Electronic Communications of the EASST
Volume 66 (2013)

Proceedings of the
Automated Verification of Critical Systems
(AVoCS 2013)

Simplifying proofs of linearisability using layers of abstraction

Brijesh Dongol and John Derrick

15 pages

ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Simplifying proofs of linearisability using layers of abstraction

Brijesh Dongol and John Derrick

Department of Computer Science
The University of Sheffield, S1 4DP, UK
`B.Dongol@sheffield.ac.uk, J.Derrick@dcs.shef.ac.uk`

Abstract: Linearisability has become the standard correctness criterion for concurrent data structures, ensuring that every history of invocations and responses of concurrent operations has a matching sequential history. Existing proofs of linearisability require one to identify so-called linearisation points within the operations under consideration, which are atomic statements whose execution causes the effect of an operation to be felt. However, identification of linearisation points is a non-trivial task, requiring a high degree of expertise. For sophisticated algorithms such as Heller et al's lazy set, it even is possible for an operation to be linearised by the concurrent execution of a statement outside the operation being verified. This paper proposes a method for verifying linearisability that does not require identification of linearisation points. Instead, using an interval-based logic, we show that every behaviour of each concrete operation over any interval is a possible behaviour of a corresponding abstraction that executes with coarse-grained atomicity. This approach is applied to Heller et al's lazy set to show that verification of linearisability is possible without having to consider linearisation points within the program code.

Keywords: Linearisability, Interval-based verification, Fine-grained atomicity

1 Introduction

Development of correct fine-grained concurrent data structures has received an increasing amount of attention over the past few years as the popularity of multi/many-core architectures has increased. An important correctness criterion for such data structures is *linearisability* [HW90], which guarantees that every history of invocations and responses of the concurrent operations on the data structure can be rearranged without violating the ordering within a process such that the rearranged history is a valid sequential history. A number of proof techniques developed over the years match concurrent and sequential histories by identifying an atomic *linearising statement* within the concrete code of each operation, whose execution corresponds to the effect of the operation taking place. However, due to the subtlety and complexity of concurrent data structures, identification of linearising statements within the concrete code is a non-trivial task, and it is even possible for an operation to be linearised by the execution of other concurrent operations. An example of such behaviour occurs in Heller et al's lazy set algorithm, which implements a set as a sorted linked list [HHL+07] (see Fig. 1). In particular, its `contains` operation may be linearised by the execution of a concurrent `add` or `remove` operation and the precise location of the linearisation point is dependent on how much of the list has been traversed by the `contains` operation. In this paper, we present a method for simplifying proofs of linearisability using Heller et al's lazy set as an example.

An early attempt at verifying linearisability of Heller et al's lazy set is that of Vafeiadis et al, who extend each linearising statement with code corresponding to the execution of the abstract operation so that execution of a linearising statement causes the corresponding abstract operation to be executed [VHHS06]. However, this technique is incomplete and cannot be used to verify the `contains` operation, and hence, its correctness is only treated informally [VHHS06]. These difficulties reappear in more recent techniques: "In [Heller et al's lazy set] algorithm, the correct abstraction map lies outside of the abstract domain of our implementation and, hence, was not found." [Vaf10]. The first complete linearisability proof of the lazy set was given by Colvin et al [CGLM06], who map the concrete program to an abstract set representation using simulation to prove data refinement. To verify the `contains` operation, a combination of forwards and backwards simulation is used, which involves the development of an intermediate program *IP* such that there is a backwards simulation from the abstract representation to *IP*, and a forwards simulation from *IP* to the concrete program. More recently, O'Hearn et al use a so-called hindsight lemma (related to backwards simulation) to verify a variant of Heller's lazy set algorithm [ORV+10]. Derrick et al use a method based on *non-atomic* refinement, which allows a single atomic step of the concrete program to be mapped to several steps of the abstract [DSW11].

Application of the proof methods in [VHHS06, CGLM06, ORV+10, DSW11] remains difficult because one must acquire a high degree of expertise of the program being verified to correctly identify its linearising statements. For complicated proofs, it is difficult to determine whether the implementation is erroneous or the linearising statements have been incorrectly chosen. Hence, we propose an approach that eliminates the need for identification of linearising statements in the concrete code by establishing a refinement between the fine-grained implementation and an abstraction that executes with coarse-grained atomicity [DD12]. The idea of mapping fine-grained programs to a coarse-grained abstraction has been proposed by Groves [Gro08] and separately Elmas et al [EQS+10], where the refinements are justified using *reduction* [Lip75]. However, unlike our approach, their methods must consider each pair of interleavings, and hence, are not compositional. Turon and Wand present a method of abstraction in a compositional rely/guarantee framework with separation logic [TW11], but only verify a stack algorithm that does not require backwards reasoning.

Capturing the behaviour of a program over its interval of execution is crucial to proving linearisability of concurrent data structures. In fact, as Colvin et al point out: "The key to proving that [Heller et al's] lazy set is linearisable is to show that, for any failed `contains(x)` operation, `x` is absent from the set at some point during its execution." [CGLM06]. Hence, it seems counter-intuitive to use logics that are only able to refer to the pre and post states of each statement (as done in [VHHS06, CGLM06, DSW11, Vaf10]). Instead, we use a framework based on [DDH12] that allows reasoning about the fine-grained atomicity of pointer-based programs over their intervals of execution. By considering complete intervals, i.e., those that cover both the invocation and response of an operation, one is able to determine the future behaviour of a program, and hence, backwards reasoning can often be avoided. For example, Bäumler et al [BSTR11] use an interval-based approach to verify a lock-free queue without resorting to backwards reasoning, as is required by frameworks that only consider the pre/post states of a statement [DGLM04]. However, unlike our approach, Bäumler et al must identify the linearising statements in the concrete program, which is a non-trivial step.

```
add(x):
 A1:  (n1, n3) :=
         locate(x);
 A2:  if n3.val != x
 A3:     n2:=
          new Node(x);
 A4:     n2.nxt := n3;
 A5:     n1.nxt := n2;
 A6:     res := true
 A7:  else res := false
     endif;
 A8:  n1.unlock();
 A9:  n3.unlock();
A10: return res
```

```
remove(x):
 R1: (n1, n2) :=
        locate(x);
 R2: if n2.val = x
 R3:    n2.mrk := true;
 R4:    n3 := n2.nxt;
 R5:    n1.nxt := n3;
 R6:    res := true
 R7: else res := false
     endif;
 R8: n1.unlock();
 R9: n2.unlock();
R10: return res
```

```
contains(x):
C1:  n1 := Head;
C2:  while (n1.val < x)
C3:    n1 := n1.nxt
     enddo;
C4:  res := (n1.val = x)
          and !n1.mrk
C5:  return res
```

```
locate(x):
     while (true) do
 L1:   pred := Head;                    L7:   curr.lock();
 L2:   curr := pred.nxt;                L8:   if !pred.mrk and !curr.mrk
 L3:   while (curr.val < x) do                   and pred.nxt = curr
 L4:     pred := curr;                  L9:     return (pred, curr)
 L5:     curr := pred.nxt enddo;       L10:   else pred.unlock();
 L6:   pred.lock();                    L11:     curr.unlock() endif enddo
```

Figure 1: A lazy set algorithm [HHL+07]

An important difference between our framework and those mentioned above is that we assume a truly concurrent execution model and only require interleaving for conflicting memory accesses [DD12, DDH12]. Each of the other frameworks mentioned above assume a strict interleaving between program statements. Thus, our approach captures the behaviour of program in a multicore/multiprocesor architecture more faithfully.

The main contribution of this paper is the use of the techniques in [DD12] to simplify verification of a complex set algorithm [HHL+07]. This algorithm presents a challenge for linearisability because the linearisation point of the `contains` operation is potentially outside the operation itself [DSW11]. We propose a method in which the proof is split into several layers of abstraction so that linearisation points of the fine-grained implementation need not be identified. As summarised in Fig. 3, one must additionally prove that the coarse-grained abstraction is linearisable, however, due to the coarse granularity of atomicity, the linearising statements are straightforward to identify and the linearisability proof itself is simpler [DD12]. Other contributions of this paper include a method for reasoning about truly concurrent program executions and an extension of the framework in [DDH12] to enable reasoning about pointer-based programs, which includes methods for reasoning about expressions non-deterministically [HBDJ13].

2 A list-based concurrent set

Heller et al [HHL+07] implement a set as a concurrent algorithm operating on a shared data structure (see Fig. 1) with operations `add` and `remove` to insert and delete elements from the set, and an operation `contains` to check whether an element is in the set. The concurrent

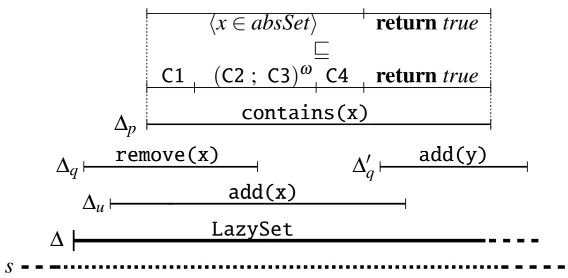

Figure 2: `contains(x)` execution over Δ_p returning *true*

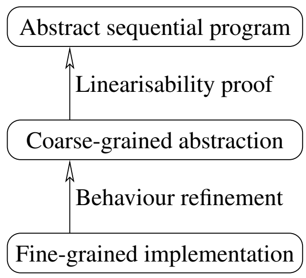

Figure 3: Proof steps

implementation uses a shared linked list of node objects with fields *val*, *nxt*, *mrk*, and *lck*, where *val* stores the value of the node, *nxt* is a pointer to the next node in the list, *mrk* denotes the marked bit and *lck* stores the identifier of the process that currently holds the lock to the node (if any) [HHL+07]. The list is sorted in strictly ascending values order (including marked nodes).

Operation `locate(x)` is used to obtain pointers to two nodes whose values may be used to determine whether or not `x` is in the list — the value of the predecessor node `pred` must always be less than `x`, and the value of the current node `curr` may either be greater than `x` (if `x` is not in the list) or equal to `x` (if `x` is in the list). Operation `add(x)` calls `locate(x)`, then if `x` is not already in the list (i.e., value of the current node `n3` is strictly greater than `x`), a new node `n2` with value field `x` is inserted into the list between `n1` and `n3` and `true` is returned. If `x` is already in the list, the `add(x)` operation does nothing and returns `false`. Operation `remove(x)` also starts by calling `locate(x)`, then if `x` is in the list the current node `n2` is removed and `true` is returned to indicate that `x` was found and removed. If `x` is not in the list, the `remove` operation does nothing and returns `false`. Note that operation `remove(x)` distinguishes between a logical removal, which sets the marked field of `n2` (the node corresponding to `x`), and a physical removal, which updates the `nxt` field of `n1` so that `n2` is no longer reachable. Operation `contains(x)` iterates through the list and if a node with value greater or equal to `x` is found, it returns `true` if the node is unmarked and its value is equal to `x`, otherwise returns `false`.

The complete specification consists of a number of processes, each of which may execute its operation on the shared data structure. For the concrete implementation, therefore, the set operations can be executed concurrently by a number of processes, and hence, the intervals in which the different operations execute may overlap. Our basic semantic model uses *interval predicates* (see Section 3), which allows formalisation of a program's behaviour with respect to an *interval* (which is a contiguous set of times), and an infinite *stream* (that maps each time to a state). For example, consider Fig. 2, which depicts an execution of the lazy set over interval Δ in stream s, a process p that executes a `contains(x)` that returns *true* over Δ_p, a process q that executes `remove(x)` and `add(y)` over intervals Δ_q and Δ'_q, respectively, and a process u that executes `add(x)` over interval Δ_u. Hence, the shared data structure may be changing over Δ_p while process p is checking to see whether x is in the set.

Correctness of such concurrent executions is judged with respect to *linearisability*, the crux of which requires the existence of an atomic *linearisation point* within each interval of an operation's execution, corresponding to the point at which the effect of the operation takes place

[HW90]. The ordering of linearisation points defines a sequential ordering of the concurrent operations and linearisability requires that this sequential ordering is valid with respect to the data structure being implemented. For the execution in Fig. 2, assuming that the set is initially empty, because `contains(x)` returns *true*, a valid linearisation corresponds to a sequential execution $Seq_1 \mathrel{\widehat{=}}$ `add(x); contains(x); remove(x); add(y)` obtained by picking linearisation points within Δ_u, Δ_p, Δ_q and Δ'_q in order. Note that a single concurrent history may be linearised by more than one valid sequential history, e.g., the execution in Fig. 2 can correspond to the sequential execution $Seq_2 \mathrel{\widehat{=}}$ `remove(x); add(x); contains(x); add(y)`. The abstract sets after completion of Seq_1 and Seq_2 are $\{y\}$ and $\{x,y\}$, respectively. Unlike Seq_1, operation `remove(x)` in Seq_2 returns *false*. Note that a linearisation of Δ'_q cannot occur before Δ_q because `remove(x)` responds before the invocation of `add(y)`.

Herlihy and Wing formalise linearisability in terms of histories of invocation and response events of the operations on the data structure in question [HW90]. Reasoning about such histories directly is infeasible, and hence, existing methods (e.g., [CGLM06, DSW11, VHHS06]) prove linearisability by identifying an atomic *linearising statement* within the operation being verified and showing that this statement can be mapped to the execution of a corresponding abstract operation. However, due to the fine granularity of the atomicity and inherent non-determinism of concurrent algorithms, identification of such a statement is difficult. The linearising statement for some operations may actually be outside the operation, e.g., none of the statements `C1`-`C5` are valid linearising statements of `contains(x)`; instead `contains(x)` is linearised by the execution of a statement within `add(x)` or `remove(x)` [DSW11].

As summarised in Fig. 3, we decompose proofs of linearisability into two steps, the first of which proves that a fine-grained implementation refines a program that executes the same operations but with coarse-grained atomicity. The second step of the proof is to show that the abstraction is linearisable. The atomicity of a coarse-grained abstraction cannot be guaranteed in hardware (without the use of contention inducing locks), however, its linearisability proof is much simpler [DDH12]. Because we prove behaviour refinement, any behaviour of the fine-grained implementation is a possible behaviour of the coarse-grained abstraction, and hence, an implementation is linearisable whenever the abstraction is linearisable. Our technique does not require identification of the linearising statements in the implementation.

A possible coarse-grained abstraction of `contains(x)` is an operation that is able to test whether `x` is in the set in a single atomic step (see Fig. 6), unlike the implementation in Fig. 1, which uses a sequence of atomic steps to iterate through the list to search for a node with value `x`. Therefore, as depicted in Fig. 2, an execution of `contains` that returns *true*, i.e., `C1` ; $($`C2` ; `C3`$)^{\omega}$; `C4` ; **return** *true*, is required to refine a coarse-grained abstraction $\langle x \in absSet \rangle$; **return** *true*, where `C1` - `C4` are the labels of `contains` in Fig. 1 and $\langle x \in absSet \rangle$ is a guard that is atomically able to test whether x is in the abstract set. In particular, $\langle x \in absSet \rangle$ holds in an interval Ω and stream s iff there is a time t in Ω such that $x \in absSet.(s.t)$. Streams are formalised in Section 3. Note that both $\langle x \in absSet \rangle$ and $\langle x \notin absSet \rangle$ may hold within Δ_p; the refinement in Fig. 2 would only be invalid if for all $t \in \Delta_p$, $x \notin absSet.(s.t)$ holds.

Proving refinement between a coarse-grained abstraction and an implementation is non-trivial due to the execution of other (interfering) concurrent processes. Furthermore, our execution model allows non-conflicting statements (e.g., concurrent writes to different locations) to be executed in a truly concurrent manner. We use compositional rely/guarantee-style reasoning

$$\begin{array}{rcl}
\mathsf{CLoop}(p,x) & \widehat{=} & ([(n1_p \mapsto val) < x]\,;\, n1_p := (n1_p \mapsto nxt))^{\omega}\,;\,[(n1_p \mapsto val) \geq x] \\
\mathsf{Contains}(p,x) & \widehat{=} & cl_1{:}\, n1_p := Head\,;\; cl_2{:}\,\mathsf{CLoop}(p,x)\,; \\
 & & cl_3{:}\, res_p := (\neg(n1_p \mapsto mrk) \wedge (n1_p \mapsto val) = x)
\end{array}$$

$$\begin{array}{rcl}
HTInit & \widehat{=} & (Head \longmapsto (-\infty, Tail, false, null)) \wedge (Tail \longmapsto (\infty, null, false, null)) \\
\mathsf{S}(p) & \widehat{=} & [\![n1_p, n2_p, n3_p, res_p \mid (\textstyle\bigsqcap_{x:\mathbb{Z}} \mathsf{Add}(p,x) \sqcap \mathsf{Remove}(p,x) \sqcap \mathsf{Contains}(p,x))^{\omega}]\!] \\
\mathsf{Set}(P) & \widehat{=} & [\![Head, Tail \mid \textsc{Rely}\ \overleftarrow{HTInit} \bullet \|_{p:P}\ \mathsf{S}(p)]\!]
\end{array}$$

Figure 4: Formal model of the lazy set operations

[Jon83] to formalise the behaviour of the environment of a process and allow the execution of an arbitrary number of processes in the environment. Note that unlike Jones [Jon83], who assumes rely conditions are two-state relations, rely conditions in our framework are interval predicates that are able to refer to an arbitrary number of states because the size of the interval is not fixed.

3 Interval-based framework

To simplify reasoning about the linked list structure of the lazy list, the domain of each state distinguishes between variables and addresses. We use a language with an abstract syntax that closely resembles program code, and use interval predicates to formalise interval-based behaviour. Fractional permissions are used to control conflicting accesses to shared locations.

Commands. We assume variable names are taken from the set Var, values have type Val, addresses have type $Addr \mathrel{\widehat{=}} \mathbb{N}$, $Var \cap Addr = \varnothing$ and $Addr \subseteq Val$. A *state* over $VA \subseteq Var \cup Addr$ has type $State_{VA} \mathrel{\widehat{=}} VA \to Val$ and a *state predicate* has type $State_{VA} \to \mathbb{B}$.

The objects of a data structure may contain fields, which we assume are of type $Field$. We assume that every object with m fields is assigned m contiguous blocks of memory and use $offset{:}\, Field \to \mathbb{N}$ to obtain the offset of $f \in Field$ within this block [Vaf07], e.g., for the fields of a node object, we assume that $offset.val = 0$, $offset.nxt = 1$, $offset.mrk = 2$ and $offset.lck = 3$.

We assume the existence of a function $eval$ that evaluates a given expression in a given state. The full details of expression evaluation are elided. To simplify modelling of pointer-based programs, for an address-valued expression ae, we introduce expressions $*ae$, which returns the value at address ae, $ae{\cdot}f$, which returns the address of f with respect to ae. For a state σ, we define $eval.(*ae).\sigma \mathrel{\widehat{=}} \sigma.(eval.ae.\sigma)$ and $(ae{\cdot}f).\sigma \mathrel{\widehat{=}} eval.ae.\sigma + offset.f$. We also define shorthand $ae \mapsto f \mathrel{\widehat{=}} *(ae{\cdot}f)$, which returns the value at $ae{\cdot}f$ in state σ.

Assuming that $Proc$ denotes the set of process ids, for a set of variables Z, state predicate c, variable or address-valued expression vae, expression e, label l, and set of processes $P \subseteq Proc$, the abstract syntax of a command is given by Cmd below, where $C, C_1, C_2, C_p \in Cmd$.

$$Cmd \quad ::= \quad \mathsf{Idle} \mid [c] \mid \langle c \rangle \mid vae := e \mid C_1\,;\, C_2 \mid C_1 \sqcap C_2 \mid C^{\omega} \mid \|_{p:P}\ C_p \mid [\![Z \mid C]\!] \mid l{:}\,C$$

Hence a command is either Idle, a guard $[c]$, an atomically evaluated guard $\langle c \rangle$, an assignment $vae := e$, a sequential composition $C_1\,;\, C_2$, a non-deterministic choice $C_1 \sqcap C_2$, a possibly infinite iteration C^{ω}, a parallel composition $\|_{p:P}\ C_p$, a command C within a context Z (denoted $[\![Z \mid C]\!]$), or a labelled command $l{:}\,C$. In $[\![Z \mid C]\!]$, the context Z is the set of variables that C may modify.

A formalisation of part of the lazy set [HHL+07] using the syntax above is given in Fig. 4, where $P \subseteq Proc$. Operations `add(x)`, `remove(x)` and `contains(x)` executed by process p are modelled by commands $\mathsf{Add}(p,x)$, $\mathsf{Remove}(p,x)$ and $\mathsf{Contains}(p,x)$, respectively. We assume that $n \longmapsto (vv,nn,mm,ll)$ denotes $(n \mapsto val = vv) \wedge (n \mapsto nxt = nn) \wedge (n \mapsto mrk = mm) \wedge (n \mapsto lck = ll)$. Details of $\mathsf{Add}(p,x)$ and $\mathsf{Remove}(p,x)$ are elided and the RELY construct is formalised in Section 5. Note that unlike the methods in [CGLM06, DSW11], where labels identify the atomicity, we use labels to simplify formalisation of the rely conditions of each process, and may correspond to a number of atomic steps. Furthermore, guard evaluation is formalised with respect to the set of states apparent to a process (see Section 4), and hence, unlike [VHHS06, CGLM06, DSW11], we need not split complex expressions into their atomic components. For example, in [VHHS06, CGLM06, DSW11], the expression at `C4` (Fig. 1) must be split into two expressions `curr.val = x` and `!curr.mrk` to explicitly model the fact that interference may occur between accesses to `curr.val` and `curr.mrk`.

Interval predicates. A (discrete) *interval* (of type *Intv*) is a contiguous set of time (of type $Time \mathrel{\widehat=} \mathbb{Z}$), i.e., $Intv \mathrel{\widehat=} \{\Delta \subseteq Time \mid \forall t,t'{:}\Delta \bullet \forall u{:}Time \bullet t \le u \le t' \Rightarrow u \in \Delta\}$. Using '.' for function application, we let $\mathsf{lub}.\Delta$ and $\mathsf{glb}.\Delta$ denote the *least upper* and *greatest lower* bounds of an interval Δ, respectively, where $\mathsf{lub}.\varnothing \mathrel{\widehat=} -\infty$ and $\mathsf{glb}.\varnothing \mathrel{\widehat=} \infty$. We define $\mathsf{inf}.\Delta \mathrel{\widehat=} (\mathsf{lub}.\Delta = \infty)$, $\mathsf{fin}.\Delta \mathrel{\widehat=} \neg\mathsf{inf}.\Delta$ and $\mathsf{empty}.\Delta \mathrel{\widehat=} (\Delta = \varnothing)$. For a set K and $i,j \in K$, we let $[i,j]_K \mathrel{\widehat=} \{k{:}K \mid i \le k \le j\}$ denote the closed interval from i to j containing elements from K. One must often reason about two *adjoining* intervals, i.e., intervals that immediately precede or follow a given interval. We say Δ adjoins Δ' iff $\Delta \propto \Delta'$, where

$$\Delta \propto \Delta' \quad \mathrel{\widehat=} \quad (\forall t{:}\Delta, t'{:}\Delta' \bullet t < t') \wedge (\Delta \cup \Delta' \in Intv)$$

Note that adjoining intervals Δ and Δ' must be disjoint, and by conjunct $\Delta \cup \Delta' \in Intv$, the union of Δ and Δ' must be contiguous. Note that both $\Delta \propto \varnothing$ and $\varnothing \propto \Delta$ hold trivially for any interval Δ.

A *stream* of behaviours over $VA \subseteq Var \cup Addr$ is given by a total function of type $Stream_{VA} \mathrel{\widehat=} Time \to State_{VA}$, which maps each time to a state over VA. To reason about specific portions of a stream, we use *interval predicates*, which have type $IntvPred_{VA} \mathrel{\widehat=} Intv \to Stream_{VA} \to \mathbb{B}$. Note that because a stream encodes the behaviour over all time, interval predicates may be used to refer to the states outside a given interval. We assume pointwise lifting of operators on stream and interval predicates in the normal manner, define *universal implication* $g_1 \Rrightarrow g_2 \mathrel{\widehat=} \forall \Delta{:}Intv, s{:}Stream \bullet g_1.\Delta.s \Rightarrow g_2.\Delta.s$ for interval predicates g_1 and g_2, and say $g_1 \equiv g_2$ holds iff both $g_1 \Rrightarrow g_2$ and $g_2 \Rrightarrow g_1$ hold. Like Interval Temporal Logic [Mos00], we may define a number of operators on interval predicates, e.g., if $g \in IntvPred_{VA}$, $\Delta \in Intv$ and $s \in Stream_{VA}$:

$$(\Box g).\Delta.s \quad \mathrel{\widehat=} \quad \forall \Delta'{:}Intv \bullet \Delta' \subseteq \Delta \Rightarrow g.\Delta'.s \qquad\qquad (\ominus g).\Delta.s \quad \mathrel{\widehat=} \quad \exists \Delta' \bullet \Delta' \propto \Delta \wedge g.\Delta'.s$$

We define two operators on interval predicates: *chop*, which is used to formalise sequential composition, and *ω-iteration*, which is used to formalise a possibly infinite iteration (e.g., a while loop). The *chop* operator ';' is a basic operator on two interval predicates [Mos00, DDH12, DH12], where $(g_1 \,;\, g_2).\Delta$ holds iff either interval Δ may be split into two parts so that g_1 holds in the first and g_2 holds in the second, or the least upper bound of Δ is ∞ and g_1 holds in Δ. The latter disjunct allows g_1 to formalise an execution that does not terminate. Using chop, we define the possibly infinite iteration (denoted g^ω) of an interval predicate g as the greatest fixed point of $z = (g \,;\, z) \vee \mathsf{empty}$, where the interval predicates are ordered using '$\Rrightarrow$' (see [DHMS12] for details). Thus, we have:

$$
\begin{array}{rcl}
(g_1 \,;\, g_2).\Delta.s & \mathrel{\widehat=} & \left(\begin{array}{l}\exists \Delta_1, \Delta_2 \colon Intv \bullet (\Delta = \Delta_1 \cup \Delta_2) \wedge \\ \qquad (\Delta_1 \propto \Delta_2) \wedge g_1.\Delta_1.s \wedge g_2.\Delta_2.s\end{array}\right) \vee (\mathsf{inf} \wedge g_1).\Delta.s \\
g^{\omega} & \mathrel{\widehat=} & \nu z \bullet (g \,;\, z) \vee \mathsf{empty}
\end{array}
$$

In the definition of $g_1 \,;\, g_2$, interval Δ_1 may be empty, in which case $\Delta_2 = \Delta$, and similarly Δ_2 may empty, in which case $\Delta_1 = \Delta$. Hence, both $(\mathsf{empty} \,;\, g) \equiv g$ and $g \equiv (g \,;\, \mathsf{empty})$ trivially hold. An iteration g^{ω} of g may iterate g a finite (including zero) number of times, but also allows an infinite number of iterations [DHMS12].

Permissions and interference. To model true concurrency, the behaviour of the parallel composition between two processes in an interval Δ is modelled by the conjunction of the behaviours of both processes executing within Δ. Because this potentially allows conflicting accesses to shared variables, we incorporate fractional permissions into our framework [Boy03, DDH12]. We assume the existence of a *permission variable* in every state $\sigma \in State_{VA}$ of type $VA \to Proc \to [0,1]_{\mathbb{Q}}$, where $VA \subseteq Var \cup Addr$ and $\mathbb{Q}$ denotes the set of rationals. A process $p \in Proc$ has *write-permission* to location $va \in VA$ in $\sigma \in State_{VA}$ iff $\sigma.\Pi.va.p = 1$; has *read-permission* to va in σ iff $0 < \sigma.\Pi.va.p < 1$; and has *no-permission* to access va in σ iff $\sigma.\Pi.va.p = 0$.

We define $\mathscr{R}.va.p.\sigma \mathrel{\widehat=} (0 < \sigma.\Pi.va.p < 1)$ and $\mathscr{W}.va.p.\sigma \mathrel{\widehat=} (\sigma.\Pi.va.p = 1)$ and $\mathscr{D}.va.p.\sigma \mathrel{\widehat=} (\sigma.\Pi.va.p = 0)$ to be state predicates on permissions. In the context of a stream s, for any time $t \in \mathbb{Z}$, process p may only write to and read from va in the transition step from $s.(t-1)$ to $s.t$ if $\mathscr{W}.va.p.(s.t)$ and $\mathscr{R}.va.p.(s.t)$ hold, respectively. Thus, $\mathscr{W}.va.p.(s.t)$ does not give p permission to write to va in the transition from $s.t$ to $s.(t+1)$ (and similarly $\mathscr{R}.va.p$). For example, to state that process p updates variable v to value k at time t of stream s, the effect of the update should imply $((v = k) \wedge \mathscr{W}.v.p).(s.t)$.

One may introduce healthiness conditions on streams that formalise our assumptions on the underlying hardware. We assume that at most one process has write permission to a location va at any time, which is guaranteed by ensuring the sum of the permissions of the processes on va at all times is at most 1, i.e., $\forall s \colon Stream, t \colon Time \bullet ((\Sigma_{p \in Proc} \Pi.va.p) \leq 1).(s.t)$. Other conditions may be introduced to model further restrictions as required [DDH12].

4 Evaluating state predicates over intervals

The set of times within an interval corresponds to a set of states with respect to a given stream. Hence, if one assumes that expression evaluation is non-atomic (i.e., takes time), one must consider evaluation with respect to a set of states, as opposed to a single state. It turns out that there are a number of possible ways in which such an evaluation can take place, with varying degrees of non-determinism [HBDJ13]. In this paper, we consider *actual states evaluation*, which evaluates an expression with respect to the set of actual states that occur within an interval and *apparent states evaluation*, which considers the set of states apparent to a given process.

Actual states evaluation allow one to reason about the true state of a system, and evaluates an expression instantaneously at a single point in time. However, a process executing with fine-grained atomicity can only read a single variable at a time, and hence, will seldom be able to view an actual state because interference may occur between two successive reads. For example, a process p evaluating ecl_3 (the expression at cl_3) cannot read both $n1_p \mapsto mrk$ and $n1_p \mapsto val$ in a single atomic step, and hence, may obtain a value for ecl_3 that is different from any actual value of

ecl_3 because interference may occur between reads to $n1_p \mapsto mrk$ and $n1_p \mapsto val$. Therefore, we define an apparent states evaluator that models fine-grained expression evaluation over intervals. Our definition of apparent states evaluation does not fix the order in which $n1_p \mapsto mrk$ and $n1_p \mapsto val$ are read. We see this as advantageous over frameworks that must make the atomicity explicit (e.g., [VHHS06, CGLM06, DSW11]), which require an ordering to be chosen, even if an evaluation order is not specified by the corresponding implementation (e.g., [HHL+07]). In [VHHS06, CGLM06, DSW11], if the order of evaluation is modified, the linearisability proof must be redone, whereas our proof is more general because it shows that any order of evaluation is valid.

Evaluation over actual states. To formalise evaluators over actual states, for an interval Δ and stream $s \in Stream_{VA}$, we define $states.\Delta.s \mathrel{\widehat{=}} \{\sigma\colon State_{VA} \mid \exists t\colon \Delta \bullet \sigma = s.t\}$. Two useful operators for a sets of actual states of a state predicate c are $\diamondsuit c$ and $\boxdot c$, which specify that c holds in *some* and *all* actual state of the given stream within the given interval, respectively.

$$(\diamondsuit c).\Delta.s \mathrel{\widehat{=}} \exists \sigma\colon states.\Delta.s \bullet c.\sigma \qquad\qquad (\boxdot c).\Delta.s \mathrel{\widehat{=}} \forall \sigma\colon states.\Delta.s \bullet c.\sigma$$

Example 1. Suppose v is a variable, fa and fb are fields, and s is a stream such that the expression $(v \mapsto fa, v \mapsto fb)$ always evaluates to $(0,0)$, $(1,0)$ and $(1,1)$ within intervals $[1,4]_{\mathbb{N}}$, $[5,10]_{\mathbb{N}}$ and $[11,16]_{\mathbb{N}}$, respectively, i.e., for example $\boxdot((v \mapsto fa, v \mapsto fb) = (0,0)).[1,4]_{\mathbb{N}}.s$. Thus, both $\boxdot((v \mapsto fa) \geq (v \mapsto fb)).[1,16]_{\mathbb{N}}.s$ and $\diamondsuit((v \mapsto fa) > (v \mapsto fb)).[1,16]_{\mathbb{N}}.s$ may be deduced.

Using $\boxdot$, we define $\overleftarrow{c}$ and $\overrightarrow{c}$, which hold iff c holds at the beginning and end of the given interval, respectively.

$$\overleftarrow{c} \quad \widehat{=} \quad (\boxdot c \wedge \neg\mathsf{empty})\,;\, true \qquad\qquad \overrightarrow{c} \quad \widehat{=} \quad true\,;\, (\boxdot c \wedge \neg\mathsf{empty})$$

Operators $\boxdot$ and $\diamondsuit$ cannot accurately model fine-grained interleaving in which processes are able to access at most one location in a single atomic step. However, both $\boxdot$ and $\diamondsuit$ are useful for modelling the actual behaviour of the system as well as the behaviour of the coarse-grained abstractions that we develop. We may use $\boxdot$ to define *stability* of a variable v, and *invariance* of a state predicate c as follows:

$$stable.v \mathrel{\widehat{=}} \exists k \bullet \ominus(\overrightarrow{va = k}) \wedge \boxdot(va = k) \qquad\qquad inv.c \mathrel{\widehat{=}} \ominus\overrightarrow{c} \Rightarrow \boxdot c$$

Such definitions of stability and invariance are necessary because adjoining intervals are assumed to be disjoint, i.e., do not share a point of overlap. Therefore, one must refer to the values at the end of some immediately preceding interval.

Evaluation over states apparent to a process. Assuming the same setup as Example 1, if p is only able to access at most one location at a time, evaluating $(v \mapsto fa) < (v \mapsto fb)$ using the states *apparent* to process p over the interval $[1,16]_{\mathbb{N}}$ may result in *true*, e.g., if the value at $v{\cdot}fa$ is read within interval $[1,4]_{\mathbb{N}}$ and the value at $v{\cdot}fb$ read within $[11,16]_{\mathbb{N}}$.

Reasoning about the apparent states with respect to a process p using function *apparent* is not always adequate because it is not enough for an apparent state to exist; process p must also be able to read the relevant variables in this apparent state. Typically, it is not necessary for a process to be able to read all of the state variables to determine the apparent value of a given state predicate. In fact, in the presence of local variables (of other processes), it will be impossible for p to read the value of each variable. Hence, we define a function $apparent_{p,W}$, where $W \subseteq Var \cup Addr$ is the set of locations whose values process p needs to determine to evaluate the given state predicate.

$$apparent_{p,W}.\Delta.s \quad \widehat{=} \quad \{\sigma\colon State_W \mid \forall va\colon W \bullet \exists t\colon \Delta \bullet (\sigma.va = s.t.va) \wedge \mathscr{R}.va.p.(s.t)\}$$

Using this function, we are able to determine whether state predicates definitely and possibly hold with respect to the apparent states of a process. For a state predicate c, interval Δ, stream s and state σ, we let $accessed.c.\sigma$ denote the smallest set of locations (variables and addresses) that must be accessed in order to evaluate c in state σ and define $locs.c.\Delta.s \mathrel{\widehat{=}} \bigcup_{t\in\Delta} accessed.c.(s.t)$. For a process p, this is used to define $(\boxast_p\ c).\Delta.s$, which states that c holds in all states apparent to p in s within Δ. (Similarly $(\diamondast_p\ c).\Delta.s$.)

$$
\begin{array}{rcl}
(\boxast_p\ c).\Delta.s & \mathrel{\widehat{=}} & \mathsf{let}\ W = locs.c.\Delta.s\ \mathsf{in}\ \forall \sigma\colon apparent_{p,W}.\Delta.s \bullet c.\sigma \\
(\diamondast_p\ c).\Delta.s & \mathrel{\widehat{=}} & \mathsf{let}\ W = locs.c.\Delta.s\ \mathsf{in}\ \exists \sigma\colon apparent_{p,W}.\Delta.s \bullet c.\sigma
\end{array}
$$

Continuing Example 1, if $c \mathrel{\widehat{=}} ((v \mapsto fa) \geq (v \mapsto fb))$, we have $(\neg \boxast_p c).[1,16]_{\mathbb{N}}.s$ holds, i.e., $(\diamondast_p \neg c).[1,16]_{\mathbb{N}}.s$ even though $(\boxdot c).[1,16]_{\mathbb{N}}.s$ holds (cf. [DDH12, HBDJ13]). One may establish a number of properties on $\boxdot$, $\diamonddot$, $\boxast$ and $\diamondast$ [HBDJ13], for example $\diamondast_p(c \wedge d) \Rrightarrow \diamondast_p c \wedge \diamondast_p d$ holds. Furthermore, for any process p, variable v, field f and constant k,

$$
stable.v \wedge \diamondast_p((v \mapsto f) = k) \Rightarrow \diamonddot((v \mapsto f) = k) \tag{1}
$$

5 Behaviours and refinement

The *behaviour* of a command C executed by a non-empty set of processes P in a context $Z \subseteq Var$ is given by interval predicate $beh_{P,Z}.C$, which is defined inductively in Fig. 5. We use $beh_{p,Z}$ to denote $beh_{\{p\},Z}$ and assume the existence of a program counter variable pc_p for each process p. We define shorthand $\mathsf{fin\_Idle} \mathrel{\widehat{=}} \textsc{Enf}\,\mathsf{fin} \bullet \mathsf{Idle}$ and $\mathsf{inf\_Idle} \mathrel{\widehat{=}} \textsc{Enf}\,\mathsf{inf} \bullet \mathsf{Idle}$ to denote finite and infinite idling, respectively and use the interval predicates below to formalise the semantics of the commands in Fig. 5.

$$
\begin{array}{rcl}
\mathsf{eval}_{p,Z}.c & \mathrel{\widehat{=}} & \diamondast_p\, c \wedge beh_{p,Z}.\mathsf{Idle} \\
\mathsf{update}_{p,Z}(va,k) & \mathrel{\widehat{=}} & \begin{cases} beh_{p,Z\backslash\{va\}}.\mathsf{Idle} \wedge \neg\mathsf{empty} \wedge \boxdot(va = k \wedge \mathscr{W}_p.va) & \text{if } va \in Var \\ beh_{p,Z\backslash\{va\}}.\mathsf{Idle} \wedge \neg\mathsf{empty} \wedge \boxdot((*va) = k \wedge \mathscr{W}_p.va) & \text{if } va \in Addr \end{cases}
\end{array}
$$

To enable compositional reasoning, for interval predicates r and g, and command C, we introduce two additional constructs $\textsc{Rely}\, r \bullet C$ and $\textsc{Enf}\, g \bullet C$, which denote a command C with a *rely condition* r and an *enforced condition* g, respectively [DDH12].

We say that a concrete command C is a refinement of an abstract command A iff every possible behaviour of C is a possible behaviour of A. Command C may use additional variables to those in A, hence, we define refinement in terms of sets of variables corresponding to the contexts of A and C. In particular, we say A with context Y is *refined* by C with context Z with respect to a set of processes P (denoted $A \sqsubseteq_P^{Y,Z} C$) iff $beh_{P,Z}.C \Rrightarrow beh_{P,Y}.A$ holds. Thus, any behaviour of the concrete command C is a possible behaviour of the abstract command A. This is akin to operation refinement [RE96], however, our definition is with respect to the intervals over which the commands execute, as opposed to their pre/post states. We write $A \sqsubseteq_P^{Z} C$ for $A \sqsubseteq_P^{Z,Z} C$, write $A \sqsubseteq_P C$ for $A \sqsubseteq_P^{\varnothing} C$, and write $A \sqsubseteq_p^{Y,Z} C$ for $A \sqsubseteq_{\{p\}}^{Y,Z} C$.

The next lemma states that an assignment of state predicate c to a variable v may be decomposed to a guard $[c]$ followed by an assignment of *true* to v and a guard $[\neg c]$ followed by an assignment of *false* to v. Furthermore, one may move the frame of a command into the refinement relation.

$$
\begin{array}{rcl@{\qquad}rcl}
beh_{p,Z}.\mathsf{Idle} & \mathrel{\widehat=} & \forall va{:}Z \bullet \boxdot\neg\mathscr{W}.va.p & beh_{P,Z}.(C_1\,;\,C_2) & \mathrel{\widehat=} & beh_{P,Z}.C_1\,;\,beh_{P,Z}.C_2 \\
beh_{p,Z}.[c] & \mathrel{\widehat=} & \circledast_p\, c \wedge beh_{p,Z}.\mathsf{Idle} & beh_{P,Z}.(C_1 \sqcap C_2) & \mathrel{\widehat=} & beh_{P,Z}.C_1 \vee beh_{P,Z}.C_2 \\
beh_{p,Z}.\langle c\rangle & \mathrel{\widehat=} & \diamondsuit c \wedge beh_{p,Z}.\mathsf{Idle} & beh_{P,Z}.(\textsc{Rely}\, r \bullet C) & \mathrel{\widehat=} & r \Rightarrow beh_{P,Z}.C \\
beh_{P,Z}.C^{\omega} & \mathrel{\widehat=} & (beh_{P,Z}.C)^{\omega} & beh_{P,Z}.(\textsc{Enf}\, g \bullet C) & \mathrel{\widehat=} & g \wedge beh_{P,Z}.C \\
beh_{p,Z}.(l{:}\,C) & \mathrel{\widehat=} & \boxdot(pc_p = l) \wedge beh_{p,Z}.C & & &
\end{array}
$$

$$
beh_{p,Z}.(vae := e) \mathrel{\widehat=} \begin{cases} \exists k \bullet \mathsf{eval}_{p,Z}.(e = k)\,;\,\mathsf{update}_{p,Z}(v,k) & \text{if } vae \in Var \\ \exists k, a \bullet \mathsf{eval}_{p,Z}.(vae = a \wedge e = k)\,;\,\mathsf{update}_{p,Z}(a,k) & \text{otherwise} \end{cases}
$$

$$
beh_{P,Z}.(\|_{p:P}\ C_p) \mathrel{\widehat=}
$$

$$
\begin{cases}
true & \text{if } P = \varnothing \\
beh_{p,Z}.C_p & \text{if } P = \{p\} \\
\begin{array}{l}\exists P_1, P_2, S_1, S_2 \bullet (P_1 \cup P_2 = P) \wedge (P_1 \cap P_2 = \varnothing) \wedge P_1 \neq \varnothing \wedge P_2 \neq \varnothing \wedge \\
\qquad S_1 \in \{\mathsf{fin\_Idle}, \mathsf{inf\_Idle}\} \wedge S_2 \in \{\mathsf{fin\_Idle}, \mathsf{inf\_Idle}\} \wedge \\
\qquad (S_1 = \mathsf{inf\_Idle} \Rightarrow S_2 \neq \mathsf{inf\_Idle}) \wedge \\
\qquad beh_{P_1,Z}.((\|_{p:P_1}\ C_p)\,;\,S_1) \wedge beh_{P_2,Z}.((\|_{p:P_2}\ C_p)\,;\,S_2)\end{array} & \text{otherwise}
\end{cases}
$$

$$
beh_{P,Z}.[\![Y \mid C]\!] \mathrel{\widehat=} (Z \cap Y = \varnothing) \wedge beh_{P,Z\cup Y}.C
$$

Figure 5: Formalisation of behaviour function

Lemma 1 *Suppose c is a state predicate, $v \in Var$, $W, X \subseteq Var$, $Y, Z \subseteq Var \cup Addr$, $p \in Proc$, $P \subseteq Proc$ and A and C are commands. Then*
1. $v := c \ \sqsubseteq_p^{Z}\ ([c]\,;\, v := true) \sqcap ([\neg c]\,;\, v := false)$, *and*
2. $[\![W \mid A]\!] \ \sqsubseteq_P^{Y,Z}\ [\![X \mid C]\!]$ *provided* $A \sqsubseteq_P^{W\cup Y, X\cup Z} C$ *and* $W \subseteq (X \cup Z)$ *and* $W \cap Y = \varnothing = X \cap Z$.

The next theorem establishes a Galois connection between rely and enforced conditions [DDH12].

Theorem 1 $(\textsc{Rely}\, r \bullet A) \sqsubseteq_P^{Y,Z} C \ \Leftrightarrow\ A \sqsubseteq_P^{Y,Z} (\textsc{Enf}\, r \bullet C)$

When modelling a lock-free algorithm [CGLM06, DSW11, VHHS06], one assumes that each process repeatedly executes operations of the data structure, and hence the processes of the system only differ in terms of the process ids. For such programs, a proof of the parallel composition may be decomposed using the following theorem [DD12].

Theorem 2 *If $p \in Proc$, $Y, Z \subseteq Var \cup Addr$, and $A(p)$ and $C(p)$ are commands parameterised by p, then $(\textsc{Rely}\, g \bullet \|_{p:P} A(p)) \sqsubseteq_P^{Y,Z} (\|_{p:P} C(p))$ holds if for some interval predicate r and some $p \in P$ and $Q \mathrel{\widehat=} P\backslash\{p\}$ both of the following hold.*

$$
\textsc{Rely}\, g \wedge r \bullet A(p) \ \sqsubseteq_p^{Y,Z}\ C(p) \tag{2}
$$

$$
g \wedge beh_{Q,Z}.(\|_{q:Q} C(q)) \ \Rrightarrow\ r \tag{3}
$$

6 Verification of the lazy set

Details of the proof are presented in [DD13]. Here, we only present a high-level overview of the proof and its decomposition (see Section 7). Furthermore, because (as already mentioned) verification of linearisability of `contains` is known to be difficult using frameworks that only

$$
\begin{array}{rcl}
\varphi^{k+1}.ua.\sigma & \mathrel{\widehat{=}} & \mathsf{if}(k=0)\,\mathsf{then}\,ua\,\mathsf{else}\ eval.((\varphi^{k}.ua.\sigma)\mapsto nxt).\sigma \\
\mathsf{RE}.ua.vb.\sigma & \mathrel{\widehat{=}} & \exists k{:}\mathbb{N}\bullet \varphi^{k}.ua.\sigma = vb \\
setAddr.\sigma & \mathrel{\widehat{=}} & \{a{:}Addr \mid \mathsf{RE}.Head.a.\sigma \wedge \neg eval.(a\mapsto mrk).\sigma\} \\
absSet.\sigma & \mathrel{\widehat{=}} & \{v{:}\,Val \mid \exists a{:}\,setAddr.\sigma \bullet v = eval.(a\mapsto val).\sigma\} \\
\mathsf{CGCon}(p,x) & \mathrel{\widehat{=}} & (\langle x\in absSet\rangle\,;\ res_p := true)\sqcap(\langle x\notin absSet\rangle\,;\ res_p := false) \\
\mathsf{CGS}(p) & \mathrel{\widehat{=}} & [\![res_p \mid (\textstyle\bigsqcap_{x:\mathbb{Z}}(\mathsf{CGAdd}(p,x)\sqcap\mathsf{CGRem}(p,x)\sqcap\mathsf{CGCon}(p,x)))^{\omega}]\!] \\
\mathsf{CGSet}(P) & \mathrel{\widehat{=}} & [\![Head, Tail \mid \textsc{Rely}\,\overleftarrow{HTInit}\bullet \|_{p:P}\ \mathsf{CGS}(p)]\!]
\end{array}
$$

Figure 6: A coarse-grained abstraction of `contains`

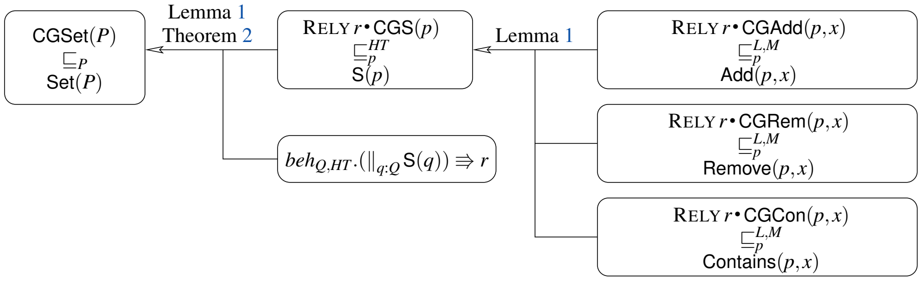

Figure 7: Proof decomposition for the lazy set verification

consider the pre/post states [CGLM06, DSW11, Vaf10, VHHS06], we focus on its proof. A coarse-grained abstraction of $\mathsf{Set}(P)$ in Fig. 4 is given by $\mathsf{CGSet}(P)$ in Fig. 6, where for example, $\mathsf{Contains}$ is replaced by CGCon, which tests to see if x is in the set using an atomic (coarse-grained) guard, then updates the return value to *true* or *false* depending on the outcome of the test. Details of CGAdd and CGRem are elided; we ask the interested reader to consult [DD13].

To prove refinement for $\mathsf{Contains}(p,x)$ in Fig. 7, we use Lemma 1 to replace $\mathsf{Contains}(p,x)$ by

$$CL\,;\ ((clt_3{:}\,([IN]\,;\ res_p := true))\sqcap(clf_3{:}\,([\neg IN]\,;\ res_p := false)))$$

where label cl_3 has been split into clt_3 and clf_3 for the true and false cases, respectively, and

$$IN \mathrel{\widehat{=}} \neg(n1_p \mapsto mrk)\wedge((n1_p\mapsto val)=x) \qquad CL \mathrel{\widehat{=}} cl_1{:}\,(n1_p := Head)\,;\ cl_2{:}\,\mathsf{CLoop}(p,x)$$

We then distribute CL within the '$\sqcap$', use monotonicity to match the abstract and concrete *true* and *false* branches, then use monotonicity again to remove the assignments to res_p from both sides of the refinement. Thus, we are required to prove the following properties.

$$\textsc{Rely}\, r\bullet\langle x\in absSet\rangle \quad \sqsubseteq_P^{L,M} \quad CL\,;\ clt_3{:}\,[IN] \tag{4}$$

$$\textsc{Rely}\, r\bullet\langle x\notin absSet\rangle \quad \sqsubseteq_P^{L,M} \quad CL\,;\ clf_3{:}\,[\neg IN] \tag{5}$$

Proof of (4). This condition states that there must be an actual state σ within the interval in which $CL\,;\ clt_3{:}\,[IN]$ executes, such that $x\in absSet.\sigma$ holds, i.e., there is a point at which the abstract set contains x. It may be the case that a process $q\neq p$ has removed x from the set by the time process p returns from the contains operation. In fact, x may be added and removed several times by concurrent add and remove operations before process p completes execution of $\mathsf{Contains}(p,x)$. However, this does not affect linearisability of $\mathsf{Contains}(p,x)$ because a state for which $x \in absSet$ holds has been found. An execution of $\mathsf{Contains}(p,x)$ that returns *true* would only be incorrect (not linearisable) if *true* is returned and $\boxdot(x \notin absSet)$ holds for the interval in which $CL\,;\, clt_3\!:\![IN]$ executes. Similarly, we prove correctness of (5) by showing that is impossible for there to be an execution that returns *false* if $\boxdot(x \in absSet)$ holds in the interval of execution.

Proof of (5). Using Theorem 1, we transfer the rely condition r to the right hand side as an enforced property, define $INV \mathrel{\widehat{=}} \mathsf{RE}.Head.n1_p \vee (n1_p \mapsto mrk)$, and require that r implies:

$$inv.INV \quad \wedge \quad \Box(\boxdot(pc_p = cl_3) \Rightarrow inv.(n1_p \mapsto mrk) \wedge \forall k\!:\!Val \bullet inv.((n1_p \mapsto val) = k)) \tag{6}$$

The behaviour of the right hand side of (4) simplifies to the following interval predicate using assumption (6) and that r is assumed to split.

$$(r \wedge beh_{p,L}.\mathsf{Idle})\,;\,(r \wedge (\boxdot INV\,;\,(\diamondsuit\neg(n1_p \mapsto mrk) \wedge \diamondsuit((n1_p \mapsto val) = x))))$$

Using assumption (6), it is possible to show that the second part of the chop implies the following, where $inSet(ua,x) \mathrel{\widehat{=}} \mathsf{RE}.Head.ua \wedge \neg(ua \mapsto mrk) \wedge (ua \mapsto val = x)$ holds iff ua with value x is in the abstract set.

$$\exists a\!:\!Addr \bullet \overrightarrow{inSet(Head,a,x)}\,;\,(\boxdot(n1_p = a) \wedge \diamondsuit\neg(a \mapsto mrk) \wedge \diamondsuit((a \mapsto val) = x))$$

This trivially implies the required result, i.e., that $\diamondsuit(x \in absSet)$.

To prove (5), as with (4), we use Theorem 1 to transfer the rely condition r to the right hand side as an enforced property. By logic, the right hand side of (5) is equivalent to command $\mathrm{ENF}\, r \wedge (\boxdot(x \in absSet) \vee \diamondsuit(x \notin absSet)) \bullet CL\,;\, clf_3\!:\![\neg IN]$. The $\diamondsuit(x \notin absSet)$ case is trivially true. For case $\boxdot(x \in absSet)$, we require that r satisfies:

$$\Box(\boxdot(x \in absSet) \Rightarrow \exists a\!:\!Addr \bullet \boxdot inSet(Head,a,x)) \tag{7}$$

$$\Box(\forall k\!:\!\mathbb{N} \bullet \varphi^k.Head \neq Tail \Rightarrow (\varphi^k.Head \mapsto val) < (\varphi^{k+1}.Head \mapsto val)) \tag{8}$$

$$\boxdot(\mathsf{RE}.n1_p.Tail) \tag{9}$$

By (7), in any interval, if the value x is in the set throughout the interval, there is an address that can be reached from *Head*, the marked bit corresponding to the node at this address is unmarked and the value field contains x. By (8) the reachable nodes of the list (including marked nodes) must be sorted in strictly ascending order and by (9) the *Tail* node must be reachable from $n1_p$. Conditions (7), (8) and (9) together imply that there cannot be a terminating execution of $\mathsf{CLoop}(p,x)$ such that $clf_3\!:\![\neg IN]$ holds, i.e., the behaviour is equivalent to *false*.

The rely condition r for the proof of `contains` must imply each of (6), (7), (8) and (9). We choose to take the weakest possible instantiation and let r be the conjunction (6) $\wedge$ (7) $\wedge$ (8) $\wedge$ (9), which, as shown in Fig. 7, must be satisfied by the rest of the program. This proof is straightforward by expanding the definitions of the behaviours and its details are elided.

7 Conclusions

We have developed a framework, based on [DDH12], for reasoning about the behaviour of a command over an interval that enables reasoning about pointer-based programs where processes may refer to states that are apparent to a process [HBDJ13]. Parallel composition is defined using conjunction and conflicting access to shared state is disallowed using fractional permissions, which models truly concurrent behaviour. We formalise behaviour refinement in our framework, which can be used to show that a fine-grained implementation is a refinement of a coarse-grained abstraction. One is only required to identify linearising statements of the abstraction (as opposed to the implementation) and the proof of linearisability itself is simplified due to the coarse-granularity of commands. For the coarse-grained contains operation in Fig. 6, the guard $\langle x \in absSet \rangle$ is the linearising statement for an execution that returns *true* and $\langle x \notin absSet \rangle$ the linearising statement of an execution that returns *false*.

Our proof method is compositional (in the sense of rely/guarantee) and in addition, we develop the rely conditions necessary to prove correctness incrementally. As an example, we have shown refinement between the `contains` operation of the lazy set [HHL+07] and an abstraction of the contains operation that executes with coarse-grained atomicity.

Acknowledgements. This work is supported by EPSRC Grant EP/J003727/1. We thank Gerhard Schellhorn and Bogdan Tofan for useful discussions, and anonymous reviewers for their insightful comments.

Bibliography

[Boy03] J. Boyland. Checking Interference with Fractional Permissions. In Cousot (ed.), *SAS*. LNCS 2694, pp. 55–72. Springer, 2003.

[BSTR11] S. Bäumler, G. Schellhorn, B. Tofan, W. Reif. Proving linearizability with temporal logic. *Formal Asp. Comput.* 23(1):91–112, 2011.

[CGLM06] R. Colvin, L. Groves, V. Luchangco, M. Moir. Formal Verification of a Lazy Concurrent List-Based Set Algorithm. In Ball and Jones (eds.), *CAV*. LNCS 4144, pp. 475–488. Springer, 2006.

[DD12] B. Dongol, J. Derrick. Proving linearisability via coarse-grained abstraction. *CoRR* abs/1212.5116, 2012.

[DD13] B. Dongol, J. Derrick. Simplifying proofs of linearisability using layers of abstraction. *CoRR* abs/1307.6958, 2013.

[DDH12] B. Dongol, J. Derrick, I. J. Hayes. Fractional Permissions and Non-Deterministic Evaluators in Interval Temporal Logic. *ECEASST* 53, 2012.

[DGLM04] S. Doherty, L. Groves, V. Luchangco, M. Moir. Formal Verification of a Practical Lock-Free Queue Algorithm. In Frutos-Escrig and Núñez (eds.), *FORTE*. LNCS 3235, pp. 97–114. Springer, 2004.

[DH12] B. Dongol, I. J. Hayes. Deriving Real-Time Action Systems Controllers from Multiscale System Specifications. In Gibbons and Nogueira (eds.), *MPC*. LNCS 7342, pp. 102–131. Springer, 2012.

[DHMS12] B. Dongol, I. J. Hayes, L. Meinicke, K. Solin. Towards an Algebra for Real-Time Programs. In Kahl and Griffin (eds.), *RAMiCS*. LNCS 7560, pp. 50–65. 2012.

[DSW11] J. Derrick, G. Schellhorn, H. Wehrheim. Verifying Linearisability with Potential Linearisation Points. In Butler and Schulte (eds.), *FM*. LNCS 6664, pp. 323–337. Springer, 2011.

[EQS+10] T. Elmas, S. Qadeer, A. Sezgin, O. Subasi, S. Tasiran. Simplifying Linearizability Proofs with Reduction and Abstraction. In Esparza and Majumdar (eds.), *TACAS*. LNCS 6015, pp. 296–311. Springer, 2010.

[Gro08] L. Groves. Verifying Michael and Scott's Lock-Free Queue Algorithm using Trace Reduction. In Harland and Manyem (eds.), *CATS*. CRPIT 77, pp. 133–142. 2008.

[HBDJ13] I. J. Hayes, A. Burns, B. Dongol, C. B. Jones. Comparing Degrees of Non-Determinism in Expression Evaluation. *Comput. J.* 56(6):741–755, 2013.

[HHL+07] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. S. III, N. Shavit. A Lazy Concurrent List-Based Set Algorithm. *Parallel Processing Letters* 17(4):411–424, 2007.

[HW90] M. P. Herlihy, J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* 12(3):463–492, 1990.

[Jon83] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Prog. Lang. and Syst.* 5(4):596–619, 1983.

[Lip75] R. J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM* 18(12):717–721, 1975.

[Mos00] B. C. Moszkowski. A Complete Axiomatization of Interval Temporal Logic with Infinite Time. In *LICS*. Pp. 241–252. 2000.

[ORV+10] P. W. O'Hearn, N. Rinetzky, M. T. Vechev, E. Yahav, G. Yorsh. Verifying linearizability with hindsight. In Richa and Guerraoui (eds.), *PODC*. Pp. 85–94. ACM, 2010.

[RE96] W. P. de Roever, K. Engelhardt. *Data Refinement: Model-oriented proof methods and their comparison*. Cambridge Tracts in Theor. Comp. Sci. 47. Cambridge University Press, 1996.

[TW11] A. J. Turon, M. Wand. A separation logic for refining concurrent objects. In Ball and Sagiv (eds.), *POPL*. Pp. 247–258. ACM, 2011.

[Vaf07] V. Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2007.

[Vaf10] V. Vafeiadis. Automatically Proving Linearizability. In Touili et al. (eds.), *CAV*. LNCS 6174, pp. 450–464. Springer, 2010.

[VHHS06] V. Vafeiadis, M. Herlihy, T. Hoare, M. Shapiro. Proving correctness of highly-concurrent linearisable objects. In Torrellas and Chatterjee (eds.), *PPOPP*. Pp. 129–136. 2006.