

Abstract addresses: an abstraction of reusable memory locations

K. Rustan M. Leino¹[0000-0003-2872-8039] and Alexander J. Summers²[0000-0001-5554-9381]✉

¹ leino@acm.org

² alex.summers@ubc.ca, University of British Columbia

Abstract. When reasoning formally about programs that dynamically allocate memory, it is convenient to conflate the lifecycle of an object allocation with its address/identity in memory. A formal semantics in which each allocation (e.g. of an object) is imagined to have a distinct identity is convenient for formalizing languages as well as for reasoning about a variety of important program properties. This semantics, however, does not match the actual run-time behavior of many languages that reclaim and reuse memory, where physical addresses are both used in equality tests and reused when memory is freed (which is important for efficiency reasons and to minimize out-of-memory errors).

In this paper, we show how to get the best of both worlds, with at most mild restrictions. We define a formal semantics in terms of *abstract addresses*, which provides desirable properties for formal reasoning. We identify a novel (but mild) condition on acceptable programs and prove formally that, under this condition, using abstract addresses for formal reasoning is sound even when the run-time behavior of the programming language actually recycles physical addresses.

Keywords: Object references · Formal language semantics · Deallocation · Abstraction

1 Introduction

The design of a programming language crucially determines the set of programming tasks—and abstractions suitable for thinking about them—that it comfortably supports. For example, by supporting instantiable classes, a language makes it easy to dynamically create objects with identity and mutable fields, useful for simulating evolving parts of a world. By removing language implementation details from the programmer’s purview, the language also has an effect on how comfortably one reasons about programs. For example, by not surfacing the storage details of algebraic datatypes, a language gives programmers the illusion of working with complex-structured mathematical values. In this paper, we study a reasoning abstraction surrounding dynamic storage that we call *abstract addresses*, namely the illusion that every dynamic-object allocation yields a never-seen-before reference.

Our setting allows both automatically and manually managed storage, that is, garbage-collected storage and explicitly freed storage. At run time, the storage is reused, but we show that it is still sound to reason about the storage as always having new, unique references. This allows specification-only (so-called *ghost*) parts of the program to continue to use the values of objects that have been reclaimed. We include ghost code [8] in our formalism to demonstrate this distinction. Our work clears up two possible surprises:

- Ghost code need not be concerned with whether or not an object has been reclaimed (specifically, it is sound to allow updates of an object’s ghost fields even if the object has been garbage collected or manually freed), and
- When storage is allowed to be manually freed, it is necessary to restrict equality comparisons in order to soundly achieve the illusion we’re after.

We start off in the next section by highlighting how common the (usually) implicit idea of abstraction over allocation and addressing is in the literature, as well as some examples illustrating *why* this abstraction is useful and relevant for reasoning about programs. Then, to make our setting precise, we define a language in Sec. 3. We give an *abstract semantics* of this language in Sec. 4: the semantics one uses to reason about the language. In this semantics, the *references* to objects are unique, as if object storage were never reused. In Sec. 5, we give a *concrete semantics*, which more directly models what happens at run time. This concrete semantics has the properties that the *addresses* of objects may be reused and that equality (and the corresponding notion of identity on objects/pointers or similar) is based on address identity. To generalize our results, we model both explicitly freed objects and automatically garbage-collected objects in our formalism (a real language would most-often choose to include just one or the other). We tie the two semantics together in Sec. 6, where we state and prove our soundness theorem, which connects convenient reasoning in the abstract semantics with program behavior in the concrete.

We are honored to contribute this article to the Festschrift that posthumously celebrates Jean-Raymond Abrial. Abrial was a master at abstraction, always able and determined to explain problems in terms of high-level, salient properties that can be understood, only later to refine these abstractions into concrete, practical, and sometimes gritty solutions.

2 Motivation and Examples

A key motivation for our work is the prevalence of various notions of abstract addressing in the formalisms used (either implicitly or explicitly) in the programming-language community. Morisset and Harper, for example, argue that formal operational semantics for higher-level languages should intentionally abstract over a concrete addressing scheme [25], and it is standard practice for such semantics to nondeterministically abstract over how addresses are chosen. The possibility of coincidental collision (in the sense that a relevant previous address is reused) is rarely discussed, even for formal logics that reason about

lower-level languages. As an example, clean formulations of traditional separation logics do not account for the possibility that dangling pointers could be refilled by such coincidences; issues such as pointer provenance are typically out of scope in vanilla formalizations of such logics. However, if the underlying programming language allows testing for these potential coincidences, then program paths that rely on such coincidences could (perhaps unintentionally) be considered valid.

Practical verification tools for imperative languages also choose to abstract over the concrete notions of addressing and allocation in a variety of ways. For example, the `Spec#` verifier models heap locations that are never created or destroyed, but instead exist for all time; an additional *ghost field* called `alloc` is used in the underlying heap model to track objects that have been allocated [4,3]. The Dafny language has also adopted such an underlying model [19,18]. However, for both of these languages, compilation will eventually be made to program representations with concrete addresses.

In this section, we aim to give intuition for *why* these varied notions of abstraction over addresses give reasoning benefits. We also present our key insights concerning when and how this abstract reasoning is sound for languages compiled concretely to a more-standard address model.

2.1 Illustrative Example

Fig. 1 shows a concrete example in which abstract addresses are fundamentally relied upon for program reasoning, written in the Dafny programming language. In Dafny, heap memory (in this example, object fields) can be mutated; to enable callers of the method to bound its potential effects, methods come with a checked *modifies clause*, which needs to declare the objects (if any) whose fields might be modified by calling the method. A caller can then assume that any object in its context that is definitely *not* in the modifies clause cannot be changed by a call. For example, the `Incr` method is annotated to show that it changes (at most) the object passed to it as a parameter.

To make this approach more ergonomic and expressive (e.g., for objects allocated temporarily during a call), Dafny allows an object to be omitted from a modifies clause if it is allocated *during* the call itself. For example, the (rather strange) `Recycle` method need not name any objects in a modifies clause, since the only object it modifies is one it allocates.

Imagine now that we run the code `Caller`. It allocates and assigns a value to a counter `c`, passes it to `Recycle`, and then uses an `assert` statement to state a condition it expects to hold. In Dafny, as in this paper, an `assert` statement is *ghost code*—a *reasoning-only* construct that is erased in the compiled code and has no effect at run time [8].

Dafny verifies this assertion, meaning that it can prove that it holds for every possible execution of this method. However, considering the implementation of `Recycle`, this may at first glance seem to be subtly incorrect. In particular, if (as is the case for Dafny) memory were managed by garbage collection, it seems possible that the passed-but-ignored argument to `Recycle` could be reclaimed just before the new allocation happens. If so, it would be possible that the same

```

1 class Counter {
2   var val: int
3 }
4
5 method Incr(c: Counter)
6   modifies c
7   {
8     c.val := c.val + 1;
9   }
10
11 method Recycle(c: Counter) returns (d: Counter) {
12   // NOTE: since c is no longer used at run time (either
13   // here or in the caller), there is a possibility that
14   // a garbage collector could reclaim c here
15   d := new Counter;
16   d.val := 7;
17 }
18
19 method Caller() {
20   var c := new Counter;
21   c.val := 42;
22   var d := Recycle(c);
23   assert c == d ==> d.val == 42; // ghost code
24 }

```

Fig. 1. Dafny example illustrating abstract identity by allocation

memory would be reused by this allocation. In this particular case, it might be unclear why the meaning of the `asserted` condition would be true³. Indeed, the value claimed conditionally for `d.val` cannot possibly be correct, given the implementation of `Recycle`.

In fact, the reason this `assert` statement is verified is Dafny’s formal model of allocation and addresses: similarly many formalisms for program reasoning, Dafny reasons about programs using an abstract addressing scheme in which addresses are *never reused*. In particular, the meaning of the equality `c==d` in an `assert` statement is comparison of these never-reused abstract addresses, and not concrete memory addresses. Without this approach, the notion of `modifies` clauses would need substantial redesign, and would need to consider all possible aliasing that an allocation might encounter with respect to garbage-collected addresses; this would be intractable, and unpalatable for a high-level language that doesn’t allow introspection on concrete memory addresses.

Nonetheless, many languages (including some compilers for Dafny itself) will compile to a lower level of abstraction, where equality tests mean direct memory

³ The reader might be tempted to think that the presence of the `assert` statement itself would prevent garbage collection and reuse of this memory, but remember that in this paper `assert` statements do not exist at run time.

```

1  method CallerA() {
2      var c := new Counter;
3      c.val := 42;
4      var d := Recycle(c);
5      if c == d { // c is still active here, so garbage
6          // collection cannot reclaim and reuse c inside Recycle;
7          // therefore, program evaluation will never get here
8          assert d.val == 42;
9      }
10 }
11
12 method CallerB() {
13     var c := new Counter;
14     c.val := 42;
15     free c;
16     var d := Recycle(c);
17     assert c == d ==> d.val == 42;
18 }
19
20 method CallerC() {
21     var c := new Counter;
22     c.val := 42;
23     free c;
24     var d := Recycle(c);
25     if c == d { // this non-ghost comparison is disallowed by
26         // our rules
27         assert d.val == 42;
28     }
29 }

```

Fig. 2. Hypothetical code examples to illustrate subtleties explained in the text

address equality. What if we construct a variation of this example where we replace the assertion's implication with a (non-ghost) `if` statement, see `CallerA` in Fig. 2. A garbage collector would now be unable to collect `c` before this point, so the `new` inside `Recycle` would necessarily yield an address that is different than `c`, and hence the original subtlety vanishes.

Now, consider some imagined language with an explicit deallocation command, `free`. We assume that freed references can be passed as arguments, provided they are never dereferenced in executable code. In `CallerB` in Fig. 2, by freeing `c` before calling `Recycle` we reintroduce the subtlety: could we now break the illusion of the abstract addressing scheme and trick the verifier into thinking a reachable path is actually unreachable? It turns out that the answer is (if one is not careful): yes! But, a simple condition (as we show for the first time in this paper) saves the day: the abstraction will work out provided identity comparisons that can affect program behavior (i.e., outside of ghost code) are

only ever made between expressions that denote allocated objects. In particular, in variant `CallerC` of our example, the `if`-condition `c==d` would now be ruled out by our requirement (while the `assert` statement in the original code is still allowed, since it is ghost).

This restriction on equality comparisons is typically not limiting in practice: in higher-level languages, one typically avoids dangling references by design, while in lower-level languages such as C, comparison of dangling pointers has (at best) implementation-dependent behavior and is usually avoided. However, as we show in this paper, our restriction is sufficient to enable the combination of formal reasoning in terms of abstract addresses, and compilation to standard concrete address-comparison semantics without any observable mismatch.

2.2 Further Reference-Related Reasoning

Several other known techniques related to program reasoning have subtle interactions with the choice of abstract or concrete address modeling. For example, *history constraints* (popularized in JML) are specifications used to describe how an object’s fields may evolve over time, expressing properties such as “A `Counter`’s `val` field never decreases” [21,17]. Similar notions arise in Rely-Guarantee reasoning [14], invariant-based reasoning techniques [5,11], and advanced separation logics that constrain the allowed transitions on a location’s state (e.g., [7,15]).

An enforced history constraint provides guarantees for code that interacts with the same memory at multiple sequential program points *between which* unknown code (in called methods or concurrently executing threads) may have made changes. For example, considering again the `Counters` of Fig. 1 equipped with the history constraint above; knowing that a `Counter` value was *previously* 42 would tell us that even after calling unknown (but verified) code, the value of the same counter will be at least this value.

Subtly, the intent behind this reasoning also relies on a sufficiently abstract notion of object identity; if a deallocated `Counter`’s memory were subsequently reused for a newly allocated `Counter` instance, it would be incorrect to assume that (due to this coincidence of addresses chosen) the latter would have a value at least as large as the former (one would expect the reallocated instance to be reinitialized with a likely-smaller initial value). Again, for the reasoning above to be simple and sound one needs to base this on an abstract notion of address that identifies *this allocation* as the notion of identity, rather than the physical address chosen for it. It is exactly the *coincidence of addresses* that our formal restrictions on comparison operations rules out from being observable in the actual execution of such programs.

3 Language

We define a small but representative language as a target for our formal results. It includes standard imperative features, both ghost and real state and commands, and constructs for managing memory allocation:

Definition 1 (Language Syntax). *A program is a sequence of named procedures p , defined by the grammar below. We write \oplus for any of the standard boolean and arithmetic operators (including arithmetic division, \div), equality ($=$, which includes equality on references), as well as a unary ghost operator **available** meaning whether or not a given object reference has been allocated (and not explicitly freed). We assume equality and **available** are the only operators that act on reference-typed arguments, and that no results of operator expressions are of reference type (note that field-deferencing is a separate construct in our grammar):*

$p ::=$	procedure $M(xt^*)$ returns (xt^*) C	<i>procedure</i>
$xt ::=$	$x : T$	<i>variable-type pair</i>
$T ::=$	bool int ref	<i>type</i>
$C ::=$		<i>command</i>
	var xt in C	
	if e then C else C	
	C^+	<i>command sequence</i>
	assert e	<i>assertion</i>
	$x := e$	<i>assignment</i>
	$x :=$ new	<i>object allocation</i>
	free e	<i>manual object deallocation</i>
	$e.f := e$	<i>object field update</i>
	$x^* := M(e^*)$	<i>procedure call</i>
$e ::=$		<i>expression</i>
	x	<i>variable (parameter or local)</i>
	$\oplus(e^*)$	<i>operator expressions</i>
	$e.f$	<i>object field access</i>

In the grammar, we have used $*$ to indicate zero or more occurrences and $+$ to indicate one or more occurrences.

Note that a concrete syntax (for a parser) would need more structure to disambiguate, e.g. using semicolons and braces, but this abstract syntax suffices for our work. When we write a specific operator (such as \div or $=$), we use standard infix notation. We consider the literals **false** and **true**, the integer literals, and the value **null** to be nullary operators, so our abstract grammar includes them in the \oplus case as well. We assume all local variables, fields of newly-allocated objects and procedure out-parameters are zero-initialized (to **false**, 0, or **null**, depending on the type).

We assume a reasonable and standard set of static checks are performed on programs before we handle them semantically: variables used are declared (and shadowing is disallowed: all named parameters and variables must be distinct in any context); arities of procedure calls are respected; the actual out-parameters of a call are distinct variables; all expressions and commands are type-correct, etc. For simplicity, we assume a single reference type **ref**, containing only objects each with (the same) fields, and no subtyping/coercions in the type system; these restrictions could easily be lifted. Formalizing these concepts would be standard but orthogonal to the results of this paper.

3.1 Ghost State and Commands

Our language supports state (in/out-parameters, local variable, and object fields) in two *dispositions* (fixed at declaration time for each named element): each is either *real*, which means it is present at run time, or *ghost*, which means it is present for reasoning about the program, but absent at run time (in particular, it will be erased by compilation to executable code). This separation requires standard restrictions, e.g. that real state can only be assigned expressions that do not use ghost features. To simplify our formalization, we also require that *all* if-guards and variables used to allocate/free memory are real.

In our language, assert statements are always ghost code. Different specification and verification methodologies have different rules about whether and how memory inaccessible in the current program state may be referred to in assertions. We choose a very liberal rule: assert statements can refer to (fields of) objects at any time after their allocation, even if they have or may have been deallocated since. For example, the (;-delimited) sequence of commands $x := \mathbf{new} ; x.f := 3 ; \mathbf{free} x ; \mathbf{assert} x.f = 3$ is a program to which we will give a semantics (and this semantics would consider the assert statement valid for all executions of the program). We show our abstract semantics in the next section.

4 Abstract semantics

The abstract semantics defines legal program evaluations in terms of abstract reasoning (in particular, abstract addresses). To differentiate failures that the semantics is concerned with (in particular, failing proof asserts, and illegal evaluations such as division by zero and null reference exceptions), our semantics uses a distinguished *error outcome* \mathbf{err} to represent failing executions. By proving that a given program has no failing executions, from any initial state, one has proved the correctness of that program. Our results do not depend on how such proofs are constructed; there are many standard techniques (e.g., [9,12,2]).

To define the abstract semantics, we define abstract states, expression evaluation, and a big-step semantics for commands.

Definition 2 (Abstract Semantics Preliminaries). *Let \mathbf{avalue} be the union of boolean (**bool**), integer (**int**), and reference (**ref**) abstract values, the latter including a distinguished value **null**. We assume a set of fields field partitioned into ghost fields gfield and real fields rfield , and analogously variables name partitioned into gname and rname .*

An abstract semantics state St is a triple (H, A, S) of an abstract heap $H: \mathbf{ref} \rightarrow \mathit{field} \rightarrow \mathbf{avalue}$ (with $\mathbf{null} \notin \mathit{dom}(H)$), available set $A \subseteq \mathit{dom}(H)$, and abstract stack frame $S: \mathit{name} \rightarrow \mathbf{avalue}$. An abstract call stack Σ is a sequence of stack frames; we write $\Sigma \blacktriangleright S$ for the stack given by pushing S onto Σ .

Heaps are partial maps (notation \rightarrow) from references to (total) field-to-value maps. For simplicity we assume a single object type (**ref**) with a fixed set of field names, field . Values of type **ref** are *abstract addresses*; no direct correspondence

with memory locations or, e.g., address arithmetic is provided. Available sets represent memory that has been allocated and not explicitly freed. Note that abstract heaps may store information for more than the available references, in particular keeping information about objects after they have been freed.

We use boldface identifiers to represent sequences. For example, \mathbf{e} denotes a sequence of expressions. We write $|\mathbf{e}|$ to denote the length of such a sequence and write \mathbf{e}_i to denote element i of the sequence, for any index i where $0 \leq i < |\mathbf{e}|$. When quantifying over such indices, we leave the range of i implicit. In list comprehensions, map comprehensions, and map updates, we use a notation like $i \mid Q(i) \cdot \mathbf{x}_i \mapsto \mathbf{v}_i$ to combine the maplets $\mathbf{x}_i \mapsto \mathbf{v}_i$ for all i satisfying $Q(i)$; when the range $Q(i)$ is understood from context, we omit the “ $\mid Q(i)$ ”.

4.1 Abstract Expression Evaluation

We define two closely related notions of expression evaluation, called *ghost evaluation* and *real evaluation*; these differ in scope, in that expressions evaluated in a ghost context may include proof features allowed in verification-time assert commands but not in the executable language. Technically, these evaluation functions are partial in the sense that they do not account for what happens if type-incorrect expressions (such as adding two booleans) occur; however, they are total for all meaningful expressions that would pass a standard preliminary type-checking phase (which we assume but do not formalize in this paper).

Definition 3 (Dispositions and Abstract Expression Evaluation). A disposition d is either *GH* (ghost) or *RL* (real). For any operator \oplus , we write $\hat{\oplus}$ for the analogous mathematical operator. We define abstract expression evaluation as a partial function $\llbracket e \rrbracket_{St}^d$ (parameterized by d , $St = (H, A, S)$, and e), by cases:

$$\begin{aligned}
\llbracket x \rrbracket_{St}^d &= S(x) \text{ if } x \in \text{rname} \text{ or } d = \text{GH} \\
\llbracket \oplus(\mathbf{e}) \rrbracket_{St}^d &= \begin{cases} \hat{\oplus}(i \cdot \llbracket \mathbf{e}_i \rrbracket_{St}^d) & \text{if } \oplus \notin \{\mathbf{available}, \div, =\} \text{ and } \forall i \cdot \llbracket \mathbf{e}_i \rrbracket_{St}^d \neq \text{err} \\ \text{err} & \text{if } \oplus \notin \{\mathbf{available}, \div, =\} \text{ and } \exists i \cdot \llbracket \mathbf{e}_i \rrbracket_{St}^d = \text{err} \end{cases} \\
\llbracket \mathbf{available} \ e \rrbracket_{St}^{\text{GH}} &= \begin{cases} \llbracket e \rrbracket_{St}^{\text{GH}} \in A & \text{if } \llbracket e \rrbracket_{St}^{\text{GH}} \neq \text{err} \\ \text{err} & \text{otherwise} \end{cases} \\
\llbracket e_0 \div e_1 \rrbracket_{St}^d &= \begin{cases} \llbracket e_0 \rrbracket_{St}^d \hat{\div} \llbracket e_1 \rrbracket_{St}^d & \text{if } \llbracket e_0 \rrbracket_{St}^d \neq \text{err} \text{ and } \llbracket e_1 \rrbracket_{St}^d \neq \text{err}, 0 \\ \text{err} & \text{otherwise} \end{cases} \\
\llbracket e_0 = e_1 \rrbracket_{St}^{\text{RL}} &= \begin{cases} (\llbracket e_0 \rrbracket_{St}^{\text{RL}} = \llbracket e_1 \rrbracket_{St}^{\text{RL}}) & \text{if } \llbracket e_0 \rrbracket_{St}^{\text{RL}} \neq \text{err} \text{ and } \llbracket e_1 \rrbracket_{St}^{\text{RL}} \neq \text{err} \text{ and} \\ & \text{(if } e_0, e_1 \text{ of type } \mathbf{ref} \text{ then} \\ & \llbracket e_0 \rrbracket_{St}^{\text{RL}} = \mathbf{null} \text{ or } \llbracket e_1 \rrbracket_{St}^{\text{RL}} = \mathbf{null} \text{ or} \\ & \llbracket e_0 \rrbracket_{St}^{\text{RL}} \in A \wedge \llbracket e_1 \rrbracket_{St}^{\text{RL}} \in A) \\ \text{err} & \text{otherwise} \end{cases} \\
\llbracket e_0 = e_1 \rrbracket_{St}^{\text{GH}} &= \begin{cases} (\llbracket e_0 \rrbracket_{St}^{\text{GH}} = \llbracket e_1 \rrbracket_{St}^{\text{GH}}) & \text{if } \llbracket e_0 \rrbracket_{St}^{\text{GH}} \neq \text{err} \text{ and } \llbracket e_1 \rrbracket_{St}^{\text{GH}} \neq \text{err} \\ \text{err} & \text{otherwise} \end{cases} \\
\llbracket e.f \rrbracket_{St}^{\text{RL}} &= \begin{cases} H(\llbracket e \rrbracket_{St}^{\text{RL}})(f) & \text{if } f \in \text{rfield} \text{ and } \llbracket e \rrbracket_{St}^{\text{RL}} \in A \\ \text{err} & \text{if } f \in \text{rfield} \text{ and } \llbracket e \rrbracket_{St}^{\text{RL}} \notin A \end{cases} \\
\llbracket e.f \rrbracket_{St}^{\text{GH}} &= \begin{cases} H(\llbracket e \rrbracket_{St}^{\text{GH}})(f) & \text{if } \llbracket e \rrbracket_{St}^{\text{GH}} \in \text{dom}(H) \\ \text{err} & \text{otherwise} \end{cases}
\end{aligned}$$

This definition enforces that ghost variables and fields, as well as **available** expressions, can be used only in ghost contexts; real contexts can dereference real fields, but only if the object is available. Ghost contexts are allowed to compare any reference values for equality, while in real contexts we insist that these references each be available (except in the special case that one is **null**).

Definition 4 (Real-Reachable References). *The predicate $Reach(v, H, S)$, read as “reference value v is real-reachable (or just reachable for short) from (H, S) ”, is the least predicate satisfying (for all variables $x \in dom(S)$, values $v \in dom(H)$, and fields f of reference type) both:*

$$\frac{x \in rname}{Reach(S(x), H, S)} \quad \frac{Reach(v, H, S) \quad f \in rfield}{Reach(H(v)(f), H, S)}$$

Informally, this captures the idea that v can be reached by some expression $x.f.g.\dots.h$, where x and the fields are all real. The definition doesn't force reachability only through available objects, but it does capture *at least* the expressions evaluable in a real context:

Proposition 1. *For an expression e of type **ref**, if $\llbracket e \rrbracket_{(H,A,S)}^{RL} = v$ and $v \neq err$, then $Reach(v, H, S)$.*

We say a reference is *active* in (St, Σ) if it is available and reachable from a variable on *some* included stack frame:

$$IsActive(v, St, \Sigma) \triangleq v \in A \wedge \exists S' \cdot S' \in \Sigma \triangleright S \wedge Reach(v, H, S')$$

A reference in the domain of the heap that is not active is said to be *inactive*. Objects behind active references may still be used in a program, so a correctly operating garbage collector will not reclaim them.

Proposition 2. *For an expression e of type **ref**, if $\llbracket e \rrbracket_{St}^{RL} = v$ and $v \in A$, then $IsActive(v, St, \Sigma)$.*

Proof. Follows from Proposition 1 and the definition of $IsActive$.

4.2 Abstract Big-Step Command Semantics

We define the semantics of commands using inductive big-step rules. The main judgment has the form

$$St, \Sigma \vdash C \dashv \mathcal{O}$$

which says that, from abstract state St and call stack Σ , command C can lead to outcome \mathcal{O} , where the outcome is either an abstract state St' or the error **err**. Commands have no net effect on the call stack, so there is no reason for the rule to repeat Σ after the \dashv . In the following rules, like elsewhere in the paper, we use St as an abbreviation for (H, A, S) .

$$\frac{(H, A, S(x \mapsto \emptyset)), \Sigma \vdash C \dashv (H', A', S')}{\text{St}, \Sigma \vdash \mathbf{var } x: T \mathbf{in } C \dashv (H', A', S' \setminus \{x\})}$$

$$\frac{(H, A, S(x \mapsto \emptyset)), \Sigma \vdash C \dashv \text{err}}{\text{St}, \Sigma \vdash \mathbf{var } x: T \mathbf{in } C \dashv \text{err}}$$

$$\frac{\llbracket e \rrbracket_{\text{St}}^{\text{RL}} = \mathbf{true} \quad \text{St}, \Sigma \vdash C_0 \dashv \mathcal{O}}{\text{St}, \Sigma \vdash \mathbf{if } e \mathbf{ then } C_0 \mathbf{ else } C_1 \dashv \mathcal{O}} \quad \frac{\llbracket e \rrbracket_{\text{St}}^{\text{RL}} = \mathbf{false} \quad \text{St}, \Sigma \vdash C_1 \dashv \mathcal{O}}{\text{St}, \Sigma \vdash \mathbf{if } e \mathbf{ then } C_0 \mathbf{ else } C_1 \dashv \mathcal{O}}$$

$$\frac{\llbracket e \rrbracket_{\text{St}}^{\text{RL}} = \text{err}}{\text{St}, \Sigma \vdash \mathbf{if } e \mathbf{ then } C_0 \mathbf{ else } C_1 \dashv \text{err}}$$

$$\frac{1 \leq k \leq |\mathbf{C}| \quad \forall i \mid i < k \cdot \mathcal{O}_i \neq \text{err} \wedge \mathcal{O}_i, \Sigma \vdash \mathbf{C}_i \dashv \mathcal{O}_{i+1} \quad k = |\mathbf{C}| \text{ or } \mathcal{O}_k = \text{err}}{\mathcal{O}_0, \Sigma \vdash \mathbf{C} \dashv \mathcal{O}_k}$$

$$\frac{\llbracket e \rrbracket_{\text{St}}^{\text{GH}} = \mathbf{true}}{\text{St}, \Sigma \vdash \mathbf{assert } e \dashv \text{St}} \quad \frac{\llbracket e \rrbracket_{\text{St}}^{\text{GH}} \neq \mathbf{true}}{\text{St}, \Sigma \vdash \mathbf{assert } e \dashv \text{err}}$$

$$\frac{\llbracket e \rrbracket_{\text{St}}^{\text{disposition}(x)} = v \quad v \neq \text{err}}{\text{St}, \Sigma \vdash x := e \dashv (H, A, S(x \mapsto v))} \quad \frac{\llbracket e \rrbracket_{\text{St}}^{\text{disposition}(x)} = \text{err}}{\text{St}, \Sigma \vdash x := e \dashv \text{err}}$$

$$\frac{v \neq \mathbf{null} \quad v \notin \text{dom}(H)}{\text{St}, \Sigma \vdash x := \mathbf{new} \dashv (H(v \mapsto (f \cdot f \mapsto \emptyset)), A \cup \{v\}, S(x \mapsto v))}$$

$$\frac{\llbracket e \rrbracket_{\text{St}}^{\text{RL}} = v \quad v \in A}{\text{St}, \Sigma \vdash \mathbf{free } e \dashv (H, A \setminus \{v\}, S)} \quad \frac{\llbracket e \rrbracket_{\text{St}}^{\text{RL}} = v \quad v \notin A}{\text{St}, \Sigma \vdash \mathbf{free } e \dashv \text{err}}$$

$$\frac{d = \text{disposition}(f) \quad \llbracket e \rrbracket_{\text{St}}^d = v \quad \llbracket e' \rrbracket_{\text{St}}^d = v' \quad \text{if } d = \text{GH} \text{ then } v \in \text{dom}(H) \text{ else } v \in A \quad v' \neq \text{err}}{\text{St}, \Sigma \vdash e.f := e' \dashv (H(v \mapsto H(v)(f \mapsto v')), A, S)}$$

$$\frac{d = \text{disposition}(f) \quad \llbracket e \rrbracket_{\text{St}}^d = v \quad \llbracket e' \rrbracket_{\text{St}}^d = v' \quad d = \text{GH} \wedge v \notin \text{dom}(H) \text{ or } d = \text{RL} \wedge v \notin A \text{ or } v' = \text{err}}{\text{St}, \Sigma \vdash e.f := e' \dashv \text{err}}$$

$$\frac{\text{procedure } M(\mathbf{a}) \text{ returns } (\mathbf{r}) \ C \quad \forall i. \llbracket \mathbf{e}_i \rrbracket_{\text{St}}^{\text{disposition}(\mathbf{a}_i)} = \mathbf{v}_i \wedge \mathbf{v}_i \neq \text{err} \\ (H, A, (i \cdot \mathbf{a}_i \mapsto \mathbf{v}_i, j \cdot \mathbf{r}_j \mapsto \emptyset)), \Sigma \triangleright S \vdash C \dashv (H', A', S')}{\text{St}, \Sigma \vdash \mathbf{x} := M(\mathbf{e}) \dashv (H', A', S(j \cdot \mathbf{x}_j \mapsto S'(\mathbf{r}_j)))}$$

$$\frac{\text{procedure } M(\mathbf{a}) \text{ returns } (\mathbf{r}) \ C \quad \llbracket \mathbf{e}_i \rrbracket_{\text{St}}^{\text{disposition}(\mathbf{a}_i)} = \text{err}}{\text{St}, \Sigma \vdash \mathbf{x} := M(\mathbf{e}) \dashv \text{err}}$$

$$\frac{\text{procedure } M(\mathbf{a}) \text{ returns } (\mathbf{r}) \ C \quad \forall i. \llbracket \mathbf{e}_i \rrbracket_{\text{St}}^{\text{disposition}(\mathbf{a}_i)} = \mathbf{v}_i \wedge \mathbf{v}_i \neq \text{err} \\ (H, A, (i \cdot \mathbf{a}_i \mapsto \mathbf{v}_i, j \cdot \mathbf{r}_j \mapsto \emptyset)), \Sigma \triangleright S \vdash C \dashv \text{err}}{\text{St}, \Sigma \vdash \mathbf{x} := M(\mathbf{e}) \dashv \text{err}}$$

Some remarks: Local variables start off zero-initialized. The **assert** command can use ghost expressions. The **new** command creates a never-seen-before reference to an object whose fields are zero-initialized. The **free** command, which requires a real expression, does not remove the object from the heap, but instead just removes the reference from the available set. Updating a real field requires an available reference. For a procedure call, the formal parameters of the callee become local variables in the procedure body, where the formal out-parameters start off zero-initialized.

4.3 Example

As a small example, consider the program snippet in Fig. 3, call it C . Provided the type of f is **int**, for any (St, Σ) the abstract semantics rules allow us to derive $\text{St}, \Sigma \vdash C \dashv \text{St}'$ for some (non-err) St' . In particular, this means that the three assertions in this program are provable.

```

var  $x$ : ref in
var  $y$ : ref in
var  $z$ : ref in (
   $x := \text{new}$ 
   $y := \text{new}$ 
   $z := y$ 
   $x.f := 3$ 
   $y.f := 5$ 
   $z.f := 7$ 
  assert  $x.f = 3$ 
  assert  $z.f = 7$ 
  free  $y$ 
  assert  $y.f = 7$ )

```

Fig. 3. Example command. According to the abstract semantics, the conditions in the three **assert** commands hold.

5 Concrete semantics

The concrete semantics differs from the abstract semantics in three major ways: there is no ghost code (e.g., **assert** commands have no effect), distinct references are replaced by (reusable) addresses, and objects may be *removed* from the concrete heap. In more detail:

- There are no ghost parameters, ghost variables, or ghost fields. Essentially, the concrete program evaluation performed by a program is a projection of an abstract program evaluation onto its real variables. Commands that update these variables and fields are erased, and so are **assert** commands.
- In the abstract semantics, objects in the heap are identified by references that are never reused. This allows ghost contexts to continue to access fields of all objects, and there is never a possibility that **new** returns the reference of an already active or inactive object. In contrast, the concrete semantics instead uses *addresses* and allows **new** to reuse the address of an object that has been freed or has become inaccessible.
- Objects that have been explicitly freed and objects that have become inaccessible may be absent in the heap or may have been replaced by other objects. There is no available set.

We will ensure that the concrete program evaluation can proceed in an analogous way to the abstract (when the abstract has no errors), despite these differences.

Our rules for the concrete semantics also define program evaluations that can reach an error. In particular, attempts to divide by zero or dereference null will fail. Note, however, that we do not include a *run-time* assert command in our language, but it can be easily encoded if desired (by branching on a real condition and using an always-failing command like division-by-zero).

5.1 Concrete State

The concrete semantics does not track an available set, but employs three state components: the heap, the local state, and the call stack. These are projections of the counterparts of the abstract semantics onto their real variables, use addresses instead of references, and may directly reduce the domain of the concrete heap.

Definition 5 (Concrete Semantics Preliminaries). *Let constants be the union of booleans (**bool**), integers (**int**), and the value **null**. Concrete values *cvalue* are the disjoint union of constants with a set of addresses *address*.*

A concrete (semantics) state st is a pair (h, s) of a (concrete) heap $h: address \rightarrow rfield \rightarrow cvalue$ (so, $\mathbf{null} \notin dom(h)$), and concrete stack frame $s: rname \rightarrow cvalue$. A concrete call stack σ is a sequence of concrete stack frames.

In our formalism, we ignore program evaluations that run out of memory, an orthogonal issue to address abstraction. That is, we assume that *address* is large enough to always have an unused address whenever the program needs one.

Definition 6 (Concrete Expression Evaluation). *We define concrete expression evaluation as a partial function $\llbracket e \rrbracket_{st}$ (parameterized by $st = (h, s)$ and*

e), by cases:

$$\begin{aligned}
\llbracket x \rrbracket_{st} &= s(x) \text{ if } x \in \text{rname} \\
\llbracket \oplus(\mathbf{e}) \rrbracket_{st} &= \begin{cases} \hat{\oplus}(i \cdot \llbracket \mathbf{e}_i \rrbracket_{st}) & \text{if } \oplus \notin \{\mathbf{available}, \div, =\} \text{ and } \forall i \cdot \llbracket \mathbf{e}_i \rrbracket_{st} \neq \text{err} \\ \text{err} & \text{if } \oplus \notin \{\mathbf{available}, \div, =\} \text{ and } \exists i \cdot \llbracket \mathbf{e}_i \rrbracket_{st} = \text{err} \end{cases} \\
\llbracket e_0 \div e_1 \rrbracket_{st} &= \begin{cases} \llbracket e_0 \rrbracket_{st} \hat{\div} \llbracket e_1 \rrbracket_{st} & \text{if } \llbracket e_0 \rrbracket_{st} \neq \text{err} \text{ and } \llbracket e_1 \rrbracket_{st} \neq \text{err} \text{ and } \llbracket e_1 \rrbracket_{st} \neq 0 \\ \text{err} & \text{otherwise} \end{cases} \\
\llbracket e_0 = e_1 \rrbracket_{st} &= \begin{cases} (\llbracket e_0 \rrbracket_{st} = \llbracket e_1 \rrbracket_{st}) & \text{if } \llbracket e_0 \rrbracket_{st} \neq \text{err} \text{ and } \llbracket e_1 \rrbracket_{st} \neq \text{err} \\ \text{err} & \text{otherwise} \end{cases} \\
\llbracket e.f \rrbracket_{st} &= \begin{cases} h(\llbracket e \rrbracket_{st})(f) & \text{if } f \in \text{rfield} \text{ and } \llbracket e \rrbracket_{st} \in \text{dom}(h) \\ \text{err} & \text{if } f \in \text{rfield} \text{ and } \llbracket e \rrbracket_{st} \notin \text{dom}(h) \end{cases}
\end{aligned}$$

Note that this function is not defined for the evaluation of **available** expressions or for ghost variables and fields; these notions can only be used in ghost code (including assert commands). More interestingly, evaluation gives an error for expressions that attempt to access a field for an object outside the domain of the heap, as could happen if a program freed an object and then tried to access its fields.

Sometimes, these rules are more permissive than in the abstract semantics. Since concrete states only store the most recent object to use a particular address, the concrete semantics is not able to determine if a field access is for the object currently at that address or if it was intended for an object that occupied that address previously. So it is also with equality, where, in contrast to the abstract semantics, the concrete semantics always allows two addresses to be compared.

Our soundness results show that, despite these differences, any reasoning performed on a program using the abstract semantics remains valid in the concrete semantics. That is, we will show a correspondence between the abstract and concrete evaluations.

5.2 Soundness of Expression Evaluation

To state the correspondence between abstract and concrete expression evaluation, we need a mapping from references to addresses, a relation that projects out ghost names, and a mechanism that allows addresses to be reclaimed.

We write the mapping from references to addresses as $m: \mathbf{ref} \rightarrow \text{address}$, where $\text{dom}(m) = \text{dom}(H)$. It maps every reference in the domain of the abstract heap to an address. (Note that **null** is neither in the domain nor image of this mapping.) We'll need to express that a program's active references map to different addresses, but there may be overlap among the addresses to which m maps inactive references.

For any mapping $m: \mathbf{ref} \rightarrow \text{address}$, we write $\hat{m}: \text{avalue} \rightarrow \text{cvalue}$ to denote m extended with the identity function on the booleans and integers and **null**:

$$\hat{m}(v) \triangleq \begin{cases} m(v) & \text{if } v \in \mathbf{ref} \wedge v \neq \mathbf{null} \\ v & \text{if } v \in \text{constants} \end{cases}$$

For any variable maps S and s , we define $L(S, s, m)$ to say that every real variable in s is also in S and the values of these are linked by \hat{m} :

$$L(S, s, m) \triangleq \forall x \in rname \cdot x \in dom(s) \implies x \in dom(S) \wedge s(x) = \hat{m}(S(x))$$

We define the main relation (used in all of our results) between the abstract and concrete state, R , as follows, where as usual $St = (H, A, S)$ and $st = (h, s)$:

$$\begin{aligned} R(St, \Sigma, st, \sigma, m) \triangleq & \\ & (\forall v_0, v_1 \cdot IsActive(v_0, St, \Sigma) \wedge IsActive(v_1, St, \Sigma) \implies \\ & \quad v_0 \neq v_1 \implies m(v_0) \neq m(v_1)) \wedge \\ & (\forall v \cdot IsActive(v, St, \Sigma) \implies \\ & \quad m(v) \in dom(h) \wedge \forall f \in rfield \cdot h(m(v))(f) = \hat{m}(H(v)(f))) \wedge \\ & |\Sigma| = |\sigma| \wedge \\ & (\forall i \cdot L(\Sigma_i, \sigma_i, m)) \wedge \\ & L(S, s, m) \end{aligned}$$

We observe that R relates the concrete state only to the real components of the abstract state, never any of its ghosts:

Proposition 3. *Relation R does not depend on the presence or absence of ghost variables and ghost fields in the abstract semantics, nor on their values.*

We can now state a correspondence theorem for expression evaluation:

Theorem 1. *For any $St, \Sigma, st, \sigma, m, e$, if $\llbracket e \rrbracket_{st}$ is defined and $R(St, \Sigma, st, \sigma, m)$ then:*

- a) *If $\llbracket e \rrbracket_{st} = \mathbf{err}$ then $\llbracket e \rrbracket_{St}^{RL} = \mathbf{err}$, and*
- b) *If $\llbracket e \rrbracket_{st} \neq \mathbf{err}$ then either $\llbracket e \rrbracket_{St}^{RL} = \mathbf{err}$ or $\hat{m}(\llbracket e \rrbracket_{St}^{RL}) = \llbracket e \rrbracket_{st}$.*

Proof. Let $St = (H, A, S)$ and $st = (h, s)$ and let Σ, σ be arbitrary. We argue by induction on e :

- (**Case $e \equiv x$**): Then a) is trivial since \mathbf{err} is not an outcome for variables. For b) we note that by $L(S, s, m)$ we have $x \in dom(s) \implies x \in dom(S) \wedge s(x) = \hat{m}(S(x))$. Therefore we have $\llbracket e \rrbracket_{st} = s(x) = \hat{m}(S(x)) = \hat{m}(\llbracket e \rrbracket_{St}^{RL})$.
- (**Case $e \equiv \oplus(e)$**): We note that the operator cannot be **available** since in this case, $\llbracket e \rrbracket_{st}$ would not be defined. We consider cases according to Def. 6:
- (**Case $\oplus \notin \{\mathbf{available}, \div, =\}$**): According to Def. 6 there are two further sub-cases:

(**Case $\llbracket e \rrbracket_{st} = \hat{\oplus}(i \cdot \llbracket e_i \rrbracket_{st})$ and, for each i , $\llbracket e_i \rrbracket_{st} \neq \mathbf{err}$**): Then, by the induction hypothesis (for each subexpression) we obtain, for each e_i , either $\llbracket e_i \rrbracket_{St}^{RL} = \mathbf{err}$ or $\hat{m}(\llbracket e_i \rrbracket_{St}^{RL}) = \llbracket e_i \rrbracket_{st}$. Suppose that there exists some j such that $\llbracket e_j \rrbracket_{St}^{RL} = \mathbf{err}$. Then, by Def. 3, we obtain that $\llbracket e \rrbracket_{St}^{RL} = \mathbf{err}$, which suffices. Conversely, if there is no such j , then, by Def. 3 we have instead for all i that $\llbracket e_i \rrbracket_{st} = \hat{m}(\llbracket e_i \rrbracket_{St}^{RL}) = \hat{m}(\llbracket e_i \rrbracket_{St}^{RL})$ (the latter, since only the **available** and $=$ operators take reference-typed arguments). Therefore, noting also that our operator expressions do not *evaluate* to references (Def. 1), we conclude $\llbracket e \rrbracket_{st} = \hat{\oplus}(i \cdot \llbracket e_i \rrbracket_{st}) = \hat{\oplus}(i \cdot \hat{m}(\llbracket e_i \rrbracket_{St}^{RL})) = \hat{\oplus}(i \cdot \llbracket e_i \rrbracket_{St}^{RL}) = \llbracket e \rrbracket_{St}^{RL} = \hat{m}(\llbracket e \rrbracket_{St}^{RL})$ as required.

- (**Case $\llbracket e \rrbracket_{\text{st}} = \mathbf{err}$ and, for some i , $\llbracket e_i \rrbracket_{\text{st}} = \mathbf{err}$**): Then, by induction we have $\llbracket e_i \rrbracket_{\text{St}}^{\text{RL}} = \mathbf{err}$ and so $\llbracket e \rrbracket_{\text{St}}^{\text{RL}} = \mathbf{err}$ by Def. 3.
- (**Case \oplus is \div**): This case is analogous to the previous except for the specific sub-case in which $\llbracket e \rrbracket_{\text{st}} = \mathbf{err}$ because $\llbracket e_1 \rrbracket_{\text{st}} = 0$. In this case, by the induction hypothesis, either $\llbracket e_1 \rrbracket_{\text{St}}^{\text{RL}} = \mathbf{err}$ or $\hat{m}(\llbracket e_1 \rrbracket_{\text{St}}^{\text{RL}}) = 0$ (which is only possible if $\llbracket e_1 \rrbracket_{\text{St}}^{\text{RL}} = 0$). In both cases we obtain $\llbracket e \rrbracket_{\text{St}}^{\text{RL}} = \mathbf{err}$ by Def. 3.
- (**Case \oplus is $=$**): Analogous to the operator cases above in all cases where the expression or its subexpressions evaluate to \mathbf{err} (in either semantics) or when the operands are not of reference type. The interesting case remaining is when we have (after applying the induction hypothesis) two reference-typed arguments such that $\mathbf{err} \neq \llbracket e_0 \rrbracket_{\text{st}} = \hat{m}(\llbracket e_0 \rrbracket_{\text{St}}^{\text{RL}})$ and $\mathbf{err} \neq \llbracket e_1 \rrbracket_{\text{st}} = \hat{m}(\llbracket e_1 \rrbracket_{\text{St}}^{\text{RL}})$. In this case, we have $\llbracket e \rrbracket_{\text{st}}$ equal to $\hat{m}(\llbracket e_0 \rrbracket_{\text{St}}^{\text{RL}}) = \hat{m}(\llbracket e_1 \rrbracket_{\text{St}}^{\text{RL}})$, while $\hat{m}(\llbracket e \rrbracket_{\text{St}}^{\text{RL}}) = \llbracket e \rrbracket_{\text{St}}^{\text{RL}}$ (since as e is not of reference type); by Def. 3 $\llbracket e \rrbracket_{\text{St}}^{\text{RL}}$ is equal to $\llbracket e_0 \rrbracket_{\text{St}}^{\text{RL}} = \llbracket e_1 \rrbracket_{\text{St}}^{\text{RL}}$ and we must have both $\llbracket e_0 \rrbracket_{\text{St}}^{\text{RL}} \in A$ and $\llbracket e_1 \rrbracket_{\text{St}}^{\text{RL}} \in A$ (or else we would again be in an \mathbf{err} case, and conclude directly). We note that since \hat{m} is a function, $\llbracket e_0 \rrbracket_{\text{St}}^{\text{RL}} = \llbracket e_1 \rrbracket_{\text{St}}^{\text{RL}} \implies \hat{m}(\llbracket e_0 \rrbracket_{\text{St}}^{\text{RL}}) = \hat{m}(\llbracket e_1 \rrbracket_{\text{St}}^{\text{RL}})$; to show equality we must conversely prove that $\hat{m}(\llbracket e_0 \rrbracket_{\text{St}}^{\text{RL}}) = \hat{m}(\llbracket e_1 \rrbracket_{\text{St}}^{\text{RL}}) \implies \llbracket e_0 \rrbracket_{\text{St}}^{\text{RL}} = \llbracket e_1 \rrbracket_{\text{St}}^{\text{RL}}$. We note further that $\llbracket e_0 \rrbracket_{\text{st}} = \mathbf{null}$ if and only if $\llbracket e_0 \rrbracket_{\text{St}}^{\text{RL}} = \mathbf{null}$, since m does not have \mathbf{null} in its image; analogously, $\llbracket e_1 \rrbracket_{\text{st}} = \mathbf{null}$ if and only if $\llbracket e_1 \rrbracket_{\text{St}}^{\text{RL}} = \mathbf{null}$. Therefore, if either subexpression evaluates to \mathbf{null} , we have $\hat{m}(\llbracket e_0 \rrbracket_{\text{St}}^{\text{RL}}) = \hat{m}(\llbracket e_1 \rrbracket_{\text{St}}^{\text{RL}})$ if and only if $\llbracket e_0 \rrbracket_{\text{St}}^{\text{RL}} = \llbracket e_1 \rrbracket_{\text{St}}^{\text{RL}}$ as required. If instead, neither evaluates to \mathbf{null} , we use Proposition 2 to deduce $IsActive(\llbracket e_0 \rrbracket_{\text{St}}^{\text{RL}}, \text{St}, \Sigma)$ and $IsActive(\llbracket e_1 \rrbracket_{\text{St}}^{\text{RL}}, \text{St}, \Sigma)$. From the definition of R , we therefore obtain $\llbracket e_0 \rrbracket_{\text{St}}^{\text{RL}} \neq \llbracket e_1 \rrbracket_{\text{St}}^{\text{RL}} \implies \hat{m}(\llbracket e_0 \rrbracket_{\text{St}}^{\text{RL}}) \neq \hat{m}(\llbracket e_1 \rrbracket_{\text{St}}^{\text{RL}})$ which is the contrapositive of our required implication.
- (**Case $e \equiv e'.f$**): By assumption, we must have $f \in rfield$. We have two sub-cases:
- (**Case $\llbracket e \rrbracket_{\text{st}} = h(\llbracket e' \rrbracket_{\text{st}})(f)$ and $\llbracket e' \rrbracket_{\text{st}} \in dom(h)$**): By the induction hypothesis, either $\llbracket e' \rrbracket_{\text{St}}^{\text{RL}} = \mathbf{err}$ (in which case we conclude as in previous cases) or $\hat{m}(\llbracket e' \rrbracket_{\text{St}}^{\text{RL}}) = \llbracket e' \rrbracket_{\text{st}}$. Suppose $\llbracket e' \rrbracket_{\text{St}}^{\text{RL}} \notin A$: then we obtain $\llbracket e \rrbracket_{\text{St}}^{\text{RL}} = \mathbf{err}$. Otherwise, by Proposition 2 we have $IsActive(\llbracket e' \rrbracket_{\text{St}}^{\text{RL}}, \text{St}, \Sigma)$ and so by the definition of R we obtain $h(m(\llbracket e' \rrbracket_{\text{St}}^{\text{RL}}))(f) = \hat{m}(H(\llbracket e' \rrbracket_{\text{St}}^{\text{RL}})(f))$. Thus, $\llbracket e \rrbracket_{\text{st}} = h(\llbracket e' \rrbracket_{\text{st}})(f) = \hat{m}(\llbracket e \rrbracket_{\text{St}}^{\text{RL}})$ as required.
- (**Case $\llbracket e \rrbracket_{\text{st}} = \mathbf{err}$ and $\llbracket e' \rrbracket_{\text{st}} \notin dom(h)$**): We show $\llbracket e \rrbracket_{\text{St}}^{\text{RL}} = \mathbf{err}$ by contradiction: suppose instead (cf. Def. 6) that $\mathbf{err} \neq \llbracket e' \rrbracket_{\text{St}}^{\text{RL}} \in A$. We have $\hat{m}(\llbracket e' \rrbracket_{\text{St}}^{\text{RL}}) = \llbracket e' \rrbracket_{\text{st}}$ by our induction hypothesis. But by Proposition 2 we obtain $IsActive(\llbracket e' \rrbracket_{\text{St}}^{\text{RL}}, \text{St}, \Sigma)$ and so (by definition of R) we have $\hat{m}(\llbracket e' \rrbracket_{\text{St}}^{\text{RL}}) \in dom(h)$, contradicting the information from our case distinction.

5.3 Concrete Big-Step Command Semantics

We define the concrete semantics of commands using inductive big-step rules. These rules more closely mimic what happens at run time. The main judgment has the form

$$\text{st}, \sigma \Vdash C \dashv\vdash \mathcal{R}$$

which says that, from concrete state st and call stack σ , command C can lead to outcome \mathcal{R} , where the outcome is either a concrete state st' or the error err . Since commands have no net effect on the call stack, we don't repeat σ after the $\dashv\vdash$.

In the following rules, like elsewhere in the paper, we use st as an abbreviation for (h, s) . In addition to rules for each command, there is a rule for garbage collection. Here are the rules for the commands:

$$\frac{x \in \text{gname} \quad \text{st}, \sigma \Vdash C \dashv\vdash \mathcal{R}}{\text{st}, \sigma \Vdash \mathbf{var} \ x: T \ \mathbf{in} \ C \dashv\vdash \mathcal{R}} \quad \frac{x \in \text{rname} \quad (h, s(x \mapsto \emptyset)), \sigma \Vdash C \dashv\vdash \text{err}}{\text{st}, \sigma \Vdash \mathbf{var} \ x: T \ \mathbf{in} \ C \dashv\vdash \text{err}}$$

$$\frac{x \in \text{rname} \quad (h, s(x \mapsto \emptyset)), \sigma \Vdash C \dashv\vdash (h', s')}{\text{st}, \sigma \Vdash \mathbf{var} \ x: T \ \mathbf{in} \ C \dashv\vdash (h', s' \setminus \{x\})}$$

$$\frac{\llbracket e \rrbracket_{\text{st}} = \mathbf{true} \quad \text{st}, \sigma \Vdash C_0 \dashv\vdash \mathcal{R}}{\text{st}, \sigma \Vdash \mathbf{if} \ e \ \mathbf{then} \ C_0 \ \mathbf{else} \ C_1 \dashv\vdash \mathcal{R}} \quad \frac{\llbracket e \rrbracket_{\text{st}} = \mathbf{false} \quad \text{st}, \sigma \Vdash C_1 \dashv\vdash \mathcal{R}}{\text{st}, \sigma \Vdash \mathbf{if} \ e \ \mathbf{then} \ C_0 \ \mathbf{else} \ C_1 \dashv\vdash \mathcal{R}}$$

$$\frac{\llbracket e \rrbracket_{\text{st}} = \text{err}}{\text{st}, \sigma \Vdash \mathbf{if} \ e \ \mathbf{then} \ C_0 \ \mathbf{else} \ C_1 \dashv\vdash \text{err}}$$

$$\frac{1 \leq k \leq |\mathbf{C}| \quad \forall i \mid i < k \cdot \mathcal{R}_i \neq \text{err} \wedge \mathcal{R}_i, \sigma \Vdash \mathbf{C}_i \dashv\vdash \mathcal{R}_{i+1} \quad k = |\mathbf{C}| \ \text{or} \ \mathcal{R}_k = \text{err}}{\mathcal{R}_0, \sigma \Vdash \mathbf{C} \dashv\vdash \mathcal{R}_k}$$

$$\frac{}{\text{st}, \sigma \Vdash \mathbf{assert} \ e \dashv\vdash \text{st}} \quad \frac{x \in \text{gname}}{\text{st}, \sigma \Vdash x := e \dashv\vdash \text{st}}$$

$$\frac{x \in \text{rname} \quad \llbracket e \rrbracket_{\text{st}} = v \quad v \neq \text{err}}{\text{st}, \sigma \Vdash x := e \dashv\vdash (h, s(x \mapsto v))} \quad \frac{x \in \text{rname} \quad \llbracket e \rrbracket_{\text{st}} = \text{err}}{\text{st}, \sigma \Vdash x := e \dashv\vdash \text{err}}$$

$$\frac{v \neq \mathbf{null} \quad v \notin \text{dom}(h)}{\text{st}, \sigma \Vdash x := \mathbf{new} \dashv\vdash (h(v \mapsto (f \cdot f \mapsto \emptyset)), s(x \mapsto v))}$$

$$\frac{\llbracket e \rrbracket_{\text{st}} = v \quad v \neq \text{err}}{\text{st}, \sigma \Vdash \mathbf{free} \ e \dashv\vdash (h \setminus \{v\}, s)} \quad \frac{\llbracket e \rrbracket_{\text{st}} = v \quad v \neq \text{err}}{\text{st}, \sigma \Vdash \mathbf{free} \ e \dashv\vdash \text{st}} \quad \frac{\llbracket e \rrbracket_{\text{st}} = \text{err}}{\text{st}, \sigma \Vdash \mathbf{free} \ e \dashv\vdash \text{err}}$$

$$\frac{f \in gfield}{st, \sigma \Vdash e.f := e' \dashv\vdash st}$$

$$\frac{f \in rfield \quad \llbracket e \rrbracket_{st} = v \quad \llbracket e' \rrbracket_{st} = v' \quad v \in dom(h) \quad v' \neq err}{st, \sigma \Vdash e.f := e' \dashv\vdash (h(v \mapsto h(v))(f \mapsto v'), s)}$$

$$\frac{f \in rfield \quad \llbracket e \rrbracket_{st} = v \quad \llbracket e' \rrbracket_{st} = v' \quad v \notin dom(h) \text{ or } v' = err}{st, \sigma \Vdash e.f := e' \dashv\vdash err}$$

$$\frac{\text{procedure } M(\mathbf{a}) \text{ returns } (\mathbf{r}) C \quad \forall i \mid \mathbf{a}_i \in rname \cdot \llbracket \mathbf{e}_i \rrbracket_{st} = \mathbf{v}_i \wedge \mathbf{v}_i \neq err \\ (h, (i \mid \mathbf{a}_i \in rname \cdot \mathbf{a}_i \mapsto \mathbf{v}_i, j \mid \mathbf{r}_j \in rname \cdot \mathbf{r}_j \mapsto \emptyset)), \sigma \triangleright s \Vdash C \dashv\vdash (h', s')}{st, \sigma \Vdash \mathbf{x} := M(\mathbf{e}) \dashv\vdash (h', s(j \mid \mathbf{r}_j \in rname \cdot \mathbf{x}_j \mapsto s'(\mathbf{r}_j)))}$$

$$\frac{\text{procedure } M(\mathbf{a}) \text{ returns } (\mathbf{r}) C \quad \mathbf{a}_i \in rname \quad \llbracket \mathbf{e}_i \rrbracket_{st} = err}{st, \sigma \Vdash \mathbf{x} := M(\mathbf{e}) \dashv\vdash err}$$

$$\frac{\text{procedure } M(\mathbf{a}) \text{ returns } (\mathbf{r}) C \quad \forall i \mid \mathbf{a}_i \in rname \cdot \llbracket \mathbf{e}_i \rrbracket_{st} = \mathbf{v}_i \wedge \mathbf{v}_i \neq err \\ (h, (i \mid \mathbf{a}_i \in rname \cdot \mathbf{a}_i \mapsto \mathbf{v}_i, j \mid \mathbf{r}_j \in rname \cdot \mathbf{r}_j \mapsto \emptyset)), \sigma \triangleright s \Vdash C \dashv\vdash err}{st, \sigma \Vdash \mathbf{x} := M(\mathbf{e}) \dashv\vdash err}$$

Any ghost variables declared are ignored by the concrete semantics, and so are assignments to ghost variables and ghost fields. This also includes ghost parameters. Since **assert** commands are ghost, they too are ignored.

The **new** command can pick any address v that is not currently in use. This v may have been used previously during the program evaluation and then discarded by the **free** command or by garbage collection.

We allow two possible, legal evaluations of the **free** command. One immediately removes the object from the heap, whereas the other does nothing. The reason for allowing the second evaluation is to allow the language to always rely on garbage collection or to perform some kind of batched deallocation. Thus, in the concrete semantics, the **free** command is just a hint to the runtime system. In either case, the **free** command does not make any other changes, so if the address of the object being freed is stored in various variables or fields (which includes e), then there is a risk that the rest of the program may use dangling pointers. Our soundness theorem will show that any program without failing abstract evaluations has no failing concrete evaluations, either.

The rule for legally updating a real field has a premise $v \in dom(h)$. If this condition does not hold, the concrete semantics says the command evaluation should result in the outcome **err**. This was also the case for the field-access

expressions in Sec. 5.1. At run time, however, determining $v \in \text{dom}(h)$ with every field access and field update can be prohibitively expensive. Still, we argue that our rules are useful, because a compiler may in practice claim to follow the concrete semantics only for non-error cases. Such a compiler would omit a run-time check for $v \in \text{dom}(h)$ and would emit code under the assumption that the condition holds. If a program has been shown to be free of failures in the abstract semantics, then our theorem in the next section says the program is also free of failures in the concrete semantics, and thus the code that a practical compiler claims to emit for that program would still operate correctly according to both the concrete and abstract semantics.

We include one final rule in the concrete semantics. This rule performs garbage collection and is allowed to “fire” at any time (between commands) during program evaluation. We first define what it means for a set K to be *closed* under a heap h :

$$\text{IsClosed}(K, h) \triangleq \forall r, f. r \in K \wedge r \in \text{dom}(h) \implies h(r)(f) \in K \cup \text{constants}$$

Garbage collection is now described by the following rule:

$$\frac{\forall s'. s' \in \sigma \triangleright s \implies \text{image}(s') \subseteq K \quad \text{IsClosed}(K, h) \quad (h \downarrow K, s), \sigma \Vdash C \dashv\vdash \mathcal{R}}{\text{st}, \sigma \Vdash C \dashv\vdash \mathcal{R}}$$

This rule allows the domain of the heap to be restricted (notation \downarrow) to any closed set K that contains all roots, that is, the addresses stored in local variables anywhere on the call stack. As we remarked in Sec. 4.1, for garbage collection to be correct, it must preserve active objects. We’ll see that in our soundness theorems in the next section.

6 Soundness

To show the use of abstract addresses to be sound, we state and prove a theorem. Roughly speaking, the theorem shows that abstract and concrete evaluations correspond via relation R , which we defined in Sec. 5.2. In more detail, the theorem says that, for any corresponding abstract and concrete states,

- If the concrete semantics for can lead to an error, then there is an abstract trace that leads to an error, and
- If the concrete semantics can lead to a state st' , then either there is an abstract trace that leads to an abstract state corresponding to st' or there is an abstract trace leading to an error.

In either case, this means that someone reasoning in terms of the abstract semantics will not be surprised by the concrete trace.

Here is the theorem in full detail:

Theorem 2. Consider any $St, \Sigma, st, \sigma, m, C, \mathcal{R}$, such that

$$R(St, \Sigma, st, \sigma, m) \text{ and } (h, \sigma), s \Vdash C \dashv\vdash \mathcal{R},$$

Then:

- If $\mathcal{R} = \text{err}$, then $St, \Sigma \vdash C \dashv\vdash \text{err}$.
- If $\mathcal{R} = st'$ for some (non-error) st' , then there is an outcome \mathcal{O} such that $St, \Sigma \vdash C \dashv\vdash \mathcal{O}$ and either
 - $\mathcal{O} = \text{err}$, or
 - there is some m' such that $R(\mathcal{O}, \Sigma, st', \sigma, m')$.

Proof. We proceed by induction over the derivation of the concrete-semantics judgment.

- For **var** $x: T$ **in** C' where $x \in \text{gname}$, the concrete-semantics judgment was derived from

$$(h, \sigma), s \Vdash C \dashv\vdash \mathcal{R},$$

On account of Proposition 3, we have

$$R((H, A, S(x \mapsto \emptyset)), \Sigma, h, s, \sigma, m)$$

If $\mathcal{R} = \text{err}$, we thus obtain by the induction hypothesis (IH),

$$(H, A, S(x \mapsto \emptyset)), \Sigma \vdash C' \dashv\vdash \text{err}$$

which, by the abstract-semantics rule, gives us our proof goal. If $\mathcal{R} = (h', s')$ for some h', s' , the IH gives us

$$(H, A, S(x \mapsto \emptyset)), \Sigma \vdash C' \dashv\vdash (H', A', S')$$

for some H', A', S' and m' . By Proposition 3, we have

$$R((H', A', S' \setminus \{x\}), \Sigma, h', s', \sigma, m')$$

which, by the abstract-semantics rule, gives us our proof goal.

- For **var** $x: T$ **in** C' where $x \in \text{rname}$, the proof is similar to the previous case, but the two R conditions in the proof are obtained directly from the definition of R .
- For the conditional command, regardless of if $\llbracket e \rrbracket_{h,s}$ returns **true**, **false**, or an error, we have $\llbracket e \rrbracket_{h,s} = \llbracket e \rrbracket_{\text{St}}^{\text{RL}}$ by Theorem 1, so our proof goal follows directly from the IH.
- For sequential composition, if one of the sub-commands gives outcome **err** in the concrete semantics, then by the IH, the corresponding sub-command gives **err** in the abstract semantics. For the sub-commands that give a non-error concrete outcome, the IH either gives us an abstract error or a corresponding abstract state.
- For **assert** e , if $\llbracket e \rrbracket_{\text{St}}^{\text{GH}} = \text{true}$, then our proof goal follows directly from the IH. Otherwise, we have $St, \Sigma \vdash C \dashv\vdash \text{err}$, which also satisfies our proof goal.

- The various cases for $x := e$ are similar to those of the **var** command, the main difference being that assignment also evaluates e (instead of setting the new variable to \emptyset). Theorem 1 takes care of the rest.
- For **new**, we need (for the first time in this proof so far) to construct a value of m' , rather than just pass along some given map. Let α denote the address v picked by the concrete-semantics rule and let r denote the reference v picked by the abstract-semantics rule. We'll then set m' to be $m(r \mapsto \alpha)$.
- For **free** e , if e concretely evaluates to an error, then by Theorem 1, e abstractly evaluates to an error as well, and since $\text{err} \notin A$, the abstract semantics for **free** e leads to an error, so our proof goal is met. Otherwise, let α be what e evaluates to concretely. By Theorem 1, e may still abstractly evaluate to an error, in which case our proof goal is met, as we just argued. Otherwise, let r be what e evaluates to abstractly, so by Theorem 1, we have $\alpha = m(r)$. Having already handled the situations where the abstract semantics of **free** e leads to an error, we thus can now assume $r \in A$. We're now going to argue that $R(\mathcal{O}', \Sigma, \text{st}', \sigma, m)$ holds (that is, we're choosing $m' = m$), since we're given that $R(\mathcal{O}, \Sigma, \text{st}, \sigma, m)$ holds. The last three conjuncts of R are unchanged, so they hold for $R(\mathcal{O}', \Sigma, \text{st}', \sigma, m)$. The first conjunct holds as well, since no new active references are introduced and the second line of the quantified formula does not change. Note that according to the first conjunct of R , r is the only active reference that m maps to α . For the second conjunct, the second line of the quantified formula can change only for an address that has been removed from $\text{dom}(h)$, which can only be α .
The abstract-semantics rule removes r from the available set ($A' = A \setminus \{r\}$), rendering r inactive. Thus, relation R in our proof goal does not constrain h' in what, if anything, it maps r to. In particular, this means our proof goal is met regardless of which of the remaining concrete-semantics rules is used.
- The case for field update requires case-splitting on the evaluation cases for its two proper sub-expressions (whether each evaluates to an error). These follow from arguments analogous to those we have presented in previous cases of the proof.
- The case for procedure call similarly follows straightforwardly, using the definition of R , Proposition 3, and the IH.
- Finally, garbage collection. This case follows from the IH, provided we can prove

$$R(\text{St}, \Sigma, h \downarrow K, s, \sigma, m)$$

Only the second conjunct of R may be affected, since only it speaks about $\text{dom}(h)$. What remains to be proved now follows from $\text{IsActive}(v, \text{St}, \Sigma) \implies m(v) \in K$.

In summary, this theorem justifies that one can reason about the possible behaviors of a program (and in particular, prove that it cannot ever reach errors) entirely in our abstract semantics, where, for reasoning convenience one need not consider recycling of addresses. This, alongside our requirement that the program is also proved not to compare possibly-freed references, is sufficient to know that the program can be compiled and executed concretely with a semantics that uses

physical addresses (in particular, in comparisons), and no errors can possibly have been missed by reasoning in the abstract world.

7 Related and Future Work

We mentioned various related work as motivation in Sec. 2, although we are not aware of similar formal work directly tackling the discrepancy between abstract references and concrete addressing. For example, in Hoare’s early paper on the semantics of Pascal [13], calls to **new** return consecutive integers. While this yields distinct addresses, the mechanism is not abstract enough to allow a different selection of concrete addresses at run time. Also, this semantics did not allow addresses to be reclaimed and reused.

There is a long history of proofs related to memory management, for example of garbage collection schemes (e.g., [6,10]); these formally prove when, in a concrete setting, it is safe to deallocate memory. This doesn’t, however, directly connect to the question of when it is sound to reason about objects as if they did not reuse this same memory.

Our soundness argument is structured similarly to a classic refinement proof: we ultimately show that the behaviors possible in the concrete semantics are included among those in the abstract semantics. This kind of refinement proof is closely related to Abrial’s B method proofs [1], and to the refinement arguments in TLA [16]. Our R relation serves the role of a refinement relation; in contrast to the B method (but in line with e.g. compiler correctness arguments [22,24,23,20]) we only show inclusion of behaviors from concrete to abstract. As future work, one could also prove that programs with abstract behaviors have at least *some* concrete behavior, although we believe this to be intuitively evident from the simulation shown in the detail of our presented proofs.

As future work, we would be interested to extend our formal result to multi-state properties such as history invariants or “old” expressions; these are needed in a specification language to explicitly relate memory values at multiple program points. In our current formalism, this relation is partially and indirectly enabled by the fact that our abstract semantics allows objects to be referred to in ghost contexts even after their deallocation. It would also be interesting to explore the extension of these results to languages with more-complex specification operators concerning memory, for example, by allowing address arithmetic of some kind.

8 Conclusion

We have defined an abstract semantics that uses references to abstract over addresses. This semantics avoids reasoning case-splits by giving the user a brand new reference with each **new** object allocation. We proved that this illusion is sound even if the concrete addresses are reclaimed and reused. Our formalization allows both manual deallocation and automatic garbage collection and is also applicable to the many programming systems that support only one or the other of these. A standard practice when reasoning about code is to use auxiliary

variables, aka ghost variables, which are erased by the compiler and absent in the concrete semantics. Our treatment allows such ghosts, and our soundness theorem shows that ghost code can continue to access ghost fields of objects even after the objects have been freed. This makes our results applicable to a large number of automated verification tools for programs that allocate memory.

References

1. Abrial, J.R.: The B-book: assigning programs to meanings. Cambridge University Press, USA (1996)
2. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification — The KeY Book — From Theory to Practice, Lecture Notes in Computer Science, vol. 10001. Springer (2016). <https://doi.org/10.1007/978-3-319-49812-6>
3. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005. Lecture Notes in Computer Science, vol. 4111, pp. 364–387. Springer (Sep 2006)
4. Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H.: Specification and verification: The Spec# experience. *Communications of the ACM* **54**(6), 81–91 (Jun 2011)
5. Cohen, E., Moskal, M., Schulte, W., Tobies, S.: Local verification of global invariants in concurrent programs. In: Touili, T., Cook, B., Jackson, P. (eds.) Computer Aided Verification. pp. 480–494. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
6. Dijkstra, E.W., Lamport, L., Martin, A.J., Scholten, C.S., Steffens, E.F.M.: On-the-fly garbage collection: an exercise in cooperation. *Communications of the ACM* **21**(11), 966–975 (Nov 1978). <https://doi.org/10.1145/359642.359655>, <https://doi.org/10.1145/359642.359655>
7. Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M.J., Vafeiadis, V.: Concurrent abstract predicates. In: D’Hondt, T. (ed.) ECOOP 2010 – Object-Oriented Programming. pp. 504–528. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
8. Filliâtre, J.C., Gondelman, L., Paskevich, A.: The spirit of ghost code. *Form. Methods Syst. Des.* **48**(3), 152–174 (Jun 2016). <https://doi.org/10.1007/s10703-016-0243-x>, <https://doi.org/10.1007/s10703-016-0243-x>
9. Floyd, R.W.: Assigning meanings to programs. *Proceedings Symposium on Applied Mathematics* **19**, 19–31 (1967)
10. Hawblitzel, C., Petrank, E.: Automated verification of practical garbage collectors. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 441–453. POPL ’09, Association for Computing Machinery, New York, NY, USA (2009). <https://doi.org/10.1145/1480881.1480935>, <https://doi.org/10.1145/1480881.1480935>
11. Hawblitzel, C., Qadeer, S., Tasiran, S.: Automated and modular refinement reasoning for concurrent programs. Tech. Rep. MSR-TR-2015-8, Microsoft Research (Feb 2015), <https://www.microsoft.com/en-us/research/publication/automated-and-modular-refinement-reasoning-for-concurrent-programs/>
12. Hoare, C.A.R.: An axiomatic basis for computer programming. *Communications of the ACM* **12**(10), 576–580,583 (Oct 1969)

13. Hoare, C.A.R.: An axiomatic definition of the programming language PASCAL. In: Ershov, A.P., Nepomniaschy, V.A. (eds.) International Symposium on Theoretical Programming, Novosibirsk, Russia. Lecture Notes in Computer Science, vol. 5, pp. 1–16. Springer (1972). https://doi.org/10.1007/3-540-06720-5_1, https://doi.org/10.1007/3-540-06720-5_1
14. Jones, C.B.: Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.* **5**(4), 596–619 (Oct 1983). <https://doi.org/10.1145/69575.69577>, <https://doi.org/10.1145/69575.69577>
15. Jung, R., Swasey, D., Sieczkowski, F., Svendsen, K., Turon, A., Birkedal, L., Dreyer, D.: Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 637–650. POPL '15, Association for Computing Machinery, New York, NY, USA (2015). <https://doi.org/10.1145/2676726.2676980>, <https://doi.org/10.1145/2676726.2676980>
16. Lampert, L.: The temporal logic of actions. *ACM Trans. Program. Lang. Syst.* **16**(3), 872–923 (May 1994). <https://doi.org/10.1145/177492.177726>, <https://doi.org/10.1145/177492.177726>
17. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes* **31**(3), 1–38 (2006)
18. Leino, K.R.M.: Specification and verification of object-oriented software. In: Broy, M., Sitou, W., Hoare, T. (eds.) Engineering Methods and Tools for Software Safety and Security, NATO Science for Peace and Security Series D: Information and Communication Security, vol. 22, pp. 231–266. IOS Press (2009), summer School Marktoberdorf 2008 lecture notes
19. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16. Lecture Notes in Computer Science, vol. 6355, pp. 348–370. Springer (Apr 2010). https://doi.org/10.1007/978-3-642-17511-4_20
20. Leroy, X.: Formal verification of a realistic compiler. *Communications of the ACM* **52**(7), 107–115 (Jul 2009). <https://doi.org/10.1145/1538788.1538814>, <https://doi.org/10.1145/1538788.1538814>
21. Liskov, B.H., Wing, J.M.: A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.* **16**(6), 1811–1841 (Nov 1994). <https://doi.org/10.1145/197320.197383>, <https://doi.org/10.1145/197320.197383>
22. McCarthy, J., Painter, J.A.: Correctness of a compiler for arithmetic expressions. In: Mathematical Aspects of Computer Science: Proceedings of Symposia in Applied Mathematics. Proceedings of Symposia in Applied Mathematics, vol. 19, pp. 33–41. American Mathematical Society, Providence, RI, USA (1967), <http://jmc.stanford.edu/articles/mcpain/mcpain.pdf>, reprint of the original paper
23. Milner, R., Weyhrauch, R.W.: Proving compiler correctness in a mechanised logic. *Machine Intelligence* **7**, 51–73 (1972)
24. Morris, F.L.: Advice on structuring compilers and proving them correct. In: Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. p. 144–152. POPL '73, Association for Computing Machinery, New York, NY, USA (1973). <https://doi.org/10.1145/512927.512941>, <https://doi.org/10.1145/512927.512941>
25. Morrisett, G., Harper, R.: Semantics of memory management for polymorphic languages, p. 175–226. Cambridge University Press, USA (1999)