition is precise.

If $\mathsf{mpost}(m)$ is imprecise, the assertion is equivalent to

$$\rho_n \mathrel{\widetilde{\models}} \widetilde{\mathsf{diff}}(\widetilde{\mathsf{diff}}(\widetilde{\mathsf{sWLP}}_n(s_n \cdot \ldots, \mathtt{true}), \mathsf{mpre}(m)[x/\mathsf{mparam}(m)]), \\ \mathsf{mpost}(m)[x, y/\mathtt{old}(\mathsf{mparam}(m)), \mathtt{result}])$$

which exemplifies the pay-as-you-go relationship between level of imprecision and run-time overhead: both $\mathsf{mpre}(m)$ and $\mathsf{mpost}(m)$ contribute to the reduction of $\widetilde{\mathsf{sWLP}}_n(s_n \cdot \ldots, \mathtt{true})$, *i.e.* increasing their precision results in smaller residual checks.

Pay-as-you-go. To formally establish the pay-as-you-go characteristic of gradual verification, we introduce a simple cost model for runtime contract checking.

Let $\mathsf{size}(\widetilde{\phi})$ be the number of conjunctive terms of (the static part of) $\widetilde{\phi}$, e.g. $\mathsf{size}($`(x = 3)` $\wedge$ `(y` $\neq$ `z)` $\wedge$ `?`$) = 2$. We assume that measure to be proportional to the time needed to evaluate a given formula. Let $\mathsf{checksize}(p)$ be the sum of the size of all residual checks in program p.

Proposition 5 (Pay-as-you-go overhead).
a) *Given programs* p_1, p_2 *such that* $p_1 \sqsubseteq p_2$*, then* $\mathsf{checksize}(p_1) \leq \mathsf{checksize}(p_2)$.
b) *If a program* p *has only* precise *contracts, then* $\mathsf{checksize}(p) = 0$.

5 Scaling up to Implicit Dynamic Frames

We applied our approach to a richer program logic, namely *implicit dynamic frames* (IDF) [20], which enables reasoning about shared mutable state. The starting point is an extended statically verified language similar to Chalice [10], called SVL_{IDF}. Compared to SVL, the language includes classes with publicly-accessible fields and instance methods. We add field assignments and object creation. Formulas may also contain field *accessibility predicates* `acc` from IDF and use the separating conjunction `*` instead of regular (non-separating) conjunction $\wedge$. An accessibility predicate `acc(`e`.`f`)` denotes exclusive access to the field e`.`f. It justifies accessing e`.`f both in the source code (*e.g.* `x.f := 3` or `y := x.f`) and in the formula itself (*e.g.* `acc(x.f) * (x.f` $\neq$ `4)`), which is called *framing.* Compared to SVL, the main challenge in gradualizing SVL_{IDF} is framing.

The linear logic ensures that accessibility predicates cannot be duplicated, hence entitling them to represent *exclusive* access to a field. SVL_{IDF} can *statically* prove that any field access during execution is justified. To formalize and prove soundness, SVL_{IDF} has a reference dynamic semantics that tracks, for each stack frame, the set of fields that it has exclusive access to; deciding at runtime whether a formula holds depends on this information. Of course, thanks to soundness, an implementation of SVL_{IDF} needs no runtime tracking.

Recall that our approach to derive a gradual language includes the insertion of runtime checks, which requires that formulas *can* be evaluated at runtime. Therefore, the overhead of the reference semantics of SVL_{IDF} carries over to a naive implementation semantics. Additionally, it is no longer clear how accessibility is split between stack frames in case of a call: SVL_{IDF} transfers exclusive access to fields that are mentioned in the precondition of a procedure from the call site to the callee's (fresh) stack frame. As we allow `?` to also plausibly represent accessibility predicates, an imprecise precondition poses a challenge.

A valid strategy is to conservatively forward *all* accesses from caller to callee. As for GVL, we can devise an efficient implementation strategy for accessibility tracking that results in pay-as-you-go overhead. The fact that reducing the precision of contracts may now result in a divergence of program states (specifically, the accessible fields) asks for an adjustment of the dynamic part of the gradual guarantee, which originally requires lock-step reduction behavior. We carefully adjust the definition, preserving the core idea that reducing precision of a valid program does not alter the *observable* behavior of that program. The formalization of SVL_{IDF} and GVL_{IDF} are available in a companion report available online, along with an interactive prototype implementation at `http://olydis.github.io/GradVer/impl/HTML5wp/`.

The prototype implementation of GVL_{IDF} displays the static and dynamic semantics of code snippets interactively, indicating the location of inserted runtime checks where necessary. It also displays the stack and heap at any point of execution. A number of predefined examples are available, along with an editable scratchpad. In particular, Example 2 demonstrates how imprecision enables safe reasoning about a recursive data structure that was not possible in SVL_{IDF}, because SVL_{IDF} lacks recursive predicates. This illustrates that, similarly to gradual types, imprecision can be used to augment the expressiveness of the static verification logic. In this case, the example does not even require a single runtime check.

6 Related Work

We have already related our work to the most-closely related research, including work on the underlying logics [8,4,15,20], the theory of gradual typing [18,17,19,7], closely related approaches to static [10] and dynamic [6,12,11,3] verification, as well as hybrid verification approaches [2,14]. Additional related work includes gradual type systems that include notions of *ownership* or *linearity*; one can think of the `acc` predicate as representing ownership of a piece of the heap, and `acc` predicates are treated linearly in the implicit dynamic frames methodology [20]. [21] developed a gradual typestate checking system, in which the state of objects is tracked in a linear type system. Similar to `acc` predicates, permissions to objects with state are passed linearly from one function to another, without being duplicated; if a strong permission is lost (*e.g.* due to a gradual specification), it can be regained with a runtime check, as long as no conflicting permission exist.

The gradual ownership approach of [16] is also related in that, like implicit dynamic frames, it aids developers in reasoning (gradually) about heap data structures. In that work, developers can specify containment relationships between objects, such that an *owned* object cannot be accessed from outside its owner. These access constraints can be checked either statically using a standard ownership type system, but if the developer leaves out ownership annotations from part of the program, dynamic checks are inserted.

Typestate and ownership are finite-state and topological properties, respectively, whereas in this work we explore gradual specification of logical contracts for the first time. Neither of these prior efforts benefited from the Abstracting Gradual Typing (AGT) methodology [7], which guided more principled design choices in our present work. Additionally, it is unclear whether the gradual guarantee of Siek *et al.* [19] holds in these proposals, which were developed prior to the formulation of the gradual guarantee.

One contrasting effort, which was also a stepping-stone to our current paper, is recent work on gradual refinement types [9]. In that approach, the AGT methodology is applied to a functional language in which types can be refined by logical predicates drawn from a decidable logic. The present work is in a different context, namely first-order imperative programs as opposed to higher-order pure functional programs. This difference has a strong impact on the technical development. The present work is simpler in one respect, because formulas do not depend on their lexical context as in the gradual refinement setting. However, we had to reformulate gradual verification based on a weakest precondition calculus, whereas the prior work could simply extend the type system setting used when the AGT methodology was proposed. In addition, we provide a runtime semantics directly designed for the gradual verification setting, rather than adapting the evidence-tracking approach set forth by the AGT methodology and used for gradual refinement types. In fact, we investigated using the evidence-based approach for the runtime semantics of gradual verification, and found that it was semantically equivalent to what we present here but introduces unnecessary complexities. Overall, by adapting the AGT methodology to the verification setting, this work shows that the abstract interpretation approach to designing gradual languages can scale beyond type systems.

7 Conclusion

We have developed the formal foundations of gradual program verification, taking as starting point a simple program logic. This work is the first adaptation of recent fundamental techniques for gradual typing to the context of program verification. We have shown how to exploit the Abstracting Gradual Typing methodology [7] to systematically derive a gradual version of a weakest precondition calculus. Gradual verification provides a continuum between static and dynamic verification techniques, controlled by the (im)precision of program annotations; soundness is mediated through runtime checks.

Later, we briefly discuss how we applied our strategy to a more advanced logic using implicit dynamic frames (IDF) [20] in order to reason about mutable state. The use of IDF presents an additional challenge for obtaining a full pay-as-you-go model for gradual verification, because the footprint has to be tracked. We point to our prototype implementation which also references a formalization of graualizing SVL_{IDF}. Further optimization of this runtime tracking is an interesting direction of future work. Another interesting challenge is to exercise our approach with other, richer program logics, as well as to study the gradualization of type systems that embed logical information, such as Hoare Type Theory [13], establishing a bridge between this work and gradual refinement types [9].

References

1. Arlt, S., Rubio-González, C., Rümmer, P., Schäf, M., Shankar, N.: The gradual verifier. In: NASA Formal Methods Symposium. pp. 313–327. Springer (2014)
2. Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of jml tools and applications. International Journal on Software Tools for Technology Transfer 7(3), 212–232 (2005), `http://dx.doi.org/10.1007/s10009-004-0167-4`
3. Cheon, Y., Leavens, G.T.: A runtime assertion checker for the java modeling language (jml) (2002)
4. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM 18(8), 453–457 (Aug 1975), `http://doi.acm.org/10.1145/360933.360975`
5. Fahndrich, M., Barnett, M., Logozzo, F.: Embedded contract languages. In: ACM SAC - OOPS. Association for Computing Machinery, Inc. (March 2010), `https://www.microsoft.com/en-us/research/publication/embedded-contract-languages/`
6. Findler, R.B., Felleisen, M.: Contracts for higher-order functions. In: Proceedings of the 7th ACM SIGPLAN Conference on Functional Programming (ICFP 2002). pp. 48–59. Pittsburgh, PA, USA (Sep 2002)
7. Garcia, R., Clark, A.M., Tanter, E.: Abstracting gradual typing. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 429–442. POPL '16, ACM, New York, NY, USA (2016), `http://doi.acm.org/10.1145/2837614.2837670`
8. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12(10), 576–580 (1969)
9. Lehmann, N., Tanter, É.: Gradual refinement types. In: Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2017). pp. 775–788. Paris, France (Jan 2017)
10. Leino, K.R.M., Müller, P., Smans, J.: Verification of concurrent programs with chalice. In: Foundations of Security Analysis and Design V, pp. 195–222. Springer (2009)
11. Meyer, B.: Eiffel: A language and environment for software engineering. Journal of Systems and Software 8(3), 199–246 (1988)
12. Meyer, B.: Object-Oriented Software Construction. Prentice Hall (1988)
13. Nanevski, A., Morrisset, G., Birkedal, L.: Hoare type theory, polymorphism and separation. Journal of Functional Programming 5-6, 865–911 (2008)
14. Nguyen, H.H., Kuncak, V., Chin, W.N.: Runtime checking for separation logic. In: International Workshop on Verification, Model Checking, and Abstract Interpretation. pp. 203–217. Springer (2008)
15. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on. pp. 55–74. IEEE (2002)
16. Sergey, I., Clarke, D.: Gradual ownership types. In: Proceedings of the 21st European Conference on Programming Languages and Systems. pp. 579–599. ESOP'12, Springer-Verlag, Berlin, Heidelberg (2012), `http://dx.doi.org/10.1007/978-3-642-28869-2_29`
17. Siek, J., Taha, W.: Gradual typing for objects. In: European Conference on Object-Oriented Programming. pp. 2–27. Springer (2007)

18. Siek, J.G., Taha, W.: Gradual typing for functional languages. In: Scheme and Functional Programming Workshop. vol. 6, pp. 81–92 (2006)
19. Siek, J.G., Vitousek, M.M., Cimini, M., Boyland, J.T.: Refined criteria for gradual typing. In: LIPIcs-Leibniz International Proceedings in Informatics. vol. 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2015)
20. Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames: Combining dynamic frames and separation logic. In: European Conference on Object-Oriented Programming. pp. 148–172. Springer (2009)
21. Wolff, R., Garcia, R., Tanter, É., Aldrich, J.: Gradual typestate. In: European Conference on Object-Oriented Programming. pp. 459–483. Springer (2011)
22. Wright, A., Felleisen, M.: A syntactic approach to type soundness. Inf. Comput. 115(1), 38–94 (Nov 1994), http://dx.doi.org/10.1006/inco.1994.1093