Gradual Program Verification with Implicit Dynamic Frames

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. We demonstrate the generality of our gradual verification approach by applying it to *implicit dynamic frames*, which enables reasoning about shared mutable state.

1 Introduction

We translate our motivating example to a language similar to Chalice, that uses *implicit dynamic frames*[18] (IDF) which enables safe reasoning about mutable state.

Example 1.
```
class Account {
    int balance;
    void withdraw(int amount)
            requires acc(this.balance) * (this.balance ≥ amount)
            ensures  acc(this.balance) * (this.balance ≥ 0) {
        int newBalance := this.balance - amount;
        this.balance := newBalance;
    }
}
a := new Account(); a.balance := 100; a.withdraw(30); a.withdraw(40);
```

In this case, we reason about the field `balance` of some bank account object `a`. The contract of `withdraw` is a direct formalization of a specification stating that `withdraw` may only be called if the balance is high enough to withdraw the given amount of coins, ensuring that no negative balance is reached. The formula `acc(this.balance)` is an accessibility predicate. The predicate denotes *exclusive* access to the `balance` field, so the contract states in the `requires` clause that this access is demanded in order to call the `withdraw` method, and is afterwards returned to the call site, as specified in the `ensures` clause. An additional guarantee of the verification logic is that `a` is not and does not become a `null` pointer throughout execution.

Note that, in formulas, "`*`" denotes the *separating conjunction* of separation logic, which ensures that its operands specify access to distinct fields, *i.e.* heap locations [13,18]. Specifically, accessibility predicates "`acc`" and the separating conjunction allow local reasoning about calls through *framing*:

Example 2 (Framing). Assume one is verifying a call `a.withdraw(30)`, knowing that the call site satisfies

$$\phi = \texttt{acc(a.balance)} * \texttt{(a.balance = 100)} * \texttt{acc(b.balance)} * \texttt{(b.balance = 100)}$$

The call to `withdraw` is allowed since the precondition (instantiated for `a`)

$$\phi_p = \texttt{acc(a.balance)} * \texttt{(a.balance} \geq \texttt{30)}$$

is implied by ϕ. To reason about a call, it is important to know which information remains *unaffected* by the call (the *frame problem* [18]). In this case, the separating conjunction guarantees that `b.balance` is not an alias of `a.balance`, so one can safely assume that `b.balance` remains unchanged during the call. Specifically, it allows the verifier to *frame off* `acc(b.balance) * (b.balance = 100)` from ϕ, since it is guaranteed to still hold after the call. Note that without the separating conjunction it would not be possible to frame off this knowledge and there would be no way of knowing whether `b.balance` is modified by the call without global knowledge. This is key for separation logic to support modular reasoning about heap manipulating programs [13].

Contributions. To demonstrate the flexibility of our gradualization approach, we extend SVL to support heap-allocated objects and equip it with a richer logic based on implicit dynamic frames (Section 2). This extension, called SVL_{IDF}, poses several challenges for gradualization, addressed in GVL_{IDF}, a gradually-verified language with implicit dynamic frames. We present the statics of GVL_{IDF} in Section 3 and its dynamics in Section 4.

We have developed a prototype implementation[4] of GVL_{IDF}. For brevity, some figures contain only selected parts of definitions. Complete definitions can be found in the appendix.

2 SVL_{IDF}: Implicit Dynamic Frames

We demonstrate the generality of our approach to gradual program verification by considering a more expressive language, with a more expressive logic. The starting point is a language similar to Chalice [8], called SVL_{IDF}. Specifically, SVL_{IDF} has classes with fields and instance methods, and uses implicit dynamic frames (IDF) [18] to safely reason about mutable state. In this section, we formally introduce SVL_{IDF} by pointing out the key differences from SVL. The following sections will show how to apply our gradualization methodology to derive the gradually-verified counterpart of SVL_{IDF}, named GVL_{IDF}. As we

[4] http://olydis.github.io/GradVer/impl/HTML5wp/

will see, implicit dynamic frames impose a number of interesting challenges for gradualization.

We next introduce the syntax of SVL$_{\text{IDF}}$ compared to that of SVL, present the extended runtime semantics containing heaps and permission masks (Sect. 2.2) and its static verification logic together with the soundness criterion that connects both (Sect. 2.3).

2.1 Syntax

$$\begin{array}{rl}
\mathit{program} ::= & \overline{\mathit{cls}}\ s \\
\mathit{cls} ::= & \texttt{class}\ C\ \texttt{\{}\ \overline{\mathit{field}}\ \overline{\mathit{method}}\ \texttt{\}} \\
\mathit{field} ::= & T\ \ f\texttt{;} \\
T ::= & ...\mid C \\
\phi ::= & \texttt{true} \mid (e\ \odot\ e) \mid \phi * \phi \mid \texttt{acc}(e.f)
\end{array}
\qquad
\begin{array}{rl}
s ::= & ...\mid x.f\ \texttt{:=}\ y \mid x\ \texttt{:= new}\ C \mid y\ \texttt{:=}\ z.m(x) \\
e ::= & ...\mid e.f \\
x ::= & ...\mid \texttt{this} \\
v ::= & ...\mid o \mid \texttt{null}
\end{array}$$

where $o \in \textsc{Loc}$ is a heap location
and $f \in \textsc{FieldName}$ are field names

Fig. 1. SVL$_{\text{IDF}}$: Syntax ("..." indicates parts of the syntax that coincide with SVL)

Figure 1 presents the syntax of SVL$_{\text{IDF}}$. Compared to SVL, the language includes classes with publicly-accessible fields and instance methods. We add field assignments and object creation. As programs contain no more global functions, the method call statement was modified to call instance methods. Note that instance methods and their calls can be thought of as syntactic sugar on top of global functions: The methods can be thought of as taking an additional parameter "`this`" and having an additional conjunctive term $(\texttt{this} \neq \texttt{null})$ in their precondition. (Note that SVL$_{\text{IDF}}$ knows no inheritance so there is also no dynamic dispatch, which enables the above interpretation.)

Formulas may now also contain the field *accessibility predicates* `acc` from implicit dynamic frames, and use the separating conjunction $*$ instead of regular (non-separating) conjunction $\wedge$. Expressions may contain field reads which must be justified with an appropriate accessibility predicate.

Implicit dynamic frames. Formulas in SVL$_{\text{IDF}}$ include *accessibility predicates*. An accessibility predicate $\texttt{acc}(e.f)$ 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.* $\texttt{acc(x.f)} * (\texttt{x.f} \neq \texttt{4})$).

The *static footprint* $A \in \textsc{StatFprint} = \mathcal{P}(\textsc{Expr} \times \textsc{FieldName})$ of a formula ϕ, denoted $\lfloor \phi \rfloor$, is the set of fields it requires access to:

$$\lfloor \texttt{acc}(e.f) \rfloor = \{\langle e, f \rangle\} \qquad \lfloor \phi_1 * \phi_2 \rfloor = \lfloor \phi_1 \rfloor \cup \lfloor \phi_2 \rfloor \qquad \lfloor \phi \rfloor = \emptyset \quad \text{otherwise}$$

A formula is said to be *framed* by a footprint A if all its expressions only mention

$$\frac{\langle e, f\rangle \in A \qquad A \vdash_{\texttt{frm}} e}{A \vdash_{\texttt{frm}} e.f}\ \textsc{FField} \qquad\qquad \frac{A \vdash_{\texttt{frm}} e_1 \qquad A \vdash_{\texttt{frm}} e_2}{A \vdash_{\texttt{frm}} e_1 \odot e_2}\ \textsc{SFComp}$$

$$\frac{A \vdash_{\texttt{frm}} e}{A \vdash_{\texttt{frm}} \texttt{acc}(e.f)}\ \textsc{SFAcc} \qquad\qquad \frac{A \vdash_{\texttt{frm}} \phi_1 \qquad A \cup \lfloor \phi_1 \rfloor \vdash_{\texttt{frm}} \phi_2}{A \vdash_{\texttt{frm}} \phi_1 * \phi_2}\ \textsc{SFSepOp}$$

Fig. 2. SVL$_{\textsc{IDF}}$: Framing (selected rules)

fields in A (Fig. 2). Note how the SFSepOp rule augments the footprint used to check the right sub-formula, using the footprint of the left sub-formula. This means that access predicates within a formula are able to frame expressions in the same formula, if mentioned in the right order. Formulas that are framed by an empty footprint are called *self-framed* and define the set of self-framed formulas $\textsc{SfrmFormula} \overset{\text{def}}{=} \{\ \phi \in \textsc{Formula} \mid \emptyset \vdash_{\texttt{frm}} \phi\ \}$. For example, the formula `(x.f = 1)` is not self-framed while `acc(x.f) * (x.f = 1)` is. We write $\widehat{\phi}$ to denote self-framed formulas. We require method contracts to use self-framed formulas, so we can refine their definition to *contract* ::= `requires` $\widehat{\phi}$ `ensures` $\widehat{\phi}$ in order to encode this restriction syntactically.

2.2 Dynamic Semantics

We now describe the dynamic semantics of SVL$_{\textsc{IDF}}$. In addition to SVL the semantics is stuck whenever a statement accesses a field that the current context does not have access to. For this purpose, the runtime tracks the set of accessible fields, called the *dynamic footprint* $A \in \textsc{DynFprint} = \mathcal{P}(\textsc{Loc} \times \textsc{FieldName})$. This footprint is expanded when new objects are created, and is split into disjoint caller- and callee-specific sets when methods are called. Soundness implies that valid SVL$_{\textsc{IDF}}$ programs do not get stuck so SVL$_{\textsc{IDF}}$ can be executed by a runtime system that does *not* track accessible fields.

Program states. Program states consist of a heap and a stack, *i.e.* $\textsc{State} = \textsc{Heap} \times \textsc{Stack}$. A heap H is a partial function from heap locations to a value mapping of object fields, *i.e.* $\textsc{Heap} = \textsc{Loc} \rightharpoonup (\textsc{FieldName} \rightharpoonup \textsc{Val})$. Stack frames are extended to also track a dynamic footprint:

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

$$E \in \textsc{StackFrame} = \textsc{VarEnv} \times \textsc{DynFprint} \times \textsc{Stmt}$$

During execution of an SVL program, expressions and statements operated on and mutated the topmost variable environment ρ. Now, programs may additionally access the heap as long as the topmost dynamic footprint contains the corresponding fields. Thus, the memory accessible at any point of execution can be viewed as a tuple $\textsc{Mem} = \textsc{Heap} \times \textsc{VarEnv} \times \textsc{DynFprint}$. We will need to adjust some definitions from SVL to operate on $\textsc{Mem}$ instead of $\textsc{VarEnv}$

The runtime semantics of SVL$_{\text{IDF}}$ always consults the footprint before accessing the heap, making sure that only accessible fields are touched.[5]

Evaluation. An expression e is evaluated according to a big-step evaluation relation $H, \rho \vdash e \Downarrow v$, yielding value v using heap H and local variable environment ρ. Variables are looked up in ρ, fields are looked up in H.

Predicate $\cdot \vDash \cdot \subseteq \textsc{Mem} \times \textsc{Formula}$ describes the rules for evaluating a formula. Fig. 3 shows the two rules that are specific to implicit dynamic frames (IDF) [18]. EAAcc checks whether access demanded by a formula is provided

$$\frac{H, \rho \vdash e \Downarrow o \qquad H, \rho \vdash e.f \Downarrow v \qquad \langle o, f\rangle \in A}{\langle H, \rho, A\rangle \vDash \texttt{acc}(e.f)}\ \text{EAAcc}$$

$$\frac{\langle H, \rho, A_1\rangle \vDash \phi_1 \qquad \langle H, \rho, A_2\rangle \vDash \phi_2}{\langle H, \rho, A_1 \uplus A_2\rangle \vDash \phi_1 * \phi_2}\ \text{EASepOp}$$

Fig. 3. SVL$_{\text{IDF}}$: Formula semantics (selected rules)

by the dynamic footprint. EASepOp implements the separating conjunction, making sure that access to the same field is not granted twice; for instance this ensures that `acc(x.f) * acc(y.f)` references two distinct fields.

Formula satisfiability and implication as defined in SVL are lifted from VarEnv to Mem accordingly.

Reduction rules. Figure 4 shows the reduction relation of SVL$_{\text{IDF}}$. We do not restate trivial rules like SsSeq or SsSeqSkip since they are virtually identical to the ones in SVL, *i.e.* the heap and footprint are irrelevant.

The field assignment rule SsFAssign updates the heap accordingly, provided that the assigned field is accessible. A similar check happens when assigning a variable, if the right-hand side expression contains a fields. (We use $\texttt{acc}(e)$ as syntactic sugar for $\texttt{acc}(e)$ in case e is a field access and `true` otherwise.) Note how SsCall now splits footprint A into disjoint sets according to the precondition (this is the implicitness about IDF). SsCallFinish recombines the footprints.

2.3 Static Verification

Figure 5 shows the adjusted and extended definition for weakest preconditions in SVL$_{\text{IDF}}$. Looking at the new definition for assignments x `:=` e we can see how the

[5] In separation logic [13] one unifies Heap and DynFprint by actually augmenting and splitting the heap itself instead of tracking footprints and passing around the full heap. We follow the implicit dynamic frames approach here.

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

$$\frac{\langle H, \rho, A\rangle \vDash \texttt{acc(}x\texttt{.}f\texttt{)} \qquad H, \rho \vdash y \Downarrow v \qquad H' = H[o \mapsto [f \mapsto v]]}{\langle H, \langle \rho, A, x\texttt{.}f\ \ \texttt{:=}\ \ y\rangle \cdot S\rangle \longrightarrow \langle H', \langle \rho, A, \texttt{skip}\rangle \cdot S\rangle}\ \textsc{SsFAssign}$$

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

$$\frac{\begin{array}{c} o \notin \mathsf{dom}(H) \\ \mathsf{fields}(C) = \overline{T_i\ \ f_i\texttt{;}} \qquad H' = H[o \mapsto [\overline{f_i \mapsto \mathsf{defaultValue}(T_i)}]] \end{array}}{\langle H, \langle \rho, A, x\ \ \texttt{:=}\ \texttt{new}\ \ C\rangle \cdot S\rangle \longrightarrow \langle H', \langle \rho[x \mapsto o], A \cup \overline{\langle o, f_i\rangle}, \texttt{skip}\rangle \cdot S\rangle}\ \textsc{SsAlloc}$$

$$\frac{\begin{array}{c} \mathsf{method}(m) = T_r\ m\texttt{(}T\ x'\texttt{)}\ \texttt{requires}\ \widehat{\phi}_p\ \texttt{ensures}\ \widehat{\phi}_q\ \texttt{\{}\ r\ \texttt{\}} \\ H, \rho \vdash z \Downarrow o \qquad H, \rho \vdash x \Downarrow v \\ \rho' = [\texttt{this} \mapsto o, x' \mapsto v] \qquad A' = \lfloor \widehat{\phi}_p \rfloor_{H, \rho'} \qquad \langle H, \rho', A'\rangle \vDash \widehat{\phi}_p \end{array}}{\langle H, \langle \rho, A, y\ \texttt{:=}\ z\texttt{.}m\texttt{(}x\texttt{);}\ s\rangle \cdot S\rangle \longrightarrow \langle H, \langle \rho', A', r\rangle \cdot \langle \rho, A \backslash A', y\ \texttt{:=}\ z\texttt{.}m\texttt{(}x\texttt{);}\ s\rangle \cdot S\rangle}\ \textsc{SsCall}$$

$$\frac{\mathsf{post}(m) = \widehat{\phi}_q \qquad \langle H, \rho', A'\rangle \vDash \widehat{\phi}_q \qquad \rho'' = \rho[y \mapsto \rho'(\texttt{result})]}{\langle H, \langle \rho', A', \texttt{skip}\rangle \cdot \langle \rho, A, y\ \texttt{:=}\ z\texttt{.}m\texttt{(}x\texttt{);}\ s\rangle \cdot S\rangle \longrightarrow \langle H, \langle \rho'', A \cup A', s\rangle \cdot S\rangle}\ \textsc{SsCallFinish}$$

Fig. 4. SVL$_{\text{IDF}}$: Small-step semantics (selected rules)

separating conjunction complicates matters in comparison to SVL. The rule can be read as follows: Find the weakest (maximum w.r.t. $\Rightarrow$) self-framed formula that implies both $\widehat{\phi}[e/x]$ (the "ordinary precondition") and whatever access is required to evaluate e. In presence of a non-separating conjunction $\wedge$ one could have defined $\mathsf{WLP}(x\ \texttt{:=}\ e, \widehat{\phi}) = \widehat{\phi}[e/x]\ \wedge\ \mathsf{acc}(e)$, however the same does not work with $*$. In case $\widehat{\phi}[e/x]$ already contains an accessibility predicate for e one would duplicate it which results in an unsatisfiable formula. The assertion rule shows this problem in its purest form.

2.4 Soundness

In addition to not getting stuck, the static semantics of SVL$_{\text{IDF}}$ also prevent null pointer exceptions. This is a result of ensuring that fields are accessible before accessing them and the way we interpret an accessibility predicate at runtime (see the premises of EAAcc in Figure 3).

Progress and preservation are defined just as in SVL. The notion of valid program states is adjusted to consider heap and dynamic footprint:

Definition 1 (Valid state). *We call the state* $\langle H, \langle \rho_n, A_n, s_n\rangle \cdot \ldots \cdot \langle \rho_1, A_1, s_1\rangle \cdot \mathsf{nil}\rangle \in \textsc{State}$ valid *if* $\langle H, \rho_i, A_i\rangle \vDash \mathsf{sWLP}_i(s_n \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$ *for all* $1 \le i \le n$.

$$
\begin{aligned}
\mathsf{WLP}(x\ \texttt{:=}\ e, \widehat{\phi}) &= \max_{\Rightarrow} \{\ \widehat{\phi}' \mid \widehat{\phi}' \Rightarrow \widehat{\phi}[e/x] \quad \wedge \quad \widehat{\phi}' \Rightarrow \mathsf{acc}(e)\ \} \\
\mathsf{WLP}(x.f\ \texttt{:=}\ y, \widehat{\phi}) &= \texttt{acc}(x.f) * \max_{\Rightarrow} \{\ \widehat{\phi}' \mid \widehat{\phi}' * \texttt{acc}(x.f) * (x.f\ \texttt{=}\ y) \Rightarrow \widehat{\phi} \quad \wedge \\
&\qquad\qquad \widehat{\phi}' * \texttt{acc}(x.f) \in \textsc{SatFormula}\ \} \\
\mathsf{WLP}(\texttt{assert}\ \phi_a, \widehat{\phi}) &= \max_{\Rightarrow} \{\ \widehat{\phi}' \mid \widehat{\phi}' \Rightarrow \widehat{\phi} \quad \wedge \quad \widehat{\phi}' \Rightarrow \phi_a\ \} \\
\mathsf{WLP}(y\ \texttt{:=}\ z.m(x), \widehat{\phi}) &= \max_{\Rightarrow} \{\ \widehat{\phi}' \mid\ y \notin \mathsf{FV}(\widehat{\phi}') \quad \wedge \\
&\quad \widehat{\phi}' \Rightarrow (z \neq \texttt{null}) * \mathsf{mpre}(m)[z, x/\texttt{this}, \mathsf{mparam}(m)] \quad \wedge \\
&\quad \widehat{\phi}' * \mathsf{mpost}(m)[z, x, y/\texttt{this}, \texttt{old}(\mathsf{mparam}(m)), \texttt{result}] \Rightarrow \widehat{\phi}\ \}
\end{aligned}
$$

Fig. 5. SVL$_{\text{IDF}}$: Weakest precondition (selected rules)

Note that heap H is used to check the precondition of all stack frames, *i.e.* there is no more strict isolation between stack frames. As a result, defining sWLP analogous to SVL leads to problems: Imagine a program state with a lower stack frame i having a WLP of `acc(x.f) * (x.f = 3)`. Assume that access to `x.f` was passed up the call stack (*i.e.* it was demanded by the preconditions of called functions), so currently executing statements can *change* the value of `x.f`. As a result, $\langle H, \rho_i, A_i \rangle \vDash \mathsf{sWLP}_i(s_n \cdot ... \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$ is violated. We solve this problem by making sure that the stack frame does *not* have a WLP of `acc(x.f) * (x.f = 3)` if it is currently buried under other stack frames that own `x.f`. Specifically, we encode framing rules into sWLP as shown in Fig. 6 and illustrated by Example 3.

sWLP^m is equivalent to sWLP, except that it weakens the top most formula just enough such that the heap locations it mentions are *disjoint* from those acquired by m. Effectively, $\widehat{\phi}_f$ represents the implicit frame of the executing function, so ownership of all fields mentioned in $\widehat{\phi}_f$ is withdrawn from the call site.

$$\mathsf{sWLP}(s \cdot \mathsf{nil}, \widehat{\phi}) = \mathsf{WLP}(s, \widehat{\phi}) \cdot \mathsf{nil}$$

$$\mathsf{sWLP}(s \cdot (y\ \texttt{:=}\ z.m(x)\texttt{;}\ s') \cdot \overline{s}, \widehat{\phi}) = \mathsf{WLP}(s, \mathsf{mpost}(m)) \cdot \mathsf{sWLP}^m((y\ \texttt{:=}\ z.m(x)\texttt{;}\ s') \cdot \overline{s}, \widehat{\phi})$$

where

$$\mathsf{sWLP}^m(\overline{s}, \widehat{\phi}) = \min_{\Rightarrow} \{\ \widehat{\phi}'_n \mid \widehat{\phi}_n \Rightarrow \widehat{\phi}_f * \widehat{\phi}'_n\ \} \cdot \widehat{\phi}_{n-1} \cdot ... \cdot \widehat{\phi}_1 \cdot \mathsf{nil}$$

where $\widehat{\phi}_n \cdot \widehat{\phi}_{n-1} \cdot ... \cdot \widehat{\phi}_1 \cdot \mathsf{nil} = \mathsf{sWLP}(\overline{s}, \widehat{\phi})$ and $\widehat{\phi}_f = \mathsf{mpre}(m)[z, x/\texttt{this}, \mathsf{mparam}(m)]$

Fig. 6. Heap aware weakest precondition across call boundaries.

Example 3 (sWLP *and shared mutable heap).* Consider the `withdraw` function from the introduction[6]:

```
void withdraw(int amount)
        requires acc(this.balance) * (this.balance ≥ amount)
        ensures  acc(this.balance) * (this.balance ≥ 0) {
    this.balance = this.balance - amount;
}
```

For a postcondition $\phi = \texttt{acc(a.id)} * \texttt{acc(a.balance)}*\texttt{(a.balance} \neq \texttt{-1)} * \texttt{(n = 5)}$ we calculate the following weakest preconditions for different stages of executing `withdraw`:

$$(1)\qquad \mathsf{sWLP}\left(\ \texttt{a.withdraw(n)}\ ,\phi\right) = \texttt{acc(a.id)} * \texttt{acc(a.balance)} * \texttt{(a.balance} \geq \texttt{n)} * \texttt{(n = 5)}$$

$$(2)\qquad \mathsf{sWLP}\left(\begin{matrix}\mathsf{mbody}(\texttt{withdraw})\\ \cdot \\ \texttt{a.withdraw(n)}\end{matrix},\phi\right) = \begin{matrix}\texttt{acc(this.balance)} * \texttt{(this.balance - amount} \geq \texttt{0)}\\ \cdot \\ \texttt{acc(a.id)} * \texttt{(n = 5)}\end{matrix}$$

$$(3)\qquad \mathsf{sWLP}\left(\begin{matrix}\texttt{skip}\\ \cdot \\ \texttt{a.withdraw(n)}\end{matrix},\phi\right) = \begin{matrix}\texttt{acc(this.balance)} * \texttt{(this.balance} \geq \texttt{0)}\\ \cdot \\ \texttt{acc(a.id)} * \texttt{(n = 5)}\end{matrix}$$

$$(4)\qquad \mathsf{sWLP}\left(\quad \texttt{skip}\quad ,\phi\right) = \texttt{acc(a.id)} * \texttt{acc(a.balance)} * \texttt{(a.balance} \neq \texttt{-1)} * \texttt{(n = 5)}$$

(Above, we have dropped the nil bottom of stacks for conciseness.) Note how entering the call (from (1) to (2)) caused the call site WLP to be reduced to `acc(a.id) * (n = 5)` since the callee demands `acc(a.balance)`. Without this reduction, the call site WLP may not reflect reality in step (3): `a.balance` has been reduced by 5 by the callee, so `(a.balance ≥ n)` will no longer hold if `a.balance` had an initial value of, say, 7.

This example demonstrates how sWLP (Fig. 6) reflects the exclusive access model of IDF.

3 GVL$_{\mathbf{IDF}}$: Static Semantics

We define:

$$\widetilde{\phi} ::= \widehat{\phi} \mid \texttt{?} * \phi \qquad\qquad \text{with syntactic sugar } \texttt{?} \overset{\text{def}}{=} \texttt{?} * \texttt{true}$$

For maximum expressiveness, we want to enable `?` to *provide* framing for a formula that is otherwise not self-framed, hence we do not require the static part of imprecise formulas to be self framed. [7]

[6] We can reason about functions returning `void` by thinking of them as returning an `int` (*e.g.* 0). At call sites, the return value would be assigned to some dummy variable.

[7] To emphasize this fact, we write the `?` first, suggesting that it can frame the later part.

3.1 Interpretation of Gradual Formulas

We adjust concretization as defined for SVL to only consider self-framed formulas:

Definition 2 (Concretization of gradual formulas). $\gamma : \widetilde{\text{FORMULA}} \rightharpoonup \mathcal{P}^{\text{FORMULA}}$ *is defined as:*

$$\begin{aligned}
&\gamma(\widehat{\phi}) = \{\ \widehat{\phi}\ \} \\
&\gamma(\texttt{?} * \phi) = \{\ \widehat{\phi}' \in \text{SATFORMULA} \mid \widehat{\phi}' \Rightarrow \phi\ \} \qquad \textit{if } \phi \in \text{SATFORMULA} \\
&\gamma(\texttt{?} * \phi) \quad \textit{undefined otherwise}
\end{aligned}$$

Allowing `?` to provide framing enables the programmer to express `(a.name = b.name)` irrespective of whether `a` and `b` alias or not. The gradual formula `?*(a.name = b.name)` captures both scenarios, since the unknown part can either denote the non-aliasing case `acc(a.name)*acc(b.name)` or the aliasing case `(a = b)*acc(b.name)`.

3.2 Lifting Predicates

Allowing `?` to provide framing for the static part of an imprecise formula requires extra care when evaluating formulas:

Lemma 1 (Consistent Formula Evaluation).
Let $\cdot \mathrel{\widetilde{\vDash}} \cdot \subseteq \text{MEM} \times \widetilde{\text{FORMULA}}$ *be defined inductively as*

$$\frac{m \vDash \widehat{\phi}}{m \mathrel{\widetilde{\vDash}} \widehat{\phi}}\ \widetilde{\text{EVALSTATIC}}$$

$$\frac{m \vDash \phi \qquad A \vdash_{\mathit{frm}} \phi \qquad \forall \langle e, f\rangle \in A.\ m \vDash \texttt{acc}(e.f)}{m \mathrel{\widetilde{\vDash}} \texttt{?} * \phi}\ \widetilde{\text{EVALGRAD}}$$

Then $\cdot \mathrel{\widetilde{\vDash}} \cdot$ *is a consistent lifting of* $\cdot \vDash \cdot$.

The two additional premises of $\widetilde{\text{EVALGRAD}}$ ensure that one can frame ϕ in a way that is supported by m. Note that although A is an existential, it can be chosen to be the set of all fields mentioned in ϕ. Thus, it can also be precomputed since formula ϕ is known statically.

3.3 Lifting Functions

Figure 7 formalizes how WLP is consistently lifted. Note how for calls one essentially lifts the function as if it had three parameters, two of them hidden in m. Algorithmic definitions for $\widetilde{\mathsf{WLP}}$ are significantly more complicated than those for GVL due to the separating conjunction. Some implementation pointers can be found in the supplementary material, furthermore our prototype implementation implements a similar function.

However, recall that this function is only required by the *static* checker and hence not the main focus of our performance efforts. The formal definition should allow SMT solvers to check validity of programs. Furthermore, it is valid to conservatively approximate $\widetilde{\mathsf{WLP}}$ (erring towards imprecision) which may lead to more runtime checks to make up for the loss of precision. It is worth noting that defining $\widetilde{\mathsf{WLP}}(s, \widetilde{\phi}) = \mathtt{?}$ is the most trivial conservative approximation and would essentially result in a fully dynamically verified language. Of course, pay-as-you-go overhead is no longer guaranteed with an approximated $\widetilde{\mathsf{WLP}}$.

Recall how we adjusted sWLP (Fig. 6) in order for it to be usable for formalizing small step soundness. Figure 8 shows the corresponding definition for $\mathrm{GVL}_{\mathrm{IDF}}$. Note that an imprecise method precondition makes it impossible to make assumptions about which access is retained at the call site. We conservatively approximate the frame by assuming that access to no fields remains with the call site: For $\widehat{\phi}'_n$, an empty footprint is enforced, which due to self-framing of $\widehat{\phi}'_n$ also prohibits the *use* of any fields, demonstrated in example 4.

Example 4 ($\widetilde{\mathsf{sWLP}}$ and shared mutable heap).
We redefine the precondition of `withdraw` used in example 3 to be `?*(this.balance` $\geq$ `amount)`.
For the same $\phi =$ `acc(a.id) * acc(a.balance)*(a.balance` $\neq$ `-1) * (n = 5)`
we calculate the following weakest preconditions:

$$(1)\qquad \widetilde{\mathsf{sWLP}}\Big(\ \mathtt{a.withdraw(n)}\ ,\phi\Big) = \mathtt{?} * \mathtt{(a.balance} \geq \mathtt{n)} * \mathtt{(n = 5)}$$

$$(2)\qquad \widetilde{\mathsf{sWLP}}\begin{pmatrix}\mathsf{mbody}(\mathtt{withdraw}) & \\ \cdot & ,\phi \\ \mathtt{a.withdraw(n)} & \end{pmatrix} = \begin{matrix}\mathtt{acc(this.balance)} * \mathtt{(this.balance - amount} \geq \mathtt{0)} \\ \cdot \\ \mathtt{?} * \mathtt{(n = 5)}\end{matrix}$$

$$(3)\qquad \widetilde{\mathsf{sWLP}}\begin{pmatrix}\mathtt{skip} & \\ \cdot & ,\phi \\ \mathtt{a.withdraw(n)} & \end{pmatrix} = \begin{matrix}\mathtt{acc(this.balance)} * \mathtt{(this.balance} \geq \mathtt{0)} \\ \cdot \\ \mathtt{?} * \mathtt{(n = 5)}\end{matrix}$$

$$(4)\qquad \widetilde{\mathsf{sWLP}}\Big(\quad \mathtt{skip}\quad ,\phi\Big) = \mathtt{acc(a.id)} * \mathtt{acc(a.balance)} * \mathtt{(a.balance} \neq \mathtt{-1)} * \mathtt{(n = 5)}$$

During the call, the call site can no longer be certain about having access to `a.id` since this field may be modified by the callee. The only static knowledge is `(n = 5)`, which does not depend on the heap.

A naive runtime semantics might perform two checks to ensure soundness: First, when transitioning from (1) to (2), ensuring that `acc(this.balance)*(this.balance - amount` $\geq$ `0)` (the callee's internal precondition) holds. Second, when transitioning from (3) to (4), ensuring that `acc(a.id) * acc(a.balance)*(a.balance` $\neq$ `-1) * (n = 5)` holds. The second check should be reduced to `acc(a.id)` (`(n = 5)` is known, the remaining part implied by `withdraw`'s postcondition). The first check can be skipped since `withdraw`'s precondition `? * (this.balance` $\geq$ `amount)`, despite being imprecise, guarantees that the above check succeeds: *Every* concretization of the precondition (by definition self-framed, see Def. 2) implies the check. If `withdraw`'s precondition was changed to `?`, the check can no longer be dropped or even reduced.

We expect the checks predicted in 4 to be realized in the following runtime semantics.

$$
\begin{aligned}
\widetilde{\mathsf{WLP}}(y\ \texttt{:=}\ z\texttt{.}m\texttt{(}x\texttt{)}, \widetilde{\phi}) = \alpha(\{\ \max_{\Rightarrow}\ \{\ \widehat{\phi}' \mid y \notin \mathsf{FV}(\widehat{\phi}') \quad &\wedge \\
\widehat{\phi}' \Rightarrow \texttt{(}z\ \neq\ \texttt{null)} * \widehat{\phi}_p[z, x/\texttt{this}, \mathsf{mparam}(m)] \quad &\wedge \\
\widehat{\phi}' * \widehat{\phi}_q[z, x, y/\texttt{this}, \texttt{old(}\mathsf{mparam}(m)\texttt{)}, \texttt{result}] \Rightarrow \widehat{\phi}\ \} & \\
\mid \widehat{\phi} \in \gamma(\widetilde{\phi}),\ \widehat{\phi}_p \in \gamma(\mathsf{mpre}(m)),\ \widehat{\phi}_q \in \gamma(\mathsf{mpost}(m))\ \}) & \\
\widetilde{\mathsf{WLP}}(s, \widetilde{\phi}) = \alpha(\{\ \mathsf{WLP}(s, \widehat{\phi}) \mid \widehat{\phi} \in \gamma(\widetilde{\phi})\ \}) \quad \text{if } s \text{ is not a call statement} &
\end{aligned}
$$

Fig. 7. GVL_{IDF}: Weakest preconditions.

$$
\begin{aligned}
\widetilde{\mathsf{WLP}}(s \cdot \mathsf{nil}, \widetilde{\phi}) &= \widetilde{\mathsf{WLP}}(s, \widetilde{\phi}) \cdot \mathsf{nil} \\
\widetilde{\mathsf{WLP}}(s \cdot (y\ \texttt{:=}\ z\texttt{.}m\texttt{(}x\texttt{)}\texttt{;}\ s') \cdot \overline{s}, \widetilde{\phi}) &= \widetilde{\mathsf{WLP}}(s, \mathsf{mpost}(m)) \cdot \widetilde{\mathsf{WLP}}^m((y\ \texttt{:=}\ z\texttt{.}m\texttt{(}x\texttt{)}\texttt{;}\ s') \cdot \overline{s}, \widetilde{\phi}) \\
\text{where} & \\
\widetilde{\mathsf{WLP}}^m(\overline{s}, \widetilde{\phi}) &= \begin{cases} \widehat{\phi}'_n \cdot \widetilde{\phi}_{n-1} \cdot \ldots \cdot \widetilde{\phi}_1 \cdot \mathsf{nil} & \text{if } \widetilde{\phi}_f \text{ and } \widetilde{\phi}_n \text{ precise} \\ \texttt{?} * \widehat{\phi}'_n \cdot \widetilde{\phi}_{n-1} \cdot \ldots \cdot \widetilde{\phi}_1 \cdot \mathsf{nil} & \text{otherwise} \end{cases} \\
& \text{where } \widetilde{\phi}_n \cdot \widetilde{\phi}_{n-1} \cdot \ldots \cdot \widetilde{\phi}_1 \cdot \mathsf{nil} = \widetilde{\mathsf{WLP}}(\overline{s}, \widetilde{\phi}) \\
& \qquad \widehat{\phi}'_n = \begin{cases} \min_{\Rightarrow} \{\ \widehat{\phi}'_n \mid \mathsf{static}(\widetilde{\phi}_n) \Rightarrow \widehat{\phi}'_n * \widetilde{\phi}_f\ \} & \text{if } \widetilde{\phi}_f \text{ precise} \\ \min_{\Rightarrow} \{\ \widehat{\phi}'_n \mid \mathsf{static}(\widetilde{\phi}_n) \Rightarrow \widehat{\phi}'_n\ \wedge\ \lfloor \widehat{\phi}'_n \rfloor = \emptyset\ \} & \text{otherwise} \end{cases} \\
& \text{and} \quad \widetilde{\phi}_f = \mathsf{mpre}(m)[z, x/\texttt{this}, \mathsf{mparam}(m)]
\end{aligned}
$$

Fig. 8. Heap aware weakest precondition across call boundaries.

4 GVL$_{\text{IDF}}$: Dynamic Semantics

Figure 9 shows a dynamic semantics using residual checks. Note that only the last premise of both SsCall and SsCallFinish represents overhead.

Apart from using consistent judgments due to the gradual contracts, we must also adjust footprint splitting in $\widetilde{\text{SsCall}}$: If the precondition is imprecise the dynamic frame of the invocation cannot be precisely determined either. Driven by the dynamic gradual guarantee, we must ensure that introducing imprecision will not introduce a runtime error caused by lack of accessibility. Hence, we conservatively pass the entire dynamic footprint A of the call site on to the callee. This is inherently a superset of the access granted in any precise setting. In contrast, the call site has no more access to the heap, which is reflected by the adjusted definition of $\mathsf{s}\widetilde{\mathsf{WLP}}$ that removes all mentions of the heap from the call site's $\widetilde{\mathsf{WLP}}$ (Fig. 8). Only when the call finishes, the footprint is merged back to the call site, restoring its accessibility.

It is worth mentioning that it is still (as in GVL) sufficient to only inject runtime assertions concerning the top most stack frame. Although $\mathsf{s}\widetilde{\mathsf{WLP}}$ no longer leaves the $\widetilde{\mathsf{WLP}}$ of lower stack frames unchanged, it only weakens them (Fig. 8). The runtime assertion check performed by $\widetilde{\text{SsCallFinish}}$ (last premise) is derived analogous to the one we derived for GVL ($|S| + 1$ selects the top most component of the sWLP; $s \cdot ...$ is the stack of statements s and the ones in S).

$$\frac{\langle H', \langle \rho'_n, A'_n, (s\texttt{;}\ s_n)\rangle \cdot ...\rangle \longrightarrow \langle H, \langle \rho_n, A_n, s_n\rangle \cdot ...\rangle}{\langle H', \langle \rho'_n, A'_n, (s\texttt{;}\ s_n)\rangle \cdot ...\rangle \mathrel{\widetilde{\longrightarrow}} \langle H, \langle \rho_n, A_n, s_n\rangle \cdot ...\rangle}\ \widetilde{\text{SsLocal}}$$

$$\frac{\begin{array}{c}\mathsf{method}(m) = T_r\ m(T\ x')\ \texttt{requires}\ \widetilde{\phi}_p\ \texttt{ensures}\ \widetilde{\phi}_q\ \texttt{\{}\ r\ \texttt{\}} \qquad H, \rho \vdash z \Downarrow o \\ H, \rho \vdash x \Downarrow v \qquad \rho' = [\texttt{this} \mapsto o, x' \mapsto v] \qquad A' = \begin{cases} \lfloor \widetilde{\phi}_p \rfloor_{H,\rho'} & \text{if } \widetilde{\phi}_p \text{ precise} \\ A & \text{otherwise} \end{cases} \\ \langle H, \rho', A'\rangle \mathrel{\widetilde{\vDash}} \widetilde{\phi}_p \qquad \langle H, \rho', A'\rangle \mathrel{\widetilde{\vDash}} \widetilde{\mathsf{diff}}(\widetilde{\mathsf{WLP}}(r, \widetilde{\phi}_q), \widetilde{\phi}_p)\end{array}}{\langle H, \langle \rho, A, y\ \texttt{:=}\ z\texttt{.}m(x)\texttt{;}\ s\rangle \cdot S\rangle \mathrel{\widetilde{\longrightarrow}} \langle H, \langle \rho', A', r\rangle \cdot \langle \rho, A \backslash A', y\ \texttt{:=}\ z\texttt{.}m(x)\texttt{;}\ s\rangle \cdot S\rangle}\ \widetilde{\text{SsCall}}$$

$$\frac{\begin{array}{c}\mathsf{mpost}(m) = \widetilde{\phi}_q \qquad \langle H, \rho', A'\rangle \mathrel{\widetilde{\vDash}} \widetilde{\phi}_q \qquad \rho'' = \rho[y \mapsto \rho'(\texttt{result})] \\ \widetilde{\phi}'_q = \widetilde{\phi}_q[z, x, y/\texttt{this}, \texttt{old(}\mathsf{mparam}(m)\texttt{)}, \texttt{result}] \\ \widetilde{\phi}'_p = \mathsf{mpre}(m)[z, x/\texttt{this}, \mathsf{mparam}(m)] \\ \rho_n \mathrel{\widetilde{\vDash}} \widetilde{\mathsf{diff}}(\widetilde{\mathsf{diff}}(\mathsf{s}\widetilde{\mathsf{WLP}}_{|S|+1}(s \cdot ..., \texttt{true}), \widetilde{\phi}'_p), \widetilde{\phi}'_q)\end{array}}{\langle H, \langle \rho', A', \texttt{skip}\rangle \cdot \langle \rho, A, y\ \texttt{:=}\ z\texttt{.}m(x)\texttt{;}\ s\rangle \cdot S\rangle \mathrel{\widetilde{\longrightarrow}} \langle H, \langle \rho'', A \cup A', s\rangle \cdot S\rangle}\ \widetilde{\text{SsCallFinish}}$$

Fig. 9. Dynamic semantics of GVL$_{\text{IDF}}$ with reduced checks.

4.1 Relying on dynamically tracked footprints

Recall that m also contains a dynamic footprint tracking the set of accessible fields. As noted in Sect. 2.2, an implementation of SVL_{IDF} need not actually maintain such a footprint because verification is entirely static, and sound. However, for gradual verification, the reduction rules of Fig. 9 rely on a realization of m that *can* decide accessibility.

Therefore we use a runtime that tracks the footprint A (as modeled in SVL_{IDF}) as starting point, giving criteria for simplifying it in the next section. The situation is comparable to that of GVL regarding `old`: An entity of the static verification semantics (but not original runtime semantics) becomes part of the gradual runtime semantics since it now performs verification, too.

4.2 Tracking Dynamic Footprints

A key observation is that precisely-annotated code results in residual formulas `true`. This not only allows the compiler to drop the corresponding runtime checks, but also implies that dynamic footprint A will not be required. With increasing imprecision, the set of fields that A could *potentially* be queried for grows.

In the following, we outline a simple data-flow based strategy to determine an upper bound to the set $\mathring{A} \subset \mathcal{P}^{\textsc{Expr}\times\textsc{FieldName}}$ of fields (represented syntactically) that might be queried for. For every allocation `x := new C`, A will only be augmented with the fields of x if they can be found in $\mathring{A}$ (compare with rule SsAlloc of Fig. 4, which unconditionally adds all fields). As a result, since $\mathring{A}$ is empty in precisely-annotated code, dynamic footprint A will remain empty throughout execution.

Note that $\mathring{A}$ may be larger than A. Specifically, $\mathring{A}$ may be infinite at the call site of a method with precondition `?` that accepts a linked list as parameter, *e.g.*

$$\mathring{A} = \{\ \langle \texttt{l}, \texttt{head}\rangle, \langle \texttt{l.tail}, \texttt{head}\rangle, \langle \texttt{l.tail.tail}, \texttt{head}\rangle, \ldots\ \}$$

However, this can only affect a finite set of allocations. For example, if the statement prior to the call is an allocation `l := new List`, then this statement will be tagged to add both $\langle o, \texttt{head}\rangle$ and $\langle o, \texttt{tail}\rangle$ to A at runtime (where o is the new heap reference). This covers elements $\langle \texttt{l}, \texttt{head}\rangle$ and $\langle \texttt{l.tail}, \texttt{head}\rangle$ of $\mathring{A}$. If there's an additional assignment `l.tail := l` then all of $\mathring{A}$ is covered by the the two elements in A due to aliasing.

Determining $\mathring{A}$ We informally outline how to iteratively determine $\mathring{A}$ for each location in the program. We look at each method in isolation, starting from the last statement of each method and working our way to the front.

If a method m's post condition is precise, we initialize $\mathring{A} = \emptyset$. Otherwise, callers of m may query for any access we could possibly return so we initialize $\mathring{A}$ to contain all fields reachable from a call to m that are not statically guaranteed to be accessible. For example, for a method with signature

`Point2D` m`(Point3D p) requires acc(p.y) ensures ? * (result.x = 3) { ... }` we initialize $\mathring{A} = \{\ \langle \mathtt{p}, \mathtt{x}\rangle, \langle \mathtt{p}, \mathtt{z}\rangle, \langle \mathtt{result}, \mathtt{y}\rangle\ \}$ since access to `p.y` and `result.x` is statically derivable.

$\mathring{A}$ is propagated upwards from there, undergoing corresponding transformations such as substituting e for x due to assignment x `:=` e. Method calls y `:=` z`.`m`(`x`)` require further inspection: Any (residual) runtime assertion checks encountered augment $\mathring{A}$ by the fields mentioned. Furthermore, all mentions of variable y may be removed as accessibility of its field must come from m. However, if m has an imprecise precondition, $\mathring{A}$ is again augmented by the potentially accessible set of fields that is not statically guaranteed. In case of the example signature above, this would be $\{\ \langle \mathtt{p}, \mathtt{x}\rangle\ \}$.

4.3 Properties of GVL$_{\text{IDF}}$

GVL$_{\text{IDF}}$ is a sound, conservative extension of SVL$_{\text{IDF}}$ and has a pay-as-you-go cost model for runtime checks. It satisfies the gradual guarantee, with a slight adjustment to the dynamic gradual guarantee. The fact that footprint tracking and splitting is influenced by increasing imprecision means that we can no longer talk about *equal* program states π_1 and π_2 as we did in GVL. We hence first refine state equality to an asymmetric *state precision* relation $\lesssim$:

Definition 3 (State Precision). *Let $\pi_1, \pi_2 \in$ State. Then π_1 is more precise than π_2, written $\pi_1 \lesssim \pi_2$, if and only if all of the following applies:*
a) *π_1 and π_2 have identical heap and stacks of size n.*
b) *The stack of variable environments and stack of statements is identical.*
c) *Let $A^1_{1..n}$ and $A^2_{1..n}$ be the stack of footprints of π_1 and π_2, respectively. Then the following holds for $1 \leq m \leq n$:*

$$\bigcup_{i=m}^{n} A^1_i \subseteq \bigcup_{i=m}^{n} A^2_i$$

The new dynamic gradual guarantee reflects the fact that imprecision potentially results in larger parts of footprints being passed up the stack, and hence residual checks, which are evaluated against the top most stack frame, are more likely to succeed in the less precise program. Therefore, Prop. 1 establishes that successful progress from a state π_1 for a program p_1 also implies successful progress for a less precise state π_2 and less precise program p_2.

Proposition 1 (Dynamic gradual guarantee of verification).
Let $p_1, p_2 \in$ Program such that $p_1 \sqsubseteq p_2$, and $\pi_1, \pi_2 \in$ State such that $\pi_1 \lesssim \pi_2$.
If $\pi_1 \xrightarrow{\sim}_{p_1} \pi_1'$ then $\pi_2 \xrightarrow{\sim}_{p_2} \pi_2'$, with $\pi_1' \lesssim \pi_2'$.

References

1. 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
2. Cheon, Y., Leavens, G.T.: A runtime assertion checker for the java modeling language (jml) (2002)
3. 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
4. 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)
5. 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
6. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12(10), 576–580 (1969)
7. 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)
8. 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)
9. Meyer, B.: Eiffel: A language and environment for software engineering. Journal of Systems and Software 8(3), 199–246 (1988)
10. Meyer, B.: Object-Oriented Software Construction. Prentice Hall (1988)
11. Nanevski, A., Morrisset, G., Birkedal, L.: Hoare type theory, polymorphism and separation. Journal of Functional Programming 5-6, 865–911 (2008)
12. 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)
13. 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)
14. 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
15. Siek, J., Taha, W.: Gradual typing for objects. In: European Conference on Object-Oriented Programming. pp. 2–27. Springer (2007)
16. Siek, J.G., Taha, W.: Gradual typing for functional languages. In: Scheme and Functional Programming Workshop. vol. 6, pp. 81–92 (2006)
17. 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)
18. 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)

19. Wolff, R., Garcia, R., Tanter, É., Aldrich, J.: Gradual typestate. In: European Conference on Object-Oriented Programming. pp. 459–483. Springer (2011)
20. 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`

A Appendix

A.1 Full Definitions

$$
\begin{aligned}
\mathit{program} &::= \overline{\mathit{cls}}\ s \\
\mathit{cls} &::= \texttt{class}\ C\ \texttt{\{}\ \overline{\mathit{field}}\ \overline{\mathit{method}}\ \texttt{\}} \\
\mathit{field} &::= T\ \ f\texttt{;} \\
\mathit{method} &::= T\ m\texttt{(}T\ x\texttt{)}\ \mathit{contract}\ \texttt{\{}\ s\ \texttt{\}} \\
\mathit{contract} &::= \texttt{requires}\ \widehat{\phi}\ \texttt{ensures}\ \widehat{\phi} \\
T &::= \texttt{int} \mid C \\
\phi &::= \texttt{true} \mid (e\ \odot\ e) \mid \phi * \phi \mid \texttt{acc(}e\texttt{.}f\texttt{)}
\end{aligned}
\qquad
\begin{aligned}
s &::= \texttt{skip} \mid s_1\texttt{;}\ s_2 \mid T\ \ x \mid x\ \texttt{:=}\ e \mid \texttt{assert}\ \phi \\
&\quad\mid x\texttt{.}f\ \ \texttt{:=}\ y \mid x\ \ \texttt{:=}\ \texttt{new}\ C \mid y\ \texttt{:=}\ z\texttt{.}m\texttt{(}x\texttt{)} \\
e &::= v \mid x \mid (e\ \oplus\ e) \mid e\texttt{.}f \\
x &::= \texttt{result} \mid \mathit{ident} \mid \texttt{old(}\mathit{ident}\texttt{)} \mid \texttt{this} \\
v &::= n \mid o \mid \texttt{null} \\
\oplus &::= \texttt{+} \mid \texttt{-} \mid \ldots \\
\odot &::= \ \texttt{=} \mid \neq \mid \texttt{<} \mid \ldots
\end{aligned}
$$

where $o \in \textsc{Loc}$ is a heap location and $f \in \textsc{FieldName}$ are field names

Fig. 10. SVL$_{\text{IDF}}$: Syntax

$$\frac{}{A_s \vdash_{\texttt{frm}} x}\ \textsc{FrmVar} \qquad \frac{}{A_s \vdash_{\texttt{frm}} v}\ \textsc{FrmVal} \qquad \frac{\langle e, f\rangle \in A \qquad A \vdash_{\texttt{frm}} e}{A \vdash_{\texttt{frm}} e.f}\ \textsc{FField}$$

$$\frac{}{A \vdash_{\texttt{frm}} \texttt{true}}\ \textsc{SfrmTrue} \qquad \frac{A \vdash_{\texttt{frm}} e_1 \qquad A \vdash_{\texttt{frm}} e_2}{A \vdash_{\texttt{frm}} e_1 \odot e_2}\ \textsc{SFComp}$$

$$\frac{A \vdash_{\texttt{frm}} e}{A \vdash_{\texttt{frm}} \texttt{acc}(e.f)}\ \textsc{SFAcc} \qquad \frac{A \vdash_{\texttt{frm}} \phi_1 \qquad A \cup \lfloor \phi_1 \rfloor \vdash_{\texttt{frm}} \phi_2}{A \vdash_{\texttt{frm}} \phi_1 * \phi_2}\ \textsc{SFSepOp}$$

Fig. 11. SVL$_{\text{IDF}}$: Framing

$$\frac{}{\langle H, \rho, A\rangle \vDash \texttt{true}}\ \textsc{EATrue} \qquad \frac{H, \rho \vdash e_1 \Downarrow v_1 \qquad H, \rho \vdash e_2 \Downarrow v_2 \qquad v_1 \odot v_2}{\langle H, \rho, A\rangle \vDash (e_1\ \odot\ e_2)}\ \textsc{EAComp}$$

$$\frac{H, \rho \vdash e \Downarrow o \qquad H, \rho \vdash e.f \Downarrow v \qquad \langle o, f\rangle \in A}{\langle H, \rho, A\rangle \vDash \texttt{acc}(e.f)}\ \textsc{EAAcc}$$

$$\frac{\langle H, \rho, A_1\rangle \vDash \phi_1 \qquad \langle H, \rho, A_2\rangle \vDash \phi_2}{\langle H, \rho, A_1 \uplus A_2\rangle \vDash \phi_1 * \phi_2}\ \textsc{EASepOp}$$

Fig. 12. SVL$_{\text{IDF}}$: Formula semantics

$$\frac{}{\langle H, \langle \rho, A, \texttt{skip}\rangle \cdot S\rangle \longrightarrow \langle H, \langle \rho, A, \texttt{skip}\rangle \cdot S\rangle}\ \textsc{SsSkip}$$

$$\frac{\langle H, \langle \rho, A, s_1\rangle \cdot S\rangle \longrightarrow \langle H, \langle \rho', A, s_1'\rangle \cdot S\rangle}{\langle H, \langle \rho, A, s_1\,\texttt{;}\ s_2\rangle \cdot S\rangle \longrightarrow \langle H, \langle \rho', A, s_1'\,\texttt{;}\ s_2\rangle \cdot S\rangle}\ \textsc{SsSeq}$$

$$\frac{}{\langle H, \langle \rho, A, \texttt{skip}\rangle \cdot S\rangle \longrightarrow \langle H, \langle \rho, A, \texttt{skip}\rangle \cdot S\rangle}\ \textsc{SsSkip}$$

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

$$\frac{\langle H, \rho, A\rangle \vDash \texttt{acc(}x\texttt{.}f\texttt{)} \qquad H, \rho \vdash y \Downarrow v \qquad H' = H[o \mapsto [f \mapsto v]]}{\langle H, \langle \rho, A, x\texttt{.}f\ \texttt{:=}\ y\rangle \cdot S\rangle \longrightarrow \langle H', \langle \rho, A, \texttt{skip}\rangle \cdot S\rangle}\ \textsc{SsFAssign}$$

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

$$\frac{\rho' = \rho[x \mapsto \mathsf{defaultValue(T)}]}{\langle H, \langle \rho, A, T\ \ x\rangle \cdot S\rangle \longrightarrow \langle H, \langle \rho', A, \texttt{skip}\rangle \cdot S\rangle}\ \textsc{SsDeclare}$$

$$\frac{\begin{array}{c}\mathsf{method}(m) = T_r\ m\texttt{(}T\ x'\texttt{)}\ \texttt{requires}\ \widehat{\phi}_p\ \texttt{ensures}\ \widehat{\phi}_q\ \texttt{\{}\ r\ \texttt{\}} \\ H, \rho \vdash z \Downarrow o \qquad H, \rho \vdash x \Downarrow v \\ \rho' = [\texttt{this} \mapsto o, x' \mapsto v] \qquad A' = \lfloor \widehat{\phi}_p \rfloor_{H,\rho'} \qquad \langle H, \rho', A'\rangle \vDash \widehat{\phi}_p\end{array}}{\langle H, \langle \rho, A, y\ \texttt{:=}\ z\texttt{.}m\texttt{(}x\texttt{)}\texttt{;}\ s\rangle \cdot S\rangle \longrightarrow \langle H, \langle \rho', A', r\rangle \cdot \langle \rho, A \backslash A', y\ \texttt{:=}\ z\texttt{.}m\texttt{(}x\texttt{)}\texttt{;}\ s\rangle \cdot S\rangle}\ \textsc{SsCall}$$

$$\frac{\mathsf{post}(m) = \widehat{\phi}_q \qquad \langle H, \rho', A'\rangle \vDash \widehat{\phi}_q \qquad \rho'' = \rho[y \mapsto \rho'\texttt{(result)}]}{\langle H, \langle \rho', A', \texttt{skip}\rangle \cdot \langle \rho, A, y\ \texttt{:=}\ z\texttt{.}m\texttt{(}x\texttt{)}\texttt{;}\ s\rangle \cdot S\rangle \longrightarrow \langle H, \langle \rho'', A \cup A', s\rangle \cdot S\rangle}\ \textsc{SsCallFinish}$$

Fig. 13. SVL_{IDF}: Local small-step semantics

$$\begin{array}{ll}
\mathsf{WLP}(\texttt{skip}, \widehat{\phi}) & = \widehat{\phi} \\
\mathsf{WLP}(s_1\texttt{;}\ s_2, \widehat{\phi}) & = \mathsf{WLP}(s_1, \mathsf{WLP}(s_2, \widehat{\phi})) \\
\mathsf{WLP}(T\ \ x, \widehat{\phi}) & = \widehat{\phi}[\mathsf{defaultValue}(T)/x] \\
\mathsf{WLP}(x\ \texttt{:=}\ e, \widehat{\phi}) & = \max\limits_{\Rightarrow}\ \{\ \widehat{\phi}'\ \mid \widehat{\phi}' \Rightarrow \widehat{\phi}[e/x] \quad \wedge \quad \widehat{\phi}' \Rightarrow \mathsf{acc}(e)\ \} \\
\mathsf{WLP}(x\texttt{.}f\ \ \texttt{:=}\ y, \widehat{\phi}) & = \texttt{acc(}x\texttt{.}f\texttt{)} * \max\limits_{\Rightarrow}\ \{\ \widehat{\phi}' \mid \widehat{\phi}' * \texttt{acc(}x\texttt{.}f\texttt{)} * \texttt{(}x\texttt{.}f\ \texttt{=}\ y\texttt{)} \Rightarrow \widehat{\phi} \quad \wedge \\
& \qquad\qquad\qquad\quad \widehat{\phi}' * \texttt{acc(}x\texttt{.}f\texttt{)} \in \textsc{SatFormula}\ \} \\
\mathsf{WLP}(x\ \ \texttt{:=}\ \texttt{new}\ C, \widehat{\phi}) & = \max\ \{\ \widehat{\phi}' \mid \widehat{\phi}' * \texttt{(}x\ \neq\ \texttt{null)} \\
& \quad * \overline{\texttt{acc(}x\texttt{.}f_i\texttt{)} * \texttt{(}x\texttt{.}f_i\ \texttt{=}\ \mathsf{defaultValue}(T_i)\texttt{)}} \Rightarrow \widehat{\phi}\ \} \\
& \quad \text{where } \mathsf{fields}(C) = \overline{T_i\ f_i} \\
\mathsf{WLP}(\texttt{assert}\ \ \phi_a, \widehat{\phi}) & = \max\limits_{\Rightarrow}\ \{\ \widehat{\phi}' \mid \widehat{\phi}' \Rightarrow \widehat{\phi} \quad \wedge \quad \widehat{\phi}' \Rightarrow \phi_a\ \} \\
\mathsf{WLP}(y\ \texttt{:=}\ z\texttt{.}m\texttt{(}x\texttt{)}, \widehat{\phi}) & = \max\limits_{\Rightarrow}\ \{\ \widehat{\phi}' \mid\ \ y \notin \mathsf{FV}(\widehat{\phi}') \quad \wedge \\
& \widehat{\phi}' \Rightarrow \texttt{(}z\ \neq\ \texttt{null)} * \mathsf{mpre}(m)[z, x/\texttt{this}, \mathsf{mparam}(m)] \quad \wedge \\
& \widehat{\phi}' * \mathsf{mpost}(m)[z, x, y/\texttt{this}, \texttt{old(}\mathsf{mparam}(m)\texttt{)}, \texttt{result}] \Rightarrow \widehat{\phi}\ \}
\end{array}$$

Fig. 14. SVL$_{\text{IDF}}$: Weakest precondition

$$\begin{array}{ll}
\widetilde{\mathsf{WLP}}(s, \widehat{\phi}) & = \mathsf{WLP}(s, \widehat{\phi}) \quad \text{if } s \text{ is not a call statement} \\
\widetilde{\mathsf{WLP}}(\texttt{skip}, \texttt{?} * \phi) & = \texttt{?} * \phi \\
\widetilde{\mathsf{WLP}}(s_1\texttt{;}\ s_2, \texttt{?} * \phi) & = \widetilde{\mathsf{WLP}}(s_1, \widetilde{\mathsf{WLP}}(s_2, \texttt{?} * \phi)) \\
\widetilde{\mathsf{WLP}}(T\ \ x, \texttt{?} * \phi) & = \texttt{?} * \phi[\mathsf{defaultValue}(\mathsf{T})/x] \\
\widetilde{\mathsf{WLP}}(x\ \texttt{:=}\ e, \texttt{?} * \phi) & = \texttt{?} * \phi[e/x] * \texttt{(}e\ \texttt{=}\ e\texttt{)} \\
\widetilde{\mathsf{WLP}}(x\texttt{.}f\ \ \texttt{:=}\ y, \texttt{?} * \phi) & = \texttt{?} * (\phi \div \langle x, f\rangle) * \texttt{acc(}x\texttt{.}f\texttt{)} \\
& \text{if } (\phi \div \langle x, f\rangle) * \texttt{acc(}x\texttt{.}f\texttt{)} * \texttt{(}x\texttt{.}f\ \texttt{=}\ y\texttt{)} \Rightarrow \phi \\
\widetilde{\mathsf{WLP}}(x\ \ \texttt{:=}\ \texttt{new}\ C, \texttt{?} * \phi) & = \texttt{?} * \phi' \quad \text{where } \phi' = \phi \div x \\
& \text{if } \phi' * \texttt{(}x\ \neq\ \texttt{null)} * \overline{\texttt{acc(}x\texttt{.}f_i\texttt{)} * \texttt{(}x\texttt{.}f_i\ \texttt{=}\ \mathsf{defaultValue}(T_i)\texttt{)}} \Rightarrow \phi \\
\widetilde{\mathsf{WLP}}(\texttt{assert}\ \ \phi_a, \texttt{?} * \phi) & = \texttt{?} * \phi * |\phi_a|
\end{array}$$

Fig. 15. GVL$_{\text{IDF}}$: Weakest precondition implementation pointers

Note that Fig. 15 uses several helping functions: A "division" $\phi \div x$ calculates the maximum formula implied by ϕ but not containing x. This can be implemented by transitively expanding (in-)equalities and then removing conjunctive terms containing x. Similarly $\phi \div A$ calculates the maximum formula implied by ϕ that has a footprint disjoint to A.

Furthermore, there exists an accessibility-predicate-free version $|\phi|$ of any formula ϕ. Having no footprint, such a formula can be concatenated to other formulas using the separating conjunction (which then behaves like "regular" disjunction). Note however, that $|\phi|$ may not be self-framed (as soon as it names a field) and thus relies on being framed. Therefore, such formulas can only be used in conjunction with imprecise formulas, where the `?` guarantees the (re-)framing of the static part. Example:

$$|\texttt{acc(x.f) * (x.f = 3) * acc(y.f) * (b = 5)}| = \texttt{(x} \neq \texttt{y) * (x.f = 3) * (y.f = y.f) * (b = 5)}$$

As part of an imprecise formula, access to both fields is restored (through concretization). Specifically, $\texttt{?} * |\phi| = \texttt{?} * \phi$ holds for all $\phi \in \textsc{Formula}$.