A Formal Semantics for Isorecursive and Equirecursive State Abstractions

Alexander J. Summers[1] and Sophia Drossopoulou[2]

[1] ETH Zurich alexander.summers@inf.ethz.ch
[2] Imperial College London s.drossopoulou@imperial.ac.uk

Abstract. Methodologies for static program verification and analysis often support recursive predicates in specifications, in order to reason about recursive data structures. Intuitively, a predicate instance represents the complete unrolling of its definition; this is the *equirecursive* interpretation. However, this semantics is unsuitable for static verification, when the recursion becomes unbounded. For this reason, most static verifiers differentiate between, e.g., a predicate instance and its corresponding body, while providing a facility to map between the two; this is the *isorecursive* semantics. While this latter interpretation is usually implemented in practice, only the equirecursive semantics is typically treated in theoretical work.
In this paper, we provide both an isorecursive and an equirecursive formal semantics for recursive definitions in the context of Chalice, a verification methodology based on implicit dynamic frames. We show that development of such formalisations requires addressing several subtle issues, such as the possibility of infinitely-recursive definitions and the need for the isorecursive semantics to correctly reflect the restrictions that make it readily implementable. These questions are made more challenging still in the context of implicit dynamic frames, where the use of heap-dependent expressions provides further pitfalls for a correct formal treatment.

1 Introduction

Recursive definitions of a program's state, are widely employed in techniques for program specification, verification and static analysis. Common techniques include recursive predicates, pure methods, abstraction functions and model fields. The ability to express recursion in specifications is needed to describe programs which themselves manipulate recursively-defined data structures, since it is impossible for a specification to explicitly describe each of the locations involved when accessing the structure. For example, a method which computes the sum of the values in a linked-list will need to access a statically-unbounded number of heap locations to do so. To solve this specification problem in the context of permission-based methodologies such as separation logic, *recursive abstract predicates* [15] were introduced. Predicate definitions can be provided as part of a program's specification, and the meaning of a predicate name is defined in terms

of an assertion (the predicate *body*), which may itself include instances of the same predicate. In this way, it is possible for a predicate instance to implicitly require permission to access, e.g., every *next* field in a linked-list. The intuitive meaning of such a predicate symbol, is that it represents everything implied by the (recursive) unrolling of its definition; this is the *equirecursive interpretation* of the recursive definition [1].

However, static verifiers (and other tools) cannot predictably reason directly in terms of an equirecursive semantics, since at verification time it is impossible to know when to stop unrolling such a definition. For this reason, many verifiers make use of ghost *fold* and *unfold* operations for handling recursively-defined predicates, which explicitly exchange a predicate name for its body (or vice versa). These operations may be explicitly provided in the source code (e.g., [9, 11]), implicitly specified via heuristic rules (e.g., [5]), or tools may try to infer them by other means (e.g., [2]); their eventual role is the same. In the absence of *fold* and *unfold* operations, the information implied by the unrolling of a currently-held predicate instance is not made available to the verifier. Such a treatment of recursive definitions differentiates between holding a predicate instance and holding its body, while providing a means by which the one can be explicitly exchanged for the other; this is the *isorecursive interpretation*.

The critical aspect of an isorecursive semantics is that it can be used as the basis for building static tools, while the equirecursive semantics cannot (without the undesirable possibility of potentially infinitely applying recursive definitions, in a so-called *matching loop* [12]). Nonetheless, an equirecursive semantics is much closer to an intuitive runtime model for a methodology, and theoretical papers which formalise verification logics typically treat recursive definitions in this natural way [16]. This creates a mismatch between the formalised assertion logic semantics, and that typically implemented in tools; one of the aims of this paper is to address this mismatch.

Abstraction functions provide a different mechanism for expressing properties of recursive structures. Function definitions can traverse data structures, and return abstract values which summarise the contents in ways which abstract over the underlying representation. Pure methods, as used in specification methodologies such as Eiffel, JML and Spec♯ play a similar role, as do model fields. Formalising such functions in a specification language requires care, since a function which is not well-defined (e.g., $bad() = bad() + 1$) can easily lead to inconsistency in the logic; this issue is complicated by the fact that many useful heap-dependent functions do not always have an obvious termination measure expressible in terms of their arguments. Indeed, a normal *length()* function will typically not terminate for cyclic list structures. Furthermore, the unrolling of function definitions also needs careful control for a static tool to handle them practically.

In this paper, we investigate the isorecursive and equirecursive semantics of recursive specification constructs. Concretely, we base our work on the *implicit dynamic frames (IDF)* specification logic of Smans [21]. This logic has been recently shown [17, 18] to have close connections with separation logic,

however, it has the advantage for us of including both recursive predicates and heap-dependent abstraction functions, as well as the ability to express *unfolding expressions* which explicitly "peek" inside recursive definitions; the combination of these features making the work presented both more challenging and more general. The recursive aspects of IDF have not been given a direct assertion semantics before; we give both isorecursive semantics (suitable as the basis for a verifier) and equirecursive semantics (suitable for comparison with a runtime model, and for proving soundness). We extend both assertion semantics to corresponding Hoare Logics, based on a subset of the Chalice programming language [10], and discuss how our isorecursive model lends itself to implementation, and the related possibility of isorecursive states not having a "real" equirecursive counterpart. We define mappings from the isorecursive model to the equirecursive, and show how the various corresponding concepts are formally related. Finally, we define a novel interleaving operational semantics for our language, and prove soundness of our Hoare Logics.

While we work in the context of IDF, the issues arising regarding recursive definitions are much more generic, and the discussions and solutions presented here can easily be adapted for the formalisation of approaches based on e.g., separation logic, and are relevant for the construction of soundness arguments for other techniques such as decision procedures and static analyses for recursive definitions. One of the goals of this work is to identify and elaborate on the challenges which arise, in order to help other researchers facing them. The development of such formalisations requires addressing several subtle issues, regarding both the possibility of infinitely-recursive definitions and the need for the isorecursive semantics to correctly reflect the restrictions that make it readily implementable. The mismatch between the intuitive (equirecursive) semantics and that implemented in tools can lead to pitfalls in practice; for example, a prototype implementation of recursive definitions in the Chalice verifier was unsound for this reason; a correct solution has only recently been proposed [7].

Contributions The contributions of this paper are:

- An equirecursive semantics for IDF expressions and assertions, including recursive functions and predicates. This is the first direct assertion semantics for IDF which handles recursive definitions.
- An isorecursive semantics for IDF expressions and assertions; to our knowledge, this is the first such assertion semantics which reflects the distinction between holding a predicate and knowing its body.
- Hoare logics for both approaches; in particular, the isorecursive Hoare Logic includes novel rules for folding and unfolding predicates, and tracking associated information with unfolding expressions.
- Encodings and results which formally relate the two semantics, connecting that used at verification time with that used in soundness proofs.
- A novel operational semantics for Chalice, and a soundness result showing that (isorecursive) verification guarantees runtime soundness. This is the first soundness proof for the Chalice approach including recursive definitions.

2 Equirecursive Semantics for Predicates and Functions

In this section we define the syntax and semantics of expressions and assertions in the equi-recursive setting. Our treatment is based on the work of Parkinson and Summers [17]. Their work did not include any kind of predicates or functions in the assertion language; we address these issues here.

2.1 Implicit Dynamic Frames

Implicit dynamic frames allow expressions e which depend on the heap, e.g., assertion $x.f.g = \mathsf{this}$. In order to make the meaning of such assertions *robust* to interference from other threads, a notion of permission is employed. Special assertions $\mathbf{acc}(e.f)$ called *accessibility predicates* denote a permission to access the heap location $e.f$, at most one of which is present in the system at any one time. Assertions used in specifications must be *self-framing*, which means they must have permissions to all heap locations that they dereference in expressions. For example, the assertion $x.f.g = \mathsf{this}$ is not self-framing, but $\mathbf{acc}(x.f) * \mathbf{acc}(x.f.g) * x.f.g = \mathsf{this}$ is. The separating conjunction $*$ is related to that of separation logic; it acts just as logical conjunction, but behaves *multiplicatively* with respect to accessibility predicates; that is, $\mathbf{acc}(x.f) * \mathbf{acc}(\mathsf{this}.f)$ requires permission to both locations *separately*, and so its meaning implicitly guarantees that this and x cannot be aliases.

Defining a formal assertion semantics for implicit dynamic frames is challenging. Parkinson and Summers [17] defined a semantics for a core of the logic, and, among other questions, addressed the question of the semantics of assertions which are *not* self-framing. For example, what should be the meaning of $x.f.g = \mathsf{this}$? In a state which does not hold permissions to the two heap locations, evaluation of this expression depends on the other threads, and so giving it a deterministic semantics seems incorrect. However, the difficulty is that a compositional definition of the semantics of assertions cannot "see" whether the appropriate permissions to $x.f$ and $x.f.g$ are held by the current thread. The solution used in [17] is, to give the expression the semantics it should have *assuming* the appropriate permissions are held; i.e., read from the heap regardless. Since all assertions are additionally checked to be self-framing, in the end this means that the above semantics is only applied in the situation in which it makes sense. As we will show in subsection 2.3, similar issues arise when adding recursive definitions to the logic, their treatment needs to be different.

2.2 Recursive Predicates and Functions

Implicit dynamic frames supports two kinds of recursive definitions in assertions. In this paper, we use the terminology of the Chalice tool [10], and call them *predicates* and *functions*. Predicates can be defined recursively, and their bodies are assertions. Allowing specifications to mention predicates as well as field permissions and boolean expressions, makes it possible for, e.g., a precondition to require all permissions to a recursive data structure. For example,

a predicate *List* storing all permissions in a linked list could be defined by: $List \equiv \mathbf{acc}(\mathsf{this}.next) * \mathbf{acc}(\mathsf{this}.val) * (next \neq null \rightarrow next.List)$

Implicit dynamic frames also supports recursively-defined *functions* as part of the expression syntax. For example, the assertions $\mathsf{this}.length() = 4$ and $\mathsf{this}.itemAt(2) = 0$ use functions to expose additional information about the internals of a list. Functions typically correspond to the "pure methods" of an implementation[3]. Termination of function definitions is essential for soundness, once they are permitted in assertions. For example, a definition $bad() = 1 + bad()$ would allow the verifier to deduce inconsistency wherever this definition was made available.

2.3 Handling Infinite Recursion

The introduction of recursively-defined functions and predicates opens up the potential of non-termination when evaluating assertions. What should be the semantics of a predicate instance whose definition can be unrolled infinitely? And what should be done with function definitions that do not terminate? It is tempting to say that definitions which *may* not terminate should be forbidden. But this would be too restrictive: for example, even though the linked list predicate *List* from the previous section does not terminate in a heap in which $\mathsf{this} = \mathsf{this}.next$, such predicate definitions are essential for traversing recursive data structures, and cannot be dispensed with. Similarly, a function $length() = (\mathsf{this}.next = null\,?\,1 : 1 + next.length())$does not terminate in the case that $\mathsf{this} = \mathsf{this}.next$. Indeed, there is not even any obvious termination measure in terms of the function's signature that could be used to prove termination of this function definition. Nonetheless, such functions cannot be dispensed with either.

For the equirecursive setting, in which an idealised mathematical semantics is appropriate, an elegant (and reasonably standard) way of handling custom predicate definitions is to make the semantics of infinitely-unrollable predicate instances *false*. We take this approach; that is, we interpret predicate definitions by their least fixed points. In this way, we build into the logic the implicit assumption that all predicate *instances* have finite definitions. Forbidding infinite predicate instances does not harm our expressiveness in practice, since, as will be explained in the next section, such predicate instances could never be obtained in a verifier based on an isorecursive semantics. Thus, any program point at which an infinite predicate instance is required (for example, in a method precondition of the form $List * \mathsf{this} = \mathsf{this}.next$) is actually unreachable code, and thus it is operationally consistent to assign *false* to such assertions.

Now consider the semantics of potentially-non-terminating functions. The example of the *length* function makes clear that we must admit function definitions which do not necessarily terminate in *all* states. This means that any assertion semantics we define needs to cope with the possibility of evaluating a function call whose naïve semantics would cause undefinedness. For example, consider

[3] Indeed, this is the terminology used in [21].

the assertion $List * length() = 3$. In the case where $\mathsf{this} = \mathsf{this}.next$ holds, *List* will be false (due to the least fix-point treatment of predicates), and we would like the overall assertion should also mean *false*. But a naïve definition for expression semantics might give the conjunct in which the function call occurs an ill-defined meaning. To avoid the need for relying on a short-cutting semantics for conjunctions, we define an expression semantics that is *total*, even for naturally non-terminating function calls. We achieve this by the introduction of *error values*, which are dummy values used in place of a non-terminating expression evaluation. Since the overall assertion will be *false* whenever these error values occur, this means that we implement the natural semantics for function calls in all situations where the meaning matters.

2.4 Syntax

Definition 1 (Expressions and Assertions). *We define the syntax of* equi-expressions *(ranged over by e) and* equi-assertions *(ranged over by a)as follows:*

$$\begin{array}{rcl} e & ::= & null \mid true \mid false \mid x \mid e.f \mid e.g(e) \mid e = e \mid (\, e \,?\, e : e) \\ a & ::= & e \mid \mathbf{acc}(e.f, q) \mid e \rightarrow a \mid a * a \mid e.P \mid Thread(x, m, y, z) \end{array}$$

In the above, x, y, z range over program variables, f over field identifiers, g over function names, P over predicate names, m over method names, and q over rational numbers in $(0, 1]$. There are three special reserved variable names*: this (representing the current receiver), X (representing the current method parameter) and method (representing name of the current method); these may not be used explicitly in expressions, and their role will be explained shortly.*

The $Thread(x, m, y, z)$ assertion is used to record information about other threads currently running, as will be explained in the next subsection. We implicitly require expressions and assertions to be type-correct, *e.g.* in $e.f$ the type of e should have a field f. Other connectives over equi-expressions, such as $\wedge$, $\vee$ and $\neg$ can be encoded. Note that the implication connective $\rightarrow$ is restricted to only allow *expressions* (rather than assertions) on the left-hand side. This restriction is common to most practical verification tools based on separation logic or implicit dynamic frames; it makes it possible to avoid an assertion semantics which needs to quantify over states (see e.g., [14, 17]); a problematic feature for automatic verification. Similarly, negation is only encodable for boolean expressions. Thus, $\mathbf{acc}(\mathsf{this}.f, 1) * \mathsf{this}.f = 5$, and $this.f = 5 \rightarrow \mathbf{acc}(\mathsf{this}.f, 1)$ are assertions according to our definitions, while $\mathbf{acc}(\mathsf{this}.f, 1) \rightarrow \mathsf{this}.f = 5$ is not.

2.5 Semantics

As in [17], the semantics of assertions is defined in terms of permission masks π, heaps H, and environments σ. As usual, we need a $*$ operator to combine permissions.

Definition 2 (Preliminaries).
We assume a set of values consisting of at least true, false, null, object identifiers (ranged over by ι), thread identifiers (ranged over by t), method names (ranged over by m), and one distinct error value per type tp (denoted by $error_{tp}$).
Environments*, ranged over by σ, are maps from variables to values.*
Heaps*, ranged over by H, are maps from pairs of either object identifier and field name or thread identifier and field name, to values.*
Equi-permission-masks*, ranged over by π, are maps from pairs of either object identifier and field name or thread identifier and field name, to nonnegative values in $\mathbb{Q}$.*
We define the operator $+$ to combine permissions as follows:
$+ : perms \times perms \to perms$ *and* $(\pi + \pi')(\iota, f) = \pi(\iota, f) + \pi'(\iota, f)$.
An equi-permission-mask π is well-formed*, written $\models \pi$, if its range is within $[0..1]$:* $\models \pi$ *iff* $\forall \iota, f.\ \pi(\iota, f) \in [0..1]$
The lookup function Body returns the definition of a function or a predicate. Predicates have the implicit parameter this*, and functions have the implicit parameters* this*, and* X.

An unusual feature above is the inclusion of field locations (in heaps and permission masks) with thread identifiers as receiver. This is in order to permit assertions which track information about other threads; we use three *ghost fields*, called `recv`, `param`, `meth`, which are used to record the receiver, parameter and current method name for a given thread identifier. These fields (which are the only fields defined for a thread identifier) are ghost in the sense that they are not present in runtime heaps (see Section 7).

A further novelty in definition 2 is the error value, $error_{tp}$, which we motivated earlier; it is used to define the value of expressions when their evaluation is infinite:

Definition 3 (Value of Equi-Expressions).
We define evaluation *of expressions in a state consisting of heap H, and environment σ, through predicate $\_ \Downarrow_{H,\sigma} \_$ as follows :*

$x \Downarrow_{H,\sigma} \sigma(x)$ $\quad$ $null \Downarrow_{H,\sigma} null$ $\quad$ $true \Downarrow_{H,\sigma} true$ $\quad$ $false \Downarrow_{H,\sigma} false$

$e.f \Downarrow_{H,\sigma} v$ $\quad$ *if* $e \Downarrow_{H,\sigma} \iota$ *and* $H(\iota, f) = v$.

$e.g(e') \Downarrow_{H,\sigma} v$ $\quad$ *if* $e \Downarrow_{H,\sigma} \iota$ *and* $e' \Downarrow_{H,\sigma} v'$ *and* $Body(g) \Downarrow_{H,\sigma'} v$, *where* $\sigma' = [\mathsf{this}, X \mapsto \iota, v']$.

$e = e' \Downarrow_{H,\sigma} true$ $\quad$ *if* $e \Downarrow_{H,\sigma} v$ *and* $e' \Downarrow_{H,\sigma} v$ *for some* v.

$e = e' \Downarrow_{H,\sigma} false$ $\quad$ *if* $e \Downarrow_{H,\sigma} v$ *and* $e' \Downarrow_{H,\sigma} v'$ *and* $v \neq v'$.

$(e_1 \,?\, e_2 : e_3) \Downarrow_{H,\sigma} v$ $\quad$ *if* $e_1 \Downarrow_{H,\sigma} true$ *and* $e_2 \Downarrow_{H,\sigma} v$.

$(e_1 \,?\, e_2 : e_3) \Downarrow_{H,\sigma} v$ $\quad$ *if* $e_1 \Downarrow_{H,\sigma} false$ *and* $e_3 \Downarrow_{H,\sigma} v$.

We now define the value *of an expression e in the context of H and σ, as follows*:

$$\llbracket e \rrbracket_{H,\sigma} = \begin{cases} v & \text{if } e \Downarrow_{H,\sigma} v \\ error_{tp} & \text{if } \nexists v.\ e \Downarrow_{H,\sigma} v \text{ and } e \text{ has type } tp. \end{cases}$$

Thus, in a configuration H_0, σ_0 where z points to a cycle, we obtain that $\llbracket z.length() \rrbracket_{H_0,\sigma_0} = error_{int}$, and $\llbracket z.length() = 3 \rrbracket_{H_0,\sigma_0} = error_{bool}$. Moreover, we can represent $z.length() \neq 3$ through $(\, z.length() = 3 \,?\, false : true)$ and then we

would obtain $[\![z.length() \neq 3]\!]_{H_0,\sigma_0} = error_{bool}$. The presence of error values in the above definition may seem surprising, but we will shortly show that such values can only occur when the meaning of the expression is irrelevant to the assertion in which it occurs. First, we define the semantics of equi-recursive assertions:

Definition 4 (Semantics of Equi-Assertions). *We define the semantics of assertions in a state comprising of permissions π, heaps H, and environment σ, as the least fixpoint of the following equations:*

$$
\begin{array}{ll}
\pi, H, \sigma \models_{\mathsf{E}} e & \iff [\![e]\!]_{H,\sigma} = true \\
\pi, H, \sigma \models_{\mathsf{E}} \mathbf{acc}(e.f, q) & \iff \pi([\![e]\!]_{H,\sigma}, f) \geq q \\
\pi, H, \sigma \models_{\mathsf{E}} e \to a & \iff [\![e]\!]_{H,\sigma} \Rightarrow \pi, H, \sigma \models_{\mathsf{E}} a \\
\pi, H, \sigma \models_{\mathsf{E}} a_1 * a_2 & \iff \exists \pi_1, \pi_2 : \pi = \pi_1 + \pi_2 \\
 & \quad \mathit{and}\ \pi_1, H, \sigma \models_{\mathsf{E}} a_1\ \mathit{and}\ \pi_2, H, \sigma \models_{\mathsf{E}} a_2 \\
\pi, H, \sigma \models_{\mathsf{E}} e.P & \iff \pi, H, \sigma \models_{\mathsf{E}} \mathit{Body}(P)[e/\mathsf{this}] \\
\pi, H, \sigma \models_{\mathsf{E}} \mathit{Thread}(x, m, y, z) & \iff \pi[\sigma(x), \mathtt{recv}] = 1 \wedge H[\sigma(x), \mathtt{recv}] = \sigma(y) \\
 & \quad \wedge\ \pi[\sigma(x), \mathtt{param}] = 1 \wedge H[\sigma(x), \mathtt{param}] = \sigma(z) \\
 & \quad \wedge\ \pi[\sigma(x), \mathtt{meth}] = 1 \wedge H[\sigma(x).\mathtt{meth}] = m
\end{array}
$$

Equi-entailment $a \models_{\mathsf{E}} a'$ holds if: $\forall \pi, H, \sigma.\ \pi, H, \sigma \models_{\mathsf{E}} a \implies \pi, H, \sigma \models_{\mathsf{E}} a'$.

Thus, in H_0, σ_0 from above we have $\pi, H_0, \sigma_0 \not\models_{\mathsf{E}} z.length() = 3$. And if, as before, we represent $z.length() \neq 3$ through $(z.length() = 3\,?\,false : true)$ we obtain $\pi, H_0, \sigma_0 \not\models_{\mathsf{E}} z.length() \neq 3$. Note that $\pi, H, \sigma \models_{\mathsf{E}} e.P$ holds, if $\pi, H, \sigma \models_{\mathsf{E}} \mathit{BodyP}[e/\mathsf{this}]$ holds in a *finite* unfolding. Therefore, we also obtain $\pi, H_0, \sigma_0 \not\models_{\mathsf{E}} z.Acyclic$.

Moreover, if we represent $\neg(z.length() = 3)$ through $(z.length() = 3) \to false$, we obtain that $\pi, H_0, \sigma_0 \models_{\mathsf{E}} \neg(z.length() = 3)$. This may seem to be a concern, given that $\pi, H_0, \sigma_0 \not\models_{\mathsf{E}} z.length() \neq 3$.

These concerns are eliminated once we add the definition of *framed* expressions and assertions. Essentially, we define a judgement which ensures for a given expression or assertion that a particular state holds enough permissions to access all fields, and that all involved function calls, or predicate applications have a finite unrolling. We then define an assertion to be *self-framing* if it can only hold in states in which it is framed. In this way, we guarantee that all assertions are either trivially false, or else their semantics will not include calculation of error values, and thus the intuitive semantics of expressions is restored.[4]

Definition 5 (Framed and Self-Framing Equi-Assertions). *An equi-expression e, or assertion a is* equi-framed *in a state consisting of H, π and σ, as the least*

[4] The concept of self-framing assertion was introduced in [21], and described algorithmically in [22]; a more abstract formalisation for assertions not including predicates was developed in [17].

fixpoint satisfying the following equations:

$\models^{\pi,H,\sigma}_{\mathsf{frmE}} \mathit{null} \quad \models^{\pi,H,\sigma}_{\mathsf{frmE}} \mathit{true} \quad \models^{\pi,H,\sigma}_{\mathsf{frmE}} \mathit{false} \quad \models^{\pi,H,\sigma}_{\mathsf{frmE}} x$

$\models^{\pi,H,\sigma}_{\mathsf{frmE}} e.f \iff \models^{\pi,H,\sigma}_{\mathsf{frmE}} e \ \wedge \ \pi(\llbracket e \rrbracket_{H,\sigma}, f) > 0$

$\models^{\pi,H,\sigma}_{\mathsf{frmE}} e.g(e') \iff \models^{\pi,H,\sigma}_{\mathsf{frmE}} e \wedge \models^{\pi,H,\sigma}_{\mathsf{frmE}} e' \wedge \models^{\pi,H,\sigma'}_{\mathsf{frmE}} \mathit{Body}(g)$
where $\sigma' = [\mathsf{this}, X \mapsto \llbracket e \rrbracket_{H,\sigma}, \llbracket e' \rrbracket_{H,\sigma}]$

$\models^{\pi,H,\sigma}_{\mathsf{frmE}} e = e' \iff \models^{\pi,H,\sigma}_{\mathsf{frmE}} e \wedge \models^{\pi,H,\sigma}_{\mathsf{frmE}} e'$

$\models^{\pi,H,\sigma}_{\mathsf{frmE}} e1?e2 : e3 \iff \models^{\pi,H,\sigma}_{\mathsf{frmE}} e1 \ \wedge \ (\pi, H, \sigma \models_{\mathsf{E}} e2 \Rightarrow \models^{\pi,H,\sigma}_{\mathsf{frmE}} e3)$
$\wedge \ (\pi, H, \sigma \not\models_{\mathsf{E}} e1 \ \Rightarrow \models^{\pi,H,\sigma}_{\mathsf{frmE}} e2)$

$\models^{\pi,H,\sigma}_{\mathsf{frmE}} \mathbf{acc}(e.f, q) \iff \models^{\pi,H,\sigma}_{\mathsf{frmE}} e$

$\models^{\pi,H,\sigma}_{\mathsf{frmE}} e \to a \iff \models^{\pi,H,\sigma}_{\mathsf{frmE}} e \ \wedge \ (\pi, H, \sigma \models_{\mathsf{E}} e \Rightarrow \models^{\pi,H,\sigma}_{\mathsf{frmE}} a)$

$\models^{\pi,H,\sigma}_{\mathsf{frmE}} a_1 * a_2 \iff \models^{\pi,H,\sigma}_{\mathsf{frmE}} a_1 \wedge \models^{\pi,H,\sigma}_{\mathsf{frmE}} a_2$

$\models^{\pi,H,\sigma}_{\mathsf{frmE}} e.P \iff \models^{\pi,H,\sigma}_{\mathsf{frmE}} e \wedge \models^{\pi,H,\sigma}_{\mathsf{frmE}} \mathit{Body}(P)[e/\mathsf{this}]$

$\models^{\pi,H,\sigma}_{\mathsf{frmE}} \mathit{Thread}(x, m, y, z) \iff \mathit{true}$

An equi-expression e is framed by an assertion a, *written* $a \models_{\mathsf{frmE}} e$, *if, (for all* π, H, σ*) we have* $\pi, H, \sigma \models_{\mathsf{E}} a$ *implies that* $\models^{\Pi,H,\sigma}_{\mathsf{frmE}} e$.

An equi-assertion a is self-framing, *written* $\models_{\mathsf{frmE}} a$ *if, for all* H, π *and* σ*:*

$\pi, H, \sigma \models_{\mathsf{E}} a \ \Rightarrow \ \models^{\pi,H,\sigma}_{\mathsf{frmE}} a$

Thus, $x = 3$ is always framed, and thus self-framing trivially, and so are $\mathbf{acc}(e.f, q)$ and $\mathit{Thread}(x, m, y, z)$, while $x.\mathit{next} \neq \mathit{null}$ is framed only when the state holds permission to the heap location $x.\mathit{next}$. Moreover, note that the expressions $x.\mathit{next} = x.\mathit{next}$ and $x.\mathit{next} \neq x.\mathit{next}$ (encoded through $(x.\mathit{next} = x.\mathit{next})?\mathit{false} : \mathit{true}$ as earlier), and the assertion $\neg(x.\mathit{next} = x.\mathit{next})$ (encoded through $((x.\mathit{next} = x.\mathit{next}) \to \mathit{false})$ as earlier) are *not* self-framing. Similarly, $x.\mathit{length}()$ is framed only in states where we hold the permissions to all the fields in the list, and also, where x is an acyclic list. In particular, $x.\mathit{List} * x.\mathit{length}() == 4$ is self-framing.

The definitions provided in this section give a direct semantics to well-defined recursive functions and predicates, in which function calls are always equal to their bodies (when this yields a well-defined value), and predicates are also evaluated in terms of their bodies directly. This semantics is unsuitable for usage in an implementation for three reasons:

1. Treating a predicate instance as meaning its full unrolling yields an unimplementable semantics for checking truth and entailment of assertions; the unbounded unrolling of such a definition cannot be performed in a static tool.
2. Treating a function call as always equal to its body also naturally leads to unbounded instantiation of the recursive definition, in order to evaluate the meaning of assertions in which the function is called.
3. Checking well-definedness and framing of predicate and function definitions is also not practical, since (according to Definition 5), this also depends on being able to evaluate the entire unrolling of the definitions.

In the next section, we turn to the corresponding isorecursive notions, to address these issues.

Discussion In the remainder of this section, we reflect on some considered design alternatives.

In earlier attempts to define a suitable semantics, we considered making the evaluation of expressions (particularly functions) dependent on holding sufficient permissions to *frame* the expressions. This seems intuitive, since reading heap locations without permissions can, at runtime, yield an undefined result (due to race conditions), while evaluating non-terminating functions is clearly undefined at runtime. However, once expression semantics is allowed to be undefined, one either needs to allow the assertion semantics to also provide undefined results, or to provide rules for when to promote undefined results to true or false. The use of additional error values $error_{tp}$ allowed us to avoid this; instead taking an "optimistic" semantics of expressions, and then enforcing framedness separately.

Verification tools often require abstraction functions to have a *precondition*, which is a predicate which guarantees that the function body terminates, and that the context of a call will hold sufficient permissions. This is natural for the modelling of partial functions, however, we found that our equirecursive semantics could exploit the use of error values (which implicitly make all functions total) to avoid explicit preconditions. This is consistent with an optimistic expression semantics, and it is not the semantics of this section which needs to be readily checkable in static tools. Preconditions for functions will appear in the isorecursive semantics of the next section.

3 Isorecursive Semantics for Predicates and Functions

In this section, we introduce an assertion semantics in the iso-recursive style. In the iso-recursive approach, predicates are differentiated from their bodies; this is handled in the logic by treating predicate names merely as another kind of permission, which can be rewritten into the corresponding body of the predicate by explicit extra *fold* and *unfold* statements in the program. With this approach, there are then two different notions of permission a thread can have; the *explicit* permissions, which can be represented in a permissions masks as usual, and the *implicit* permissions, which are those which are folded (perhaps recursively) inside of predicates held in the explicit permissions. Moreover, in the iso-world, functions are equipped with preconditions, which control when the function may be called; thus, the precondition ensures that the body of the function is well-defined in the particular configuration. To differentiate isorecursive definitions from their equirecursive counterparts, we typically use corresponding upper-case metavariables (Π for masks, E for expressions, etc.).

Definition 6 (Iso-Expressions and Iso-Assertions). *We define the syntax of* iso-expressions *(ranged over by E) and* iso-assertions *(ranged over by A), by the following grammars, in which x ranges over program variables, f ranges over field identifiers, g over function names, P over predicate identifiers, and q ranges over rational numbers in the range* $(0, 1]$*:*

$$E \;::=\; \mathit{null} \mid \mathit{true} \mid \mathit{false} \mid x \mid E.f \mid E.g(E) \mid E = E \mid (\,E\,?\,E : E) \mid \mathbf{unfolding}\, E.P\, \mathbf{in}\, E$$
$$A \;::=\; E \mid \mathbf{acc}(E.f, q) \mid E \to A \mid A * A \mid \mathbf{acc}(E.P) \mid \mathit{Thread}(x, m, y, z)$$

In addition to the lookup function Body from the equi-programs, we also require a lookup function Pre which is given the name of of an iso-function and returns an assertion, which is the precondition *of a function.*

To continue our example from earlier, a length function in the iso-setting, *length*, would come with a precondition, $\mathit{Pre}(\mathit{length}) = \mathbf{acc}(\mathsf{this}.\mathit{List})$, where the predicate *List* would be defined through:

$\mathbf{acc}(\mathsf{this}.\mathit{next}, 1) * ((\,\mathsf{this}.\mathit{next} = \mathit{null}\,?\,\mathit{true} : \mathbf{acc}(\mathsf{this}.\mathit{next}.\mathit{List})))$,

and where *Body*(*length*) would be

$\mathbf{unfolding}\, \mathsf{this}.\mathit{List}\, \mathbf{in}\, (\,\mathsf{this}.\mathit{next} = \mathit{null}\,?\,0 : 1 + \mathsf{this}.\mathit{next}.\mathit{length}())$.

The syntax of iso-expressions is the same as that of equi-expressions, with the exception of $\mathbf{unfolding}\, E_1.P\, \mathbf{in}\, E_2$ which does not exist in the equirecursive setting. The value of this expression is the same as that of E_2, however, the way such an expression is checked to be *framed* in a state is different; the unfolding of the predicate P means that E_2 may depend on permissions which come from the body of $E_1.P$.

The difference between equi-assertions and iso-assertions is the replacement of the assertion of predicate $e.P$, by the *permissions* to the predicate, denoted $\mathbf{acc}(E.P)$ above. The semantics of $e.P$ differs from that of $\mathbf{acc}(E.P)$, in that the former unfolds all recursive definitions, while the latter only requires permission to the predicate instance - more in definition 9. To keep the presentation simple, we only support *full* permissions on predicates; i.e., we do not allow predicate instances themselves to be "split". Some tools support this, and the corresponding extension of our model would be straightforward.

Definition 7 (Semantics of Iso-Expressions). *We define the evaluation of iso-expressions E in a state comprising of heaps H, and environment σ, in the analogous manner to definition 3, for example,*

$E = E' \Downarrow_{H,\sigma} \mathit{false}$ *if* $E \Downarrow_{H,\sigma} v$ *and* $E \Downarrow_{H,\sigma} v'$ *and* $v \neq v'$.

with the following additional case:

$\mathbf{unfolding}\, E_1.P\, \mathbf{in}\, E_2 \Downarrow_{H,\sigma} v$ *if* $E_2 \Downarrow_{H,\sigma} v$.

Moreover, as in definition 3, if E has type t, then

$[\![E]\!]_{H,\sigma} = v$, *if* $E \Downarrow_{H,\sigma} v$, *and* error_{tp}, *otherwise.*

The full definition appears in the appendix.

Thus, in a state H_1, σ_1 where z points to an acyclic list of two elements, we would have $[\![z.\mathit{Length}]\!]_{H_1,\sigma_1}$=2, and in H_0, σ_0 from earlier, we would have $[\![z.\mathit{Length}]\!]_{H_0,\sigma_0} = \mathit{error}_{int}$.

In order to model iso-assertions, we extend the concept of permission mask, so that it also holds permissions to predicate instances.

Definition 8 (Iso-Permissions and Permissions Collection). Isorecursive permission masks, *$\Pi \in \mathit{Perms}$, are mappings from pairs of object or thread identifiers and field names to non-negative values in $\mathbb{Q}$, and from pairs of object identifiers and predicate identifiers to non-negative values in $\mathbb{Z}$.*

The function $\mathcal{P}_{\mathsf{I}}$ *collects the permissions* explicitly *required by an assertion,*

$$\mathcal{P}_{\mathsf{I}} \;:\; \mathit{IsoAssertion} \times \mathit{Heap} \times \mathit{Env} \to \mathit{Perms}$$

$$\begin{array}{lcl}
\mathcal{P}_{\mathsf{I}}(E, H, \sigma) & = & \emptyset \\
\mathcal{P}_{\mathsf{I}}(\mathbf{acc}(E.f, q), H, \sigma) & = & \{\ (\llbracket E \rrbracket_{H,\sigma}, f) \mapsto q\ \} \\
\mathcal{P}_{\mathsf{I}}(E \to A, H, \sigma) & = & \mathcal{P}_{\mathsf{I}}(A, H, \sigma) \text{ if } \llbracket E \rrbracket_{H,\sigma} = \mathit{true}, \quad \emptyset \text{ otherwise} \\
\mathcal{P}_{\mathsf{I}}(A * A', H, \sigma) & = & \mathcal{P}_{\mathsf{I}}(A, H, \sigma) + \mathcal{P}_{\mathsf{I}}(A', H, \sigma) \\
\mathcal{P}_{\mathsf{I}}(\mathbf{acc}(E.P), H, \sigma) & = & \{\ (\llbracket E \rrbracket_{H,\sigma}, P) \mapsto 1\ \} \\
\mathcal{P}_{\mathsf{I}}(\mathit{Thread}(x, m, y, z), H, \sigma) & = & \{\ (\sigma(x), \mathtt{recv}) \mapsto 1, (\sigma(x), \mathtt{param}) \mapsto 1, \\
 & & \quad (\sigma(x), \mathtt{meth}) \mapsto 1\}
\end{array}$$

The operation $\odot$*, applied to permission mask* Π*, address* ι *and predicate identifier* P *in a heap* H*, is defined only when* $\Pi(\sigma(x), \mathtt{meth}]) \geq 1$*, and removes the permission to* $\iota.P$ *from the* Π*, and adds all the permissions obtained by unfolding the predicate body once.*

$$\Pi \odot_H \iota.P = \ \mapsto \Pi(\iota, P) - 1] + \mathcal{P}_{\mathsf{I}}(\mathit{Body}(P), H, \sigma), \quad \text{where} \ \ \sigma(\mathsf{this}) = \iota.$$

As for the equirecursive case, we have a liberal treatment of permission masks Π, which allows permissions to fields be *any* rational numbers, even if they exceed 1. Because we do not work with full knowledge of the permissions contained within predicate instances, we cannot in general rule out the possibility that the assertions we handle implicitly require more than the full permission to a field. In other words, there is always the possibility for isorecursive permission masks to have no corresponding well-formed equirecursive permission mask. For uniformity, therefore, we ignore this issue in our isorecursive model, and address it in the following section.

Definition 9 (Semantics of Iso-Assertions). *We define the semantics of iso-assertions in a state comprising of an iso-permission mask* Π*, heap* H*, and an environment* σ*, as the smallest fixed point satisfying the following properties:*

$$\begin{array}{lcl}
\Pi, H, \sigma \models_{\mathsf{I}} E & \iff & \llbracket E \rrbracket_{H,\sigma} = \mathit{true} \\
\Pi, H, \sigma \models_{\mathsf{I}} \mathbf{acc}(E.f, q) & \iff & \Pi(\llbracket E \rrbracket_{H,\sigma}, f) \geq q \\
\Pi, H, \sigma \models_{\mathsf{I}} E \to A & \iff & \Pi, H, \sigma \models_{\mathsf{I}} E \ \Rightarrow\ \Pi, H, \sigma \models_{\mathsf{I}} A \\
\Pi, H, \sigma \models_{\mathsf{I}} A_1 * A_2 & \iff & \exists \Pi_1, \Pi_2 : \Pi = \Pi_1 + \Pi_2 \ \wedge\ \Pi_1, H, \sigma \models_{\mathsf{I}} A_1 \\
 & & \quad \wedge \Pi_2, H, \sigma \models_{\mathsf{I}} A_2 \\
\Pi, H, \sigma \models_{\mathsf{I}} \mathbf{acc}(E.P) & \iff & \Pi(\llbracket E \rrbracket_{H,\sigma}, P) \geq 1 \\
\Pi, H, \sigma \models_{\mathsf{I}} \mathit{Thread}(x, m, y, z) & \iff & \Pi[\sigma(x), \mathtt{recv}] = 1 \wedge H[\sigma(x), \mathtt{recv}] = \sigma(y) \\
 & & \quad \wedge\ \Pi[\sigma(x), \mathtt{param}] = 1 \wedge H[\sigma(x), \mathtt{param}] = \sigma(z) \\
 & & \quad \wedge\ \Pi[\sigma(x), \mathtt{meth}] = 1 \wedge H[\sigma(x).\mathtt{meth}] = m
\end{array}$$

Iso-entailment $A \models_{\mathsf{I}} A'$ *holds if:* $\forall \Pi, H, \sigma.\ \Pi, H, \sigma \models_{\mathsf{I}} A \implies \Pi, H, \sigma \models_{\mathsf{I}} A'$.

Crucially, the semantics of predicate permissions $\mathbf{acc}(E.P, n)$ do not involve recursion; it is sufficient to simply check that permission to the *predicate* instance is in the direct permissions. This does not directly enforce that the body of the predicate holds in the current state, but, as we shall show in the next section, we push this concern to the definition of a "good" isorecursive state; since a verifier cannot in general enforce that a recursive definition holds, this has to be

pushed to the soundness of the underlying methodology (i.e., the equirecursive semantics).

We can now define the notion of framing for iso-expressions and iso-assertions (cf. Definition 5 for the equi-world).

Definition 10 (Framed and Self-Framing Iso-Assertions). *An iso-assertion expression E, or assertion A is* iso-framed *in a state consisting of H, Π and σ, as defined by judgements $\models^{\Pi,H,\sigma}_{\mathsf{frml}} E$ and $\models^{\Pi,H,\sigma}_{\mathsf{frml}} A$. Full definitions are provided in the appendix, but all cases of these judgements are analogous to those of Definition 5, with the following exceptions:*

$$\models^{\Pi,H,\sigma}_{\mathsf{frml}} E.g(E') \iff \models^{\Pi,H,\sigma}_{\mathsf{frml}} E \wedge \models^{\Pi,H,\sigma}_{\mathsf{frml}} E' \wedge \Pi, H, \sigma' \models_{\mathsf{I}} \mathit{Pre}(g) \text{ where } \sigma' = [\mathsf{this}, \mathsf{X} \mapsto \lceil\!\lceil E \rceil\!\rceil_{H,\sigma}, \lceil\!\lceil E' \rceil\!\rceil_{H,\sigma}]$$

$$\models^{\Pi,H,\sigma}_{\mathsf{frml}} \mathbf{unfolding}\, E.P\, \mathbf{in}\, E' \iff \models^{\Pi,H,\sigma}_{\mathsf{frml}} E \wedge \Pi(\lceil\!\lceil E \rceil\!\rceil_{H,\sigma}, P) \geq 1 \wedge \models^{\Pi',H,\sigma}_{\mathsf{frml}} E' \text{ where } \Pi' = \Pi \odot_H \lceil\!\lceil E \rceil\!\rceil_{H,\sigma}.P$$

$$\models^{\Pi,H,\sigma}_{\mathsf{frml}} \mathbf{acc}(E.P) \iff \models^{\Pi,H,\sigma}_{\mathsf{frml}} E$$

An iso-expression E is framed by an assertion A, *written $A \models_{\mathsf{frml}} E$, if, (for all Π, H, σ) we have $\Pi, H, \sigma \models_{\mathsf{I}} A$ implies that $\models^{\Pi,H,\sigma}_{\mathsf{frml}} E$.*
An iso-assertion A is self-framing*, written $\models_{\mathsf{frml}} A$ if, for all H, Π and σ:*
$\Pi, H, \sigma \models_{\mathsf{I}} A \;\Rightarrow\; \models^{\Pi,H,\sigma}_{\mathsf{frml}} A$

For example, $\mathit{Thread}(x, m, y, z)$ and $\mathsf{this}.\mathit{List}$ are self-framing assertions, while $\mathsf{this}.\mathit{next}{=}\mathit{null}$ is not.

The rule at the heart of the iso-expressions is the one describing framing for $\mathbf{unfolding}\, E.P\, \mathbf{in}\, E'$: it requires that the context holds permission to the predicate $E.P$, and *adds* the permissions from the body of $E.P$ into the currently held permissions, in order to check framedness of E'. Therefore, in a context where $\Pi(\sigma(\mathsf{this}), \mathit{next}) = 0$, and $\Pi(\sigma(\mathsf{this}), \mathit{List}) = 1$, the expression $\mathsf{this}.\mathit{next}$ is not framed, while the expression $\mathbf{unfolding}\, \mathsf{this}.\mathit{List}\, \mathbf{in}\, \mathsf{this}.\mathit{next}$ is.

Most importantly, the notion of framing no longer requires a recursive traversal of predicate definitions, as opposed to that from definition 5, which required potentially infinite unrolling. This can only be justified with two further ingredients; firstly, that predicate definitions are always self-framing and functions are only applied in contexts where their bodies are guaranteed to terminate, and secondly that holding a predicate always implicitly guarantees that its body holds in the same state. The former of these two ingredients is provided by the following definition, while the second is provided by the notion of "good state" in the next section.

Definition 11 (Well-formed Definitions).

The definition of an iso-predicate P is well-formed*, if $\models_{\mathsf{frml}} \mathit{Body}(P)$.*

The definition of an iso-function g is well-formed*, if:* (1) $\models_{\mathsf{frml}} \mathit{Pre}(g)$*, and* (2) $\forall \Pi, H, \sigma.(\Pi, H, \sigma \models_{\mathsf{I}} \mathit{Pre}(g) \implies (\models^{\Pi,H,\sigma}_{\mathsf{frml}} \mathit{Body}(g) \wedge \lceil\!\lceil \mathit{Body}(g) \rceil\!\rceil_{H,\sigma} \neq \mathit{error}_{tp}))$*, where tp is the return type of g.*

A program is well-formed *if all function and predicate definitions are well-formed.*

For example, a function *bad*, defined as $bad(x) = bad(x) + 1$, would be well-formed it is precondition was *false*, and not otherwise. Moreover, the function *length* with precondition *List* is well-formed, and would not be well-formed with precondition, say, *true*.

Note that this notion of well-formedness can be checked *without* unrolling definitions recursively. In the next section, we will show how the isorecursive definitions above correctly approximate the corresponding equirecursive notions.

Discussion We considered making the error value $error_{tp}$, an (unknown) element of the type *tp*. This approach of *underspecification* is taken in some classical logic based handlings of partial functions. This would work correctly for the semantics of assertions, but would not work correctly for Definition 11, since we check that functions do not evaluate to this value. For example, equating $error_{bool}$ with *true* or with *false*, would turn a boolean function $g(x) = (x > 3)$, into a badly-formed function.

Returning to the three points outlined at the end of the previous section, we can see that our isorecursive definitions directly handle the first and third points of the list (the semantics of predicates, and the checking of framing of expressions and assertions), without requiring a recursive unrolling of any definition. These can therefore be implemented effectively in a static tool. With respect to the second point of our list, our isorecursive expression semantics still defines function calls directly in terms of their bodies. Many tools handle this problem by applying a variety of additional heuristics, ghost code markers or triggers [12], in order to implement a constrained version of the definition we employ here. Since we did not want to prescribe exactly how this definition should be constrained, we left this point open in the above. In particular, it is possible for a reasonably complete handling of this definition to be achieved based only on the folding and unfolding of *predicates* [7].

4 Comparing the Assertion Semantics

We now turn to relating our two semantics. Our eventual goal is to define an *erasure*, mapping our isorecursive constructions to their equirecursive counterparts, in order to show that verification based on isorecursive semantics gives a sound approximation of verification based on equirecursive semantics. In particular, we will show that fold, unfold and unfolding (which are essential for defining semantics in the isorecursive sense) can all be eliminated from the language, and the resulting program still satisfies the erased version of its specifications. Since permissions (let alone permissions to predicates) are not reflected at runtime, this leads us closer to a runtime model suitable for proving soundness with respect to an operational semantics. In this section, we focus on the relationship between the two semantics for assertions.

Definition 12 (Encoding). *The encoding* $\langle\!\langle \_ \rangle\!\rangle$ *maps isorecursive expressions and assertions to their equirecursive counterparts, typically by injection, e.g.*

$\langle\!\langle\, x \,\rangle\!\rangle = x, \quad \langle\!\langle\, E.f \,\rangle\!\rangle = \langle\!\langle\, E \,\rangle\!\rangle.f, \quad$ *and* $\quad \langle\!\langle\, \mathbf{acc}(E.f, q) \,\rangle\!\rangle = \mathbf{acc}(\langle\!\langle\, E \,\rangle\!\rangle.f, q).$

For the cases specific to iso-expressions and assertions, we have
$\langle\!\langle\, \mathbf{unfolding}\, E.P\, \mathbf{in}\, E' \,\rangle\!\rangle = \langle\!\langle\, E' \,\rangle\!\rangle$, *and* $\langle\!\langle\, \mathbf{acc}(E.P) \,\rangle\!\rangle = \langle\!\langle\, E \,\rangle\!\rangle.P$.
The full definition appears in the appendix.

Unfolding expressions are unnecessary in the equirecursive world, where predicates and their bodies are not differentiated between. However, the unrolling of predicate definitions may still lead to the discovery that too many field permissions were implicitly held in an isorecursive state; not all isorecursive masks have an equirecursive analogue. This is described in the next definition.

Definition 13 (Encoding permissions). *The (partial) translation* $\langle\!\langle\, \_ \,\rangle\!\rangle$ *encodes isorecursive permission maps* Π *into equirecursive permission maps* π*,*

$$\langle\!\langle\, \Pi \,\rangle\!\rangle_H = \{(\iota, f) \mapsto q \mid \Pi[\iota, f] = q\} + \{(t, f) \mapsto q \mid \Pi[t, f] = q\} + \langle\!\langle\, \Pi' \,\rangle\!\rangle_H$$
$$\text{where } \Pi' = \textstyle\sum_{\Pi(\iota.P)>0}(\mathcal{P}_{\mathsf{I}}(Body(P), H, [\mathsf{this} \mapsto \iota]))$$

Note that $\langle\!\langle\, \Pi \,\rangle\!\rangle_H$ may not be well-defined (if a predicate instance held in Π has an infinite unfolding; then the definition above will not terminate), and even when defined, it may not yield a well-formed equi-permissions-mask (cf. Definition 2), if too many field permissions are accumulated. Furthermore, it could be that the constraints required in the bodies of predicates are not always correctly reflected in an iso-recursive state. Thus, we define the notion of a "good" isorecursive state.

Definition 14 (Good Iso-States). *An isorecursive state defined by heap* H*, iso-permissions-mask* Π *and environment* σ *is "good", if:*

1. $\langle\!\langle\, \Pi \,\rangle\!\rangle_H$ *is defined, and satisfies* $\models (\langle\!\langle\, \Pi \,\rangle\!\rangle_H)$.
2. *For all* ι, P *such that* $\Pi[\iota, P] > 0$, $(\langle\!\langle\, \Pi \,\rangle\!\rangle_H), H, \sigma' \models_{\mathsf{E}} w.P$ *is satisfied, where* w *is a fresh variable, and* σ' *is the environment* σ *extended with the mapping* $[w \mapsto \iota]$.

Note, in particular, that the second of these two requirements enforces that the original state does not hold permission to any predicate instance whose definition can be unrolled infinitely. As motivated in Section 2, such predicates are interpreted as false in the equirecursive semantics in any case, so ruling out such states in advance is consistent with this view. In general, good iso-states are those which can have a meaningful corresponding equirecursive counterpart; all others are artifacts of the incomplete knowledge provided by isorecursive definitions.

In Lemma 1 we show that we can map a judgement back from the equirecursive world to the isorecursive, starting from a "good" state in the isorecursive world. Then, in Theorem 1, we show our erasure results.

Lemma 1. *In a well-formed program the following properties hold*:

1. *If* $\Pi(\iota.P) > 0$*, then* $\langle\!\langle\, \Pi \,\rangle\!\rangle_H = \langle\!\langle\, \Pi \odot_H \iota.P \,\rangle\!\rangle_H$.
2. *If* $\pi, H, \sigma \models_{\mathsf{E}} \langle\!\langle\, A \,\rangle\!\rangle$*, then* $\exists \Pi$ *s.t.* (Π, H, σ) *is a good iso-state,* $\langle\!\langle\, \Pi \,\rangle\!\rangle_H = \pi$*, and* $\Pi, H, \sigma \models_{\mathsf{I}} A$.

Theorem 1 (Erasure Results).

1. $\lVert\!\langle\!\langle E \rangle\!\rangle\!\rVert_{H,\sigma} = \lVert E \rVert_{H,\sigma}$.
2. *If* $\Pi, H, \sigma \models_{\mathsf{I}} A$ *and* (Π, H, σ) *is a good iso-state, then* $\langle\!\langle \Pi \rangle\!\rangle_H, H, \sigma \models_{\mathsf{E}} \langle\!\langle A \rangle\!\rangle$.
3. *If* (Π, H, σ) *is a good iso-state and* $\models_{\mathsf{frmI}}^{\Pi,H,\sigma} A$, *then* $\models_{\mathsf{frmE}}^{\langle\!\langle \Pi \rangle\!\rangle,H,\sigma} \langle\!\langle A \rangle\!\rangle$.
4. *If all functions and predicates are well-formed, then if an iso-assertion* A *is self-framing, then* $\langle\!\langle A \rangle\!\rangle$ *is self-framing.*
5. *If* $A \models_{\mathsf{I}} A'$, *then* $\langle\!\langle A \rangle\!\rangle \models_{\mathsf{E}} \langle\!\langle A' \rangle\!\rangle$.

The proof sketches are given in the appendix. These results demonstrate that the more-readily-checkable isorecursive notions supplied in the previous section, accurately approximate the intended underlying equirecursive notions.

In the following sections, we extend this argument to Hoare Logics for a small language. We will then be in a position to prove that verifying a program with respect to *isorecursive* definitions, is sufficient to guarantee soundness, via our erasure results, and a soundness proof with respect to equirecursive semantics.

5 Hoare Logic for Iso-Assertions

In this section, we define an axiomatic semantics for a small (but representative) subset of Chalice, with respect to our iso-recursive assertion semantics (as defined in Section 3). This is the semantics closest to most implementations, but it is not so intuitive or useful as a runtime model. In the following section, we will define a corresponding Hoare Logic for our equirecursive semantics, and demonstrate the relationship between the two.

5.1 Chalice Syntax

We begin by defining our Chalice subset:

Definition 15 (Isorecursive Chalice Syntax). *We assume a set of predefined methods, ranged over by* m. *For simplicity, methods are assumed to have exactly one parameter, and to always return a value. Furthermore, method names are assumed to be unique in the whole program.*
Simple statements*, ranged over by* R*, and* statements*, ranged over by* S*, are mutually defined by the following grammars:*

$$\begin{array}{rcl} R & ::= & \texttt{skip} \mid x{:=}E \mid x.f{:=}y \mid \texttt{return}\ x \mid x{:=}\ \texttt{new}\ c \mid (\texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2) \\ & \mid & x{:=}\ \texttt{fork}\ y.m(z) \mid y{:=}\ \texttt{join}\ x \mid \textbf{fold}\, x.p \mid \textbf{unfold}\, x.p \\ S & ::= & R \mid (R; S) \end{array}$$

A statement S *is* return-ended *if the right-most simple statement occurring in its structure is of the form* `return` x*; i.e., it can be constructed by the following sub-grammar:* $S ::= \texttt{return}\ x \mid (R; S)$
Composition of statements s_1 *and* s_2*, for which we use the (overloaded) notation* $(S_1; S_2)$*, results in a statement which represents appending the two sequences of simple statements; i.e., when* S_1 *is not a simple statement (say,* $S_1 = (R; S')$*), is defined by recursively rewriting* $((R; S'); S_2) = (R; (S'; S_2))$.

Our syntax only allows for sequential compositions to be nested to the right, which simplifies the definition of the operational semantics (see Section 7), since we do not need a separate concept of evaluation contexts for such a simple language. Our language only allows general expressions e within variable assignments, and otherwise employs only program variables for expressions, but the generalisation is easily encoded (or made). Multi-threading is achieved by the ability to *fork* and *join* threads. The statement $w{:=}\ \texttt{fork}\ m.y(z)$ has the meaning of starting an invocation of a call to method m (with receiver y and parameter z) in a new thread. The returned value (stored in w) is a *token*, which gives a means of referring to this newly-spawned thread. Such a token can be used to *join* the thread later (which has the operational meaning of waiting for the thread to terminate, and then receiving its return value); this is provided by the $x{:=}\ \texttt{join}\ w$ statement. In order to model information about active threads/tokens, we need to extend our assertion logic with an additional assertion $\mathit{Thread}(x, m, y, z)$, which has the meaning that x stores a token corresponding to a forked thread executing method m with receiver y and parameter z. In order to represent this in our assertion semantics, we extend our heaps with extra thread objects, which store the information m,y,z in ghost fields. These fields cannot be referred to directly in assertions, but we extend our semantic judgement with the following rule:

$\pi, H, \sigma \models_{\mathsf{E}} \mathit{Thread}(x, m, y, z) \Leftrightarrow$
$\quad \pi[\sigma(x), \mathit{recv}] = \pi[\sigma(x), \mathit{param}] = \pi[\sigma(x), \mathit{meth}] = 1$
$\quad \wedge\ H[\sigma(x).\mathit{recv}] = \sigma(y) \wedge H[\sigma(x).\mathit{param}] = \sigma(z) \wedge H[\sigma(x).\mathit{meth}] = m$

and identically for its isorecursive semantics. Thus, $\mathit{Thread}(x, m, y, z)$ is a self-framing assertion, which can be passed between threads to indicate which thread has permission to join a token.

We do not include loops, since they present no relevant challenges compared with recursion. While we do not support a method call statement, we do allow the fork and join of method invocations. A normal method call of the form $x{:=}\ m.y(z)$ can be encoded by a sequence $(w{:=}\ \texttt{fork}\ m.y(z)\ ; x{:=}\ \texttt{join}\ w\ )$ (for some fresh variable w).

5.2 Hoare Logic

We now define the Hoare Logic corresponding to isorecursive assertion semantics.

Definition 16 (Isorecursive Hoare Logic). Isorecursive Hoare Triples *are written* $\vdash_I \{A\}\ S\ \{A'\}$*, where* A *and* A' *are self-framing isorecursive assertions, and* S *is an Isorecursive Chalice statement. The rules are shown in Figure 1. We leave implicit the requirement that* A *and* A' *are always self-framing; in particular, whenever we write a triple (even as a new conclusion of a derivation rule), this requirement must be satisfied in addition to the explicit premises.*

Our Hoare triples employ isorecursive assertions as pre- and post-conditions, with the restriction that the assertions used must always be *self-framing*. The restriction to self-framing assertions is important for soundness. For example, without this requirement, it would naturally be possible to derive triples such as

$\vdash_I \{x.f = 1\}$ (`skip`; `skip`) $\{x.f = 1\}$, which, when evaluated at runtime, might not be sound (another thread could modify the location $x.f$ during execution). Indeed, in our soundness proof, the requirement that every thread has a self-framing pre-condition is essential to the argument.

$$\frac{}{\vdash_I \{A\}\ \texttt{skip}\ \{A\}}\ (\mathit{skipI}) \qquad \frac{A[E/x] \models_{\mathsf{frml}} E}{\vdash_I \{A[E/x]\}\ x{:=}E\ \{A\}}\ (\mathit{varassI})$$

$$\frac{}{\vdash_I \{x \neq \mathit{null} * \mathbf{acc}(x.f, 1)\}\ x.f{:=}y\ \{\mathbf{acc}(x.f, 1) * x.f = y\}}\ (\mathit{fldassI})$$

$$\frac{}{\vdash_I \{A\}\ \texttt{return}\ x\ \{A * \mathsf{result} = x\}}\ (\mathit{retI})$$

$$\frac{\overline{f_i} = \mathit{fields}(c)}{\vdash_I \{\mathit{true}\}\ x{:=}\ \texttt{new}\ c\ \{(\overline{*\mathbf{acc}(x.f_i, 1)})\}}\ (\mathit{newI})$$

$$\frac{\vdash_I \{A * B\}\ S_1\ \{A'\} \quad \vdash_I \{A * \neg B\}\ S_2\ \{A'\}}{\vdash_I \{A\}\ (\texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2)\ \{A'\}}\ (\mathit{ifI})$$

$$\frac{A = \mathit{Pre}(m)[y/\mathsf{this}][z/\mathsf{X}]}{\vdash_I \{y \neq \mathit{null} * A\}\ x{:=}\ \texttt{fork}\ y.m(z)\ \ \{\mathit{Thread}(x, m, y, z)\}}\ (\mathit{forkI})$$

$$\frac{A' = \mathit{Post}(m)[y/\mathsf{this}][z/\mathsf{X}][w/\mathsf{result}]}{\vdash_I \{\mathit{Thread}(x, m, y, z)\}\ w{:=}\ \texttt{join}\ x\ \ \{A'\}}\ (\mathit{joinI})$$

$$\frac{\vdash_I \{A\}\ R\ \{A'\} \quad \vdash_I \{A'\}\ S\ \{A''\}}{\vdash_I \{A\}\ (R; S)\ \{A''\}}\ (\mathit{seqI})$$

$$\frac{A_1 \models_{\mathsf{I}} A_3 \quad \vdash_I \{A_3\}\ s\ \{A_4\} \quad A_4 \models_{\mathsf{I}} A_2}{\vdash_I \{A_1\}\ s\ \{A_2\}}\ (\mathit{consI})$$

$$\frac{\vdash_I \{A\}\ S\ \{A'\} \quad \models_{\mathsf{frml}} A'' \quad \mathit{mods}(S) \cap \mathit{FV}(A'') = \emptyset}{\vdash_I \{A'' * A\}\ S\ \{A' * A''\}}\ (\mathit{frameI})$$

$$\frac{A' = \mathit{Body}(P)[x/\mathsf{this}]}{\vdash_I \{A * A' * B\}\ \mathbf{fold}\,\mathbf{acc}(x.P, q)\ \{A * \mathbf{acc}(x.P, q) * \mathbf{unfolding}\,x.P\,\mathbf{in}\,B\}}\ (\mathit{foldI})$$

$$\frac{A' = \mathit{Body}(P)[x/\mathsf{this}]}{\vdash_I \{A * \mathbf{acc}(x.P, q) * \mathbf{unfolding}\,x.P\,\mathbf{in}\,B\}\ \mathbf{unfold}\,\mathbf{acc}(x.P, q)\ \{A * A' * B\}}\ (\mathit{unfoldI})$$

Fig. 1. Hoare Logic for Isorecursive semantics

Some of the rules (such as the treatment of conditionals, and the rule of consequence, (*consI)*)) are standard, but others warrant discussion. The *frame rule* (*frameI)*) (whose role is to preserve parts of the state which are not relevant for

the execution of the particular statement) is similar to that typically employed in separation logics [8]. The extra assertion A'' can be "framed on" under two conditions; firstly, that no variables mentioned in A'' are modified by the statement s, and secondly, that A'' is self-framing. The two conditions are necessary for similar reasons; if the execution of s could change the meaning of A'', then to simplify conjoin it unchanged to both pre- and post-condition would be incorrect. The two ways in which the state can change in our language are through variable assignments (whose effects are tamed by the first requirement) and field assignments (which cannot affect the meaning of A'', since A'' is self-framing, and therefore comes along with sufficient permission to rule out assignment to the fields on which its meaning depends).

The (*varassI*) rule is similar to a standard Hoare Logic rule for assignment, but with the extra requirement that the expression to be assigned is readable in the pre-condition state. The premise guarantees that fields are only read when appropriate permissions are known to be available, and functions are only applied when their pre-conditions are known in the state. The rule (*fldassI*) is slightly subtle: it must avoid the possibility of outdated heap-dependent expressions surviving the assignment; the requirement for *full* permission to the written field location from the pre-condition forces any information previously-known about that location to be discarded (i.e., using rule (*consI*)) prior to applying this rule.

The rules for forking and joining threads employ the special $\mathit{Thread}(x, m, y, z)$ assertion, discussed above. Our formulation does not allow this knowledge to be split amongst threads (though it can be passed from thread to thread in contracts).

The two most important rules for the isorecursive semantics are those for folding and unfolding predicate instances. For example, consider folding a predicate instance, as defined by rule (*foldI*)). It is easy to see that this rule should exchange the body of the predicate instance (the assertion A' for a permission to the predicate itself). The challenge is to enable the preservation of information which was previously framed by the predicate's contents, even though those permissions have (after folding) been stored into the predicate body. For example, consider a predicate P whose definition is $\mathbf{acc}(\mathit{this}.g, 1)$. In a state in which we know $\mathbf{acc}(\mathit{this}.g) * \mathit{this}.g = 4$, we could not treat a fold of P as a simple exchange of $\mathbf{acc}(\mathit{this}.g)$ for $\mathbf{acc}(\mathit{this}.P)$, since, in the resulting state, $\mathit{this}.g = 4$ would not be framed. Instead, our rule allows us to derive the post-condition $\mathbf{acc}(\mathit{this}.P) * \mathbf{unfolding}\, \mathit{this}.P\, \mathbf{in}\, \mathit{this}.g = 4$, which is self-framing. Furthermore, in order to handle the possibility that we wish to preserve an expression which is framed *partially* by the permissions required by a predicate body, we allow the presence of a further assertion A in the rule. This allows, e.g., a pre-condition such as $\mathbf{acc}(\mathit{this}.f) * \mathbf{acc}(\mathit{this}.g) * \mathit{this}.f = \mathit{this}.g$ for a statement $\mathbf{fold}\, \mathbf{acc}(\mathit{this}.P)$ to be used to derive a post-condition $\mathbf{acc}(\mathit{this}.f) * \mathbf{acc}(\mathit{this}.P) * \mathbf{unfolding}\, \mathit{this}.P\, \mathbf{in}\, \mathit{this}.f = \mathit{this}.g$, in which the additional assertion ($\mathbf{acc}(\mathit{this}.f)$ in this case) provides additional permissions required for self-framing. The condition that A must be self-framing is necessary

to avoid that A itself might represent information which was only framed by permissions from the body of P.

The rule for unfolding predicates is exactly symmetrical with that for folding predicates. In particular, it enables information from unfolding expressions depending on the predicate to be unfolded, to be preserved (without an unfolding expression) in the post-state.

We can characterise the derivable triples in our Hoare Logic, using a *generation lemma*; an example case is shown here.

Lemma 2 (Generation Lemma).

- $\vdash_I \{A\}\ x{:=}E\ \{A'\}\ \Leftrightarrow$
  $\exists A''.(A''\ s.f.\ \wedge\ A \models_{\mathsf{I}} A''[E/x]\ \wedge\ A''[E/x] \models_{\mathsf{frml}} E\ \wedge\ A'' \models_{\mathsf{I}} A')$
- *Remaining cases in the appendix.*

6 Hoare Logic for Equi-Assertions

In this section, we employ a second Hoare Logic based on our equirecursive assertion semantics. Firstly, we define an "erased" form of our statement syntax, in which only equi-recursive expressions are used, and no fold and unfold statements may occur.

Definition 17 (Equirecursive Chalice Syntax). Simple runtime statements*, ranged over by* r*, and* runtime statements*, ranged over by* s*, are mutually defined by the following grammars:*

$$\begin{array}{rcl} r & ::= & \texttt{skip} \mid x{:=}e \mid x.f{:=}y \mid \texttt{return}\ x \mid x{:=}\ \texttt{new}\ c \mid (\texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2) \\ & \mid & x{:=}\ \texttt{fork}\ y.m(z) \mid y{:=}\ \texttt{join}\ x \\ s & ::= & r \mid (r; s) \end{array}$$

The notions of return-ended statements *and* composition of statements *are analogous to those of Definition 15.*

We can now define the equirecursive analogue of Definition 16.

Definition 18 (Equirecursive Hoare Logic). Equirecursive Hoare Triples *are written* $\vdash_E \{a\}\ s\ \{a'\}$*, where* a *and* a' *are self-framing equirecursive assertions, and* s *is an equirecursive Chalice statement. For space reasons, the full rules are not given here. However, all rules are analogous to those of our Isorecursive Hoare Logic (Definition 16), except that the corresponding equirecursive notions of entailment, self-framing, statements, assertions etc. are used throughout. In addition, there are no rules for fold and unfold statements (since these do not occur in equirecursive Chalice). The full rules are given in the appendix (Figure 4).*

We now extend our previous erasure results (mapping isorecursive to equirecursive assertions) to also define an erasure on statements. This operation applies erasure to all assertions and expressions, and replaces all *fold*/*unfold* statements with *skip*.

Definition 19 (Encoding iso-statements to equi-statements). *We overload the encoding* $\langle\!\langle\ \_\ \rangle\!\rangle$ *to map isorecursive to equirecursive statements, as*:

$$\langle\!\langle\, x{:=}E \,\rangle\!\rangle = x{:=}\langle\!\langle\, E \,\rangle\!\rangle \quad \langle\!\langle\, (S_1; S_2) \,\rangle\!\rangle = (\langle\!\langle\, S_1 \,\rangle\!\rangle; \langle\!\langle\, S_2 \,\rangle\!\rangle)$$
$$\langle\!\langle\, (\texttt{if}\ E\ \texttt{then}\ S_1\ \texttt{else}\ S_2) \,\rangle\!\rangle = (\texttt{if}\ \langle\!\langle\, E \,\rangle\!\rangle\ \texttt{then}\ \langle\!\langle\, S_1 \,\rangle\!\rangle\ \texttt{else}\ \langle\!\langle\, S_2 \,\rangle\!\rangle)$$
$$\langle\!\langle\, \mathbf{fold}\, x.P \,\rangle\!\rangle = \texttt{skip} = \langle\!\langle\, \mathbf{unfold}\, x.P \,\rangle\!\rangle$$
$$\langle\!\langle\, S \,\rangle\!\rangle = S \quad \textit{otherwise}$$

Theorem 2. *If* A, A' *are self-framing iso-assertions, and* S *is an isorecursive Chalice statement, then*

$$\vdash_I \{A\}\ S\ \{A'\}\ \Rightarrow \vdash_E \{\langle\!\langle\, A \,\rangle\!\rangle\}\ \langle\!\langle\, S \,\rangle\!\rangle\ \{\langle\!\langle\, A' \,\rangle\!\rangle\}$$

7 Operational Semantics and Soundness

In this section, we show soundness of our formalisations, with respect to an interleaving small-step operational semantics. We formalise our runtime model with respect to a collection of *threads* and *objects*, together referred to as *runtime entities*. We do not model explicit object allocation; instead, we assume that all objects are pre-existing (and already have classes), but have a flag which indicates whether they are truly allocated or not. When unallocated, an object holds the permission to all of its own fields. Thus, we never need to create or destroy permission in the system; it is merely transferred from entity to entity. Similarly, we do not model creation of new threads, but just assume that idle thread entities exist in the system, which can be assigned a task (i.e., a method invocation) to begin executing.

Definition 20 (Runtime Ingredients). *Recall that object identifiers are ranged over by* ι*, and the (disjoint) set of thread identifiers is ranged over by* t*. We assume a fixed mapping* $cls(\iota)$ *from object identifiers to class names.*
A runtime heap h *is a mapping from pairs of object identifier and field name, to values.*
A thread configuration T *is defined by* $T ::= \{\sigma, s\} \mid \texttt{idle}$
A thread entity *is a thread configuration labelled with a thread identifier,* T_t*. A thread entity is called* active *if it is of the form* $\{\sigma, s\}_t$*.*
An object state O *is defined by* $O ::= \texttt{alloc} \mid \texttt{free}$*, and an* object entity *is an object state labelled with an object identifier, written* O_ι*.*
A labelled entity N_n *is defined by* $N_n ::= T_t \mid O_\iota$*, where the label* n *denotes the thread or object identifier of the entity, respectively.*

Note that, in contrast to the heaps of Definition 2, runtime heaps do not store ghost information about thread identifiers. At runtime, this information is directly available via the thread configurations present.

We define two main types of small-step transitions, which we call *local* and *paired* transitions. A local transition is one which affects only a single (thread) entity and the heap.

Definition 21 (Local transitions). Local transitions *map a heap and thread entity to a new heap and thread entity (with the same thread identifier), and are written* $h, T_t \xrightarrow[l]{} h', T'_t$. *These rules have the expected shape, e.g.*

$$\frac{[\![e]\!]_{h,\sigma} = v}{h, \{\sigma, (x{:=}e; s)\}_t \xrightarrow[l]{} h, \{\sigma[x{\mapsto}v], s\}_t} \; (varassS)$$

The full rules for local transitions and are defined in Figure 5 in the appendix.

$$\frac{\sigma(y) = \iota \quad \sigma' = [\mathsf{this} \mapsto \iota, \mathsf{X} \mapsto \sigma(z), \mathsf{method} \mapsto m] \quad s' = \langle\!\langle\, Body(m) \,\rangle\!\rangle}{\begin{array}{c} h, (\{\sigma, (x{:=}\ \mathtt{fork}\ y.m(z)\ ; s)\}_{t_1} \| \mathtt{idle}_{t_2}) \\ \xrightarrow[p]{} h, (\{\sigma[x{\mapsto}t_2], s\}_{t_1} \| \{\sigma', s'\}_{t_2}) \end{array}} \; (forkS)$$

$$\frac{\sigma_1(y) = t_2}{\begin{array}{c} h, (\{\sigma_1, (x{:=}\ \mathtt{join}\ y\ ; s)\}_{t_1} \| \{\sigma_2, \mathtt{return}\ z\}_{t_2}) \\ \xrightarrow[p]{} h, (\{\sigma_1[x{\mapsto}\sigma_2(z)], s\}_{t_1} \| \mathtt{idle}_{t_2}) \end{array}} \; (joinS)$$

$$\frac{cls(\iota) = c \quad \overline{f_i} = fields(c) \quad h' = h[\overline{(\iota, f_i){\mapsto}null}]}{h, (\{\sigma, (x{:=}\ \mathtt{new}\ c; s)\}_t \| \mathtt{free}(c)_\iota) \xrightarrow[p]{} h', (\{\sigma[x{\mapsto}\iota], s\}_t \| \mathtt{alloc}(c)_\iota)} \; (newS)$$

Fig. 2. Paired transitions

The more complex transitions are concerned with forking and joining threads, and with object allocation. In the case of forking and joining, we define transitions which simultaneously involve *two* thread entities; one which is executing the *fork/join* statement, and one which represents the thread being forked/joined. In the case of a fork, the second thread entity must be initially idle, while in the case of a join, it must have finished executing and be ready to return. Object allocation, on the other hand, is a transition involving a thread entity and an object entity together; it takes an object entity in the `free` state (and of the appropriate class), and switches it to `alloc`.

Definition 22 (Paired transitions). Paired transitions *map a heap and a pair of entities to a new heap and pair of entities (with the same identifiers), and are written* $h, (T_t \| N_n) \xrightarrow[p]{} h', (T'_t \| N'_n)$. *They are defined in Figure 2.*

We can now define the operational semantics for a whole system.

Definition 23 (Runtime Configurations and Operational Semantics). *A* runtime entity collection *C is a pair $(\overline{T_t}, \overline{O_\iota})$ consisting of a set of thread entities (one for each thread identifier t) and a set of object entities (one for each object identifier ι). A* runtime configuration *is a pair h, C of a runtime heap and a runtime entity collection.*

The interleaving operational semantics of such a configuration is given by a transition relation of the form $h, C \xrightarrow[c]{} h', C'$, and is defined by the reflexive, transitive closure of the rules in Figure 3.

$$\frac{C[t] = (T_t) \quad h, T_t \xrightarrow[l]{} h', T_t'}{h, C \xrightarrow[c]{} h', C[t \mapsto T_t']} \, (selectLocalS)$$

$$\frac{C[t] = (T_t) \quad C[n] = (N_n) \quad h, (T_t \| N_n) \xrightarrow[p]{} h', (T_t' \| N_n')}{h, C \xrightarrow[c]{} h', C[t \mapsto T_t'][n \mapsto N_n']} \, (selectPairS)$$

Fig. 3. Configuration transitions

In order to reuse our equirecursive assertion logic semantics for runtime configurations, we define a mapping *Heap* back from runtime configurations to heaps (cf. Definition 2). An application of this map $Heap(h, C)$ simply reconstructs the ghost information about threads, from the information in the runtime entity collection, adding it to the heap information in h. We also require an equirecursive permission collection function $\mathcal{P}_{\mathtt{E}}$, which collects all of the permissions explicitly or implicitly required in equi-assertions a. Both operators are defined in the appendix.

We can now turn to the central notion of our soundness proof; what it means for a runtime configuration to be *valid*. Essentially, this prescribes that the permissions to the fields of all allocated objects can be notionally divided amongst the active threads (point 3 below), and suitable preconditions for the statements of each thread can be chosen that are satisfied in the current runtime configuration, and for which each statement can be verified (via our equirecursive Hoare Logic) with respect to the thread's current postcondition.

Definition 24 (Valid configuration). *A runtime configuration (h, C), is* valid *if there exists a set of equirecursive assertions $\overline{a_t}$, (one for each thread identifier t), such that:*

1. *For each thread entity of the form* $\mathtt{idle}_t$, $a_t = true$.
2. *For each thread entity of the form $\{\sigma_t, s_t\}_t$ in C, letting $H = Heap(h, C)$, we have both $\mathcal{P}_{\mathtt{E}}(a_t, H, \sigma_t), H, \sigma_t \models_{\mathtt{E}} a_t$ and $\vdash_E \{a_t\}\ s_t\ \{Post(\sigma_t(\mathsf{method}))\}$.*
3. $\models (\sum_{t \in C} \mathcal{P}_{\mathtt{E}}(a_t, h, \sigma_t)) + (\sum_{\mathtt{free}(c)_\iota \in C} \sum_{f \in fields(c)} \{(\iota, f) \mapsto 1\}) +$
$(\sum_{\mathtt{idle}_t \in C} \{(t, \mathtt{recv}) \mapsto 1, (t, \mathtt{param}) \mapsto 1, (t, \mathtt{meth}) \mapsto 1\})$

Finally, we can turn to our main soundness result, which shows that modular verification of each definition in the program, using our isorecursive semantics, is sound with respect to the interleaving operational semantics of the language.

Theorem 3 (Soundness of Isorecursive Hoare Logic). *For a well-formed program, if all method definitions satisfy $\vdash_I \{Pre(m)\}\ Body(m)\ \{Post(m)\}$, and if h, C is a valid configuration, and if $C, h \xrightarrow[c]{} C', h'$, then C', h' is a valid configuration.*

Note that the use of our *isorecursive* Hoare Logic here, reflects the fact that a program must be verifiable statically. However, our earlier results easily allow us to connect (in the proof) with the equirecursive notions, which are closer to the actual runtime. A proof sketch is provided in the appendix.

8 Related Work and Conclusions

This paper has explored the many challenges involved in handling flexible recursive specification constructs in ways which are both amenable for formal mathematical proofs (the equirecursive setting), and implementation in practical static tools (the isorecursive setting). Our work is set in the context of implicit dynamic frames, which supports an interesting combination of recursive predicates, functions and unfolding expressions, each of which provides additional challenges. However, the issues we have described show up in many other settings, including those which do not support all three features simultaneously.

The first formally rigorous treatment of recursive predicates in the context of permission-based logics was proposed in [15] for separation logic [8, 13]; this treatment was further developed in [16]. In both works, and in many subsequent formal papers, the (only) meaning of recursive predicates is given by the least fix-point of the unrolling of their bodies, i.e., the equirecursive treatment.

Many existing verification tools based on separation logic, such as jStar [5] and VeriFast [9], support custom recursive definitions in the form of abstract predicates. jStar applies a sequence of inbuilt heuristics (which can be user-defined) to decide on the points in the code at which to fold or unfold recursive definition, while VeriFast requires the user to provide **fold** and **unfold** statements explicitly (which can nonetheless be inferred in some cases). The full unrolling of a recursive definition is not made available to the verifier; the isorecursive interpretation is used for the implementations.

The problem of handling partial functions in a setting with only total functions has received much prior attention, in several areas (see [20] for an excellent summary). We aimed to avoid allowing "undefined" to be a possible outcome in our semantics, for the reasons explained in Section 2. As an alternative, we could have considered taking the approach of *semi-classical logics* (e.g., [23]), and allowing undefined expressions but assertions. In a sense, our solution is somewhat similar, since we use the extra $error_{tp}$ values to circumvent potential undefinedness for expressions.

The combination of fractional permissions [4] with separation logic for concurrent programming was proposed in [3]. These ideas were adapted to concurrent object oriented programming and formalised in [6], and further adapted to the implicit dynamic frames [21] setting and implemented in the form of Chalice [10]. The Chalice approach has been formalised [19] through a Hoare Logic for implicit dynamic frames. However, neither [6], nor [19] give a treatment of recursive predicates and functions. A verification condition generation semantics for implicit dynamic frames was developed and proven sound in [22].

As future work, we would like to investigate how easily one can build a soundness proof for such a particular *implementation* on top of the formalisms we have provided here. We would also like to explore how to connect the notions of isorecursive definitions provided here with other related areas, such as tools for shape and static analysis, in which different but related issues regarding the bounding of recursive definitions arise.

References

1. M. Abadi and M. P. Fiore. Syntactic considerations on recursive types. In *Proceedings of the Eleventh Annual IEEE Symposium on Logic in Computer Science (LICS 1996)*, pages 242–252. IEEE Computer Society Press, July 1996.
2. J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *FMCO*, 2005.
3. R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *POPL*, pages 259–270, 2005.
4. J. Boyland. Checking interference with fractional permissions. In *SAS*, 2003.
5. D. DiStefano and M. J. Parkinson. jStar: Towards practical verification for Java. In *OOPSLA*. ACM Press, 2008.
6. C. Haack and C. Hurlin. Separation logic contracts for a Java-like language with fork/join. In *International Conference on Algebraic Methodology and Software Technology (AMAST'08)*, July 2008.
7. S. Heule, I. T. Kassios, P. Müller, and A. J. Summers. Verification condition generation for permission logics with abstract predicates and abstraction functions. Technical Report 776, ETH Zurich, 2012.
8. S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, pages 14–26. ACM Press, 2001.
9. B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java - invited paper. In *NFM*, 2011.
10. K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In G. Castagna, editor, *European Symposium on Programming (ESOP)*, volume 5502 of *Lecture Notes in Computer Science*, pages 378–393. Springer-Verlag, 2009.
11. K. R. M. Leino, P. Müller, and J. Smans. Verification of concurrent programs with Chalice. In *Foundations of Security Analysis and Design V*, volume 5705 of *LNCS*, pages 195–222. Springer-Verlag, 2009.
12. M. Moskał. Programming with triggers. In *SMT '09: Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, 2009.
13. P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL*, pages 1–19, London, UK, 2001. Springer-Verlag.
14. M. Parkinson. *Local Reasoning for Java*. PhD thesis, University of Cambridge, November 2005.
15. M. Parkinson and G. Bierman. Separation logic and abstraction. In *POPL*, pages 247–258. ACM Press, 2005.
16. M. Parkinson and G. Bierman. Separation logic, abstraction and inheritance. In *POPL*, pages 75–86. ACM Press, 2008.
17. M. Parkinson and A. J. Summers. The Relationship between Separation Logic and Implicit Dynamic Frames. In *ESOP*, 2011.
18. M. Parkinson and A. J. Summers. The Relationship between Separation Logic and Implicit Dynamic Frames. *Logical Methods in Computer Science*, 2012. To appear.
19. A. Raad and S. Drossopoulou. A sip of the chalice. In *FTfJP*, July 2011.
20. M. Schmalz. *Formalizing the logic of event-B*. PhD thesis, ETH Zurich, November 2012.
21. J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *ECOOP*, volume 5653, pages 148–172, July 2009.
22. J. Smans, B. Jacobs, and F. Piessens. Implicit Dynamic Frames. *ToPLAS*, 2012.
23. W.M.Farmer. A partial functions version of church's simple theory of types. *Journal of Symbolic Logic*, 55:1269–91, 1990.

A Full Definitions, Proofs, Further Lemmas

A.1 Iso-Semantics

Definition 7 [Value of Iso-Recursive Expressions] We define evaluation of expressions in a state consisting of heap H, and stack frame σ, through predicate $\_ \Downarrow_{H,\sigma} \_$ as follows :

$x \Downarrow_{H,\sigma} \sigma(x) \qquad null \Downarrow_{H,\sigma} null \qquad true \Downarrow_{H,\sigma} true \qquad false \Downarrow_{H,\sigma} false$

$E.f \Downarrow_{H,\sigma} v$	if $E \Downarrow_{H,\sigma} \iota$ and $H(\iota, f) = v$.
$E.g(E') \Downarrow_{H,\sigma} v$	if $E \Downarrow_{H,\sigma} \iota$ and $E' \Downarrow_{H,\sigma} v'$ and $Body(g) \Downarrow_{H,\sigma'} v'$, where $\sigma' = [\mathsf{this}, \mathsf{X} \mapsto \iota, v]$.
$E = E' \Downarrow_{H,\sigma} true$	if $E \Downarrow_{H,\sigma} v$ and $E' \Downarrow_{H,\sigma} v$ for some v.
$E = E' \Downarrow_{H,\sigma} false$	if $E \Downarrow_{H,\sigma} v$ and $E' \Downarrow_{H,\sigma} v'$ and $v \neq v'$.
$(E_1 \,?\, E_2 : E_3) \Downarrow_{H,\sigma} v$	if $E_1 \Downarrow_{H,\sigma} true$ and $E_2 \Downarrow_{H,\sigma} v$.
$(E_1 \,?\, E_2 : E_3) \Downarrow_{H,\sigma} v$	if $E_1 \Downarrow_{H,\sigma} false$ and $E_3 \Downarrow_{H,\sigma} v$.
$\mathbf{unfolding}\, E_1.P \,\mathbf{in}\, E_2 \Downarrow_{H,\sigma} v$	if$E \Downarrow_{H,\sigma} v$

We now define the *value* of an expression e in the context of heap H as follows:

$$[\![E]\!]_{H,\sigma} = \begin{cases} v & \text{if } E \Downarrow_{H,\sigma} v \\ error_t \quad \text{if } \nexists v. E \Downarrow_{H,\sigma} v \;\text{ and }\; E \text{ has type } t. \end{cases}$$

Definition 10 [Framed and Self-Framing iso-assertions]

We define the notion of iso-assertion expressions E, or assertions A being *iso-framed* in a state consisting of H, Π and σ, written $\models^{\Pi,H,\sigma}_{\mathsf{frml}} E$ and $\models^{\Pi,H,\sigma}_{\mathsf{frml}} A$ as the least fixed point satisfying the following equations:

$\models^{\Pi,H,\sigma}_{\mathsf{frml}} null \qquad \models^{\Pi,H,\sigma}_{\mathsf{frml}} true \qquad \models^{\Pi,H,\sigma}_{\mathsf{frml}} false \qquad \models^{\Pi,H,\sigma}_{\mathsf{frml}} x$

$\models^{\Pi,H,\sigma}_{\mathsf{frml}} E.f$	$\iff$	$\models^{\Pi,H,\sigma}_{\mathsf{frml}} E \;\wedge\; \Pi([\![E]\!]_{H,\sigma}, f) > 0$
$\models^{\Pi,H,\sigma}_{\mathsf{frml}} E.g(E')$	$\iff$	$\models^{\Pi,H,\sigma}_{\mathsf{frml}} E \wedge \models^{\Pi,H,\sigma}_{\mathsf{frml}} E' \wedge\; \Pi, H, \sigma' \models_{\mathsf{I}} Pre(g)$ where $\sigma' = [\mathsf{this}, \mathsf{X} \mapsto [\![E]\!]_{H,\sigma}, [\![E']\!]_{H,\sigma}]$
$\models^{\Pi,H,\sigma}_{\mathsf{frml}} E = E'$	$\iff$	$\models^{\Pi,H,\sigma}_{\mathsf{frml}} E \wedge \models^{\Pi,H,\sigma}_{\mathsf{frml}} E'$
$\models^{\Pi,H,\sigma}_{\mathsf{frml}} E1?E2 : E3$	$\iff$	$\models^{\Pi,H,\sigma}_{\mathsf{frml}} E \;\wedge\; (\Pi, H, \sigma \models_{\mathsf{I}} E \;\Rightarrow \models^{\Pi,H,\sigma}_{\mathsf{frml}} E') \wedge (\Pi, H, \sigma \not\models_I E \;\Rightarrow \models^{\Pi,H,\sigma}_{\mathsf{frml}} E'')$
$\models^{\Pi,H,\sigma}_{\mathsf{frml}} \mathbf{unfolding}\, E.P \,\mathbf{in}\, E'$	$\iff$	$\models^{\Pi,H,\sigma}_{\mathsf{frml}} E \;\wedge \Pi([\![E]\!]_{H,\sigma}, P) \geq 1 \wedge\; \models^{\Pi',H,\sigma}_{\mathsf{frml}} E'$, where $\Pi' = \Pi \odot_H [\![E]\!]_{H,\sigma}.P\sigma$
$\models^{\Pi,H,\sigma}_{\mathsf{frml}} \mathbf{acc}(E.P)$	$\iff$	$\models^{\Pi,H,\sigma}_{\mathsf{frml}} E$
$\models^{\Pi,H,\sigma}_{\mathsf{frml}} E \to A$	$\iff$	$\models^{\Pi,H,\sigma}_{\mathsf{frml}} E \;\wedge\; (\Pi, H, \sigma \models_{\mathsf{I}} E \Rightarrow \models^{\Pi,H,\sigma}_{\mathsf{frml}} A)$
$\models^{\Pi,H,\sigma}_{\mathsf{frml}} A_1 * A_2$	$\iff$	$\models^{\Pi,H,\sigma}_{\mathsf{frml}} A_1 \wedge\; \models^{\Pi,H,\sigma}_{\mathsf{frml}} A_2$

An iso-assertion A is *self-framing*, written $\models_{\mathsf{frml}} A$ if, for all H, Π and σ:
$\Pi, H, \sigma \models_{\mathsf{I}} A \;\Rightarrow\; \models^{\Pi,H,\sigma}_{\mathsf{frml}} A$

A.2 Comparing the Assertion Semantics

Definition 12[Encoding iso-assertions to equi-assertions]

We now define the encoding $\langle\!\langle\ \_\ \rangle\!\rangle$ which maps isorecursive expressions and statements to their equirecursive counterparts:

$$\begin{array}{rcl}
\langle\!\langle\, null \,\rangle\!\rangle & = & null \\
\langle\!\langle\, false \,\rangle\!\rangle & = & false \\
\langle\!\langle\, true \,\rangle\!\rangle & = & true \\
\langle\!\langle\, x \,\rangle\!\rangle & = & x \\
\langle\!\langle\, E.f \,\rangle\!\rangle & = & \langle\!\langle\, E \,\rangle\!\rangle.f \\
\langle\!\langle\, E1 = E2 \,\rangle\!\rangle & = & \langle\!\langle\, E1 \,\rangle\!\rangle = \langle\!\langle\, E2 \,\rangle\!\rangle \\
\langle\!\langle\, E.g(E') \,\rangle\!\rangle & = & \langle\!\langle\, E \,\rangle\!\rangle.g(\langle\!\langle\, E' \,\rangle\!\rangle) \\
\langle\!\langle\, E?E' : E'' \,\rangle\!\rangle & = & \langle\!\langle\, E \,\rangle\!\rangle?\langle\!\langle\, E' \,\rangle\!\rangle : \langle\!\langle\, E'' \,\rangle\!\rangle \\
\langle\!\langle\, \mathbf{unfolding}\, E.P\, \mathbf{in}\, E' \,\rangle\!\rangle & = & \langle\!\langle\, E' \,\rangle\!\rangle \\
 & & \\
\langle\!\langle\, \mathbf{acc}(E.f, q) \,\rangle\!\rangle & = & \mathbf{acc}(E.f, q) \\
\langle\!\langle\, \mathbf{acc}(E.P) \,\rangle\!\rangle & = & \langle\!\langle\, E \,\rangle\!\rangle.P \\
\langle\!\langle\, A * A' \,\rangle\!\rangle & = & \langle\!\langle\, A \,\rangle\!\rangle * \langle\!\langle\, A' \,\rangle\!\rangle \\
\langle\!\langle\, E \to A \,\rangle\!\rangle & = & \langle\!\langle\, E \,\rangle\!\rangle \to \langle\!\langle\, A \,\rangle\!\rangle
\end{array}$$

Lemma 1 In a well-formed program,

1. If Π, H, σ is a good iso-state, and $\Pi(\iota.P) > 0$, then $\langle\!\langle\, \Pi \,\rangle\!\rangle_H = \langle\!\langle\, \Pi \odot_H \iota.P \,\rangle\!\rangle_H$.
2. If $\pi, H, \sigma \models_{\mathsf{E}} \langle\!\langle\, A \,\rangle\!\rangle$, then $\exists \Pi$ s.t. (Π, H, σ) is a good iso-state, $\langle\!\langle\, \Pi \,\rangle\!\rangle_H = \pi$, and $\pi, H, \sigma \models_{\mathsf{I}} A$.

Proof. Part 1, by direct application of the definitions. Part 2 follows by induction on the structure of A; when A is an E, using Theorem 1.1.

Theorem 1

1. $[\![\langle\!\langle\, E \,\rangle\!\rangle]\!]_{H,\sigma} = [\![E]\!]_{H,\sigma}$.
2. If $\Pi, H, \sigma \models_{\mathsf{I}} A$, and (Π, H, σ) is a good iso-state, then $\langle\!\langle\, \Pi \,\rangle\!\rangle_H, H, \sigma \models_{\mathsf{E}} \langle\!\langle\, A \,\rangle\!\rangle$.
3. If (Π, H, σ) is a good iso-state, and $\models_{\mathsf{frmI}}^{\Pi,H,\sigma} A$, then $\models_{\mathsf{frmE}}^{\langle\!\langle\, \Pi \,\rangle\!\rangle,H,\sigma} \langle\!\langle\, A \,\rangle\!\rangle$.
4. If all functions and predicates are well-formed, then if A is iso-self-framing, then $\langle\!\langle\, A \,\rangle\!\rangle$ is equi-self-framing
5. If $A \models_{\mathsf{I}} A'$, then $\langle\!\langle\, A \,\rangle\!\rangle \models_{\mathsf{E}} \langle\!\langle\, A' \,\rangle\!\rangle$.

Proof. Part *1* by showing that termination of E or $\langle\!\langle\, E \,\rangle\!\rangle$ implies that $[\![\langle\!\langle\, E \,\rangle\!\rangle]\!]_{H,\sigma} = [\![E]\!]_{H,\sigma}$ - this is by induction on the derivation of $\_ \Downarrow_{H,\sigma} \_$. This results also gives that the term E loops forever if and only if $\langle\!\langle\, E \,\rangle\!\rangle$ loops forever. Part *2* by induction on the structure of A.

Part *3* follows by induction on A, and requires an analogous lemma for expressions.

Part *4*, follows from Lemma 1 and part *3*.

For Part *5*, take any H, H', π, σ, st: $\pi, H, \sigma \models_{\mathsf{E}} \langle\!\langle\, A \,\rangle\!\rangle$. We want to show that $\pi, H, \sigma \models_{\mathsf{E}} \langle\!\langle\, A' \,\rangle\!\rangle$. Because we have $\pi, H, \sigma \models_{\mathsf{E}} \langle\!\langle\, A \,\rangle\!\rangle$, by application of Lemma 1, we obtain that there exists a Π, s.t. $\langle\!\langle\, \Pi \,\rangle\!\rangle_H = \pi$, and $H \models \Pi$, and $\Pi, H, \sigma \models_{\mathsf{I}} A$. By the assumption, we obtain that $\Pi, H, \sigma \models_{\mathsf{I}} A'$. This, together with part 2 gives that $\langle\!\langle\, \Pi \,\rangle\!\rangle_H, H, \sigma \models_{\mathsf{E}} \langle\!\langle\, A' \,\rangle\!\rangle$.

A.3 Isorecursive Hoare Logic

Lemma 2 [Generation Lemma]

1. $\vdash_I \{A\}\ \texttt{skip}\ \{A'\} \ \Leftrightarrow\ A \models_{\mathsf{I}} A'$
2. $\vdash_I \{A\}\ x{:=}E\ \{A'\} \ \Leftrightarrow$
   $\exists A''.(A''\ s.f.\ \ \wedge\ \ A \models_{\mathsf{I}} A''[E/x]\ \ \wedge\ \ A''[E/x] \models_{\mathsf{frml}} E\ \ \wedge\ \ A'' \models_{\mathsf{I}} A')$
3. $\vdash_I \{A\}\ x.f{:=}y\ \{A'\} \ \Leftrightarrow$
   $\exists A''.(A''\ s.f.\ \ \wedge\ \ A \models_{\mathsf{I}} \mathbf{acc}(x.f,1) * A''\ \ \wedge\ \ \mathbf{acc}(x.f,1) * x.f = y * A'' \models_{\mathsf{I}} A')$
4. $\vdash_I \{A\}\ (\texttt{if}\ B\ \texttt{then}\ s_1\ \texttt{else}\ s_2)\ \{A'\} \ \Leftrightarrow\ \vdash_I \{A \wedge B\}\ s_1\ \{A'\}\ \wedge$
   $\vdash_I \{A \wedge \neg B\}\ s_2\ \{A'\}$
5. $\vdash_I \{A\}\ (r;s)\ \{A'\} \ \Leftrightarrow\ \exists A''.(\vdash_I \{A\}\ r\ \{A''\}\ \wedge\ \vdash_I \{A''\}\ s\ \{A'\})$
6. $\vdash_I \{A\}\ \texttt{return}\ x\ \{A'\} \ \Leftrightarrow\ A * \mathsf{result} = x \models_{\mathsf{I}} A'$
7. $\vdash_I \{A\}\ x{:=}\ \texttt{fork}\ y.m(z)\ \ \{A'\} \ \Leftrightarrow\ A \models_{\mathsf{I}} y \neq \mathit{null} * (\mathit{Pre}(m)[y/\mathsf{this}][z/\mathsf{X}])\ \wedge$
   $\mathit{Thread}(x,m,y,z) \models_{\mathsf{I}} A'$
8. $\vdash_I \{A\}\ w{:=}\ \texttt{join}\ x\ \ \{A'\} \ \Leftrightarrow$
   $\exists A'', m.(A \models_{\mathsf{I}} A'' * \mathit{Thread}(x,m,y,z)\ \wedge$
   $A'' * \mathit{Post}(m)[y/\mathsf{this}][z/\mathsf{X}][w/\mathsf{result}] \models_{\mathsf{I}} A')$
9. $\vdash_I \{A\}\ x{:=}\ \texttt{new}\ c\ \{A'\} \ \Leftrightarrow\ A * (\overline{*\mathbf{acc}(x.f_i,1)}) \models_{\mathsf{I}} A'$
10. $\vdash_I \{A\}\ \mathbf{fold}\,\mathbf{acc}(x.P,q)\ \{A'\} \ \Leftrightarrow$
    $\exists A'', B.(A''\ s.f.\ \ \wedge\ \ A \models_{\mathsf{I}} (A'' * \mathit{Body}(P)[x/\mathsf{this}] * B)\ \wedge$
    $(A'' * \mathbf{acc}(x.P,q) * \mathbf{unfolding}\, x.P\, \mathbf{in}\, B) \models_{\mathsf{I}} A'$
11. $\vdash_I \{A\}\ \mathbf{unfold}\,\mathbf{acc}(x.P,q)\ \{A'\} \ \Leftrightarrow$
    $\exists A'', B.(A''\ s.f.\ \ \wedge\ \ A \models_{\mathsf{I}} (A'' * \mathbf{acc}(x.P,q) * \mathbf{unfolding}\, x.P\, \mathbf{in}\, B)\ \wedge$
    $(A'' * \mathit{Body}(P)[x/\mathsf{this}] * B) \models_{\mathsf{I}} A'$

A.4 Equirecursive Hoare Logic

(See Figure 4)

A.5 Operational Semantics

The operational semantics for local transitions appears in figure 5.

A.6 Hoare Logic for Equi Assertions and Programs

A.7 Soundness

Definition 25 (Mapping Runtime Configurations to Heaps). *The mapping Heap maps from runtime configurations to heaps, according to the following definition:*

$\mathit{Heap}(h,C) = H$ *where:*
$\forall \iota.\ \forall f \in \mathit{fields}(\mathit{cls}(\iota)).\ H[\iota,f] = h[\iota,f]$ *, and,*
$\forall \{\sigma, s\}_t \in C.\ H[t,\texttt{recv}] = \sigma(\mathit{this}) \wedge H[t,\texttt{param}] = \sigma(\mathsf{X}) \wedge H[t,\texttt{meth}] = \sigma(\mathit{method})$

$$\frac{}{\vdash_E \{a\}\ \texttt{skip}\ \{a\}}\ (skipE)$$

$$\frac{a[e/x] \models_{\mathsf{frmE}} e}{\vdash_E \{a[e/x]\}\ x{:=}e\ \{a\}}\ (varassE)$$

$$\frac{}{\vdash_E \{x \neq null * \mathbf{acc}(x.f,1)\}\ x.f{:=}y\ \{\mathbf{acc}(x.f,1) * x.f = y\}}\ (fldassE)$$

$$\frac{}{\vdash_E \{a\}\ \texttt{return}\ x\ \{a * \mathsf{result} = x\}}\ (retE)$$

$$\frac{\overline{f_i} = fields(c)}{\vdash_E \{true\}\ x{:=}\ \texttt{new}\ c\ \{(\overline{*\mathbf{acc}(x.f_i,1)})\}}\ (newE)$$

$$\frac{\vdash_E \{a * b\}\ s_1\ \{a'\} \quad \vdash_E \{a * \neg b\}\ s_2\ \{a'\}}{\vdash_E \{a\}\ (\texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2)\ \{a'\}}\ (ifE)$$

$$\frac{a = Pre(m)[y/\mathsf{this}][z/\mathsf{X}]}{\vdash_E \{y \neq null * a\}\ x{:=}\ \texttt{fork}\ y.m(z)\ \ \{Thread(x,m,y,z)\}}\ (forkE)$$

$$\frac{a' = Post(m)[y/\mathsf{this}][z/\mathsf{X}][w/\mathsf{result}]}{\vdash_E \{Thread(x,m,y,z)\}\ w{:=}\ \texttt{join}\ x\ \ \{a'\}}\ (joinE)$$

$$\frac{\vdash_E \{a\}\ r\ \{a'\} \quad \vdash_E \{a'\}\ s\ \{a''\}}{\vdash_E \{a\}\ (r;s)\ \{a''\}}\ (seqE)$$

$$\frac{a_1 \models_{\mathsf{I}} a_3 \quad \vdash_E \{a_3\}\ s\ \{a_4\} \quad a_4 \models_{\mathsf{I}} a_2}{\vdash_E \{a_1\}\ s\ \{a_2\}}\ (consE)$$

$$\frac{\vdash_E \{a\}\ s\ \{a'\} \quad \models_{\mathsf{frmE}} a'' \quad mods(s) \cap FV(a'') = \emptyset}{\vdash_E \{a'' * a\}\ s\ \{a' * a''\}}\ (frameE)$$

Fig. 4. Hoare Logic for Equirecursive semantics

Definition 26 (Permissions Collection for Equi-Assertions).

$$\frac{}{h, \{\sigma, (\texttt{skip}; s)\}_t \xrightarrow[l]{} h, \{\sigma, s\}_t} \; (skipS)$$

$$\frac{[\![e]\!]_{h,\sigma} = v}{h, \{\sigma, (x{:=}e; s)\}_t \xrightarrow[l]{} h, \{\sigma[x \mapsto v], s\}_t} \; (varassS)$$

$$\frac{\sigma(x) = \iota \quad h' = h[(\iota, f) \mapsto \sigma(y)]}{h, \{\sigma, (x.f{:=}y; s)\}_t \xrightarrow[l]{} h', \{\sigma, s\}_t} \; (fldassS)$$

$$\frac{[\![b]\!]_{h,\sigma} = true}{h, \{\sigma, ((\texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2); s_3)\}_t \xrightarrow[l]{} h, \{\sigma, (s_1; s_3)\}_t} \; (iftrueS)$$

$$\frac{[\![b]\!]_{h,\sigma} = false}{h, \{\sigma, ((\texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2); s_3)\}_t \xrightarrow[l]{} h, \{\sigma, (s_2; s_3)\}_t} \; (iffalseS)$$

Fig. 5. Local transitions

The function $\mathcal{P}_{\mathsf{E}}$ *collects the permissions* explicitly *or* implicitly *required by equi-assertions* $\quad \mathcal{P}_{\mathsf{E}} \; : \; EquiAssertion \times Heap \times Env \to Perms$

$$\begin{array}{lcl}
\mathcal{P}_{\mathsf{E}}(a, h, \sigma) & = & \{(t, \mathit{this}), (t, X), (t, \texttt{meth}) \mapsto 1 \mid \\
 & & \qquad \exists z \in VarId, t \in ThrdId.\ \sigma(z) = t\} \\
 & & \cup\ \mathcal{P}^{aux}_{\mathsf{E}}(a, h, \sigma) \\
\mathcal{P}^{aux}_{\mathsf{E}}(e, h, \sigma) & = & \emptyset \\
\mathcal{P}^{aux}_{\mathsf{E}}(\mathbf{acc}(e.f, q), h, \sigma) & = & \{\ ([\![e]\!]_{h,\sigma}, f) \mapsto q\ \} \\
\mathcal{P}^{aux}_{\mathsf{E}}(e \to a, h, \sigma) & = & \mathcal{P}^{aux}_{\mathsf{E}}(a, h, \sigma) \ \textit{if}\ [\![e]\!]_{h,\sigma} = true, \qquad \emptyset \ \textit{otherwise} \\
\mathcal{P}^{aux}_{\mathsf{E}}(a * a', h, \sigma) & = & \mathcal{P}^{aux}_{\mathsf{E}}(a, h, \sigma) + \mathcal{P}^{aux}_{\mathsf{E}}(a', h, \sigma) \\
\mathcal{P}^{aux}_{\mathsf{E}}(e.P, h, \sigma) & = & \mathcal{P}^{aux}_{\mathsf{E}}(Body(P), h, \sigma'), \quad \textit{where}\ \sigma'(\mathit{this}) = [\![e]\!]_{h,\sigma}.
\end{array}$$

Theorem 3 [Soundness of Verification] For a well-formed program, if all method definitions satisfy $\vdash_I \{Pre(m)\}\ Body(m)\ \{Post(m)\}$, and if h, C is a valid configuration, and $h, C \xrightarrow[c]{} h', C'$, then h', C' is a valid configuration.

Proof. Sketch: By induction on the length of the execution $C \xrightarrow[c]{} C'$. The proof relies on three main points. Firstly, for every local and paired transition, the corresponding Hoare triple never requires more permission in the postcondition than in the precondition. Secondly, for every such transition, a heap location is only modified if full permission to the location was required in the precondition of the corresponding triple. Thirdly, since all pre-conditions are self-framing, the preconditions of all other entities remain unaffected by these modifications. This relies on the (easy) lemma that, if $\pi, H, \sigma \models_{\mathsf{E}} a$ holds, and a is self-framing, then in a heap H' which agrees with H on at least the values of locations to which π requires permission, $\pi, H', \sigma \models_{\mathsf{E}} a$ also holds (this is sometimes called *stability* of assertions).