Data Refinement with Low-level Pointer Operations

Ivana Mijajlović[1] and Hongseok Yang[2⋆]

[1] Queen Mary, University of London, UK
[2] ERC-ACI, Seoul National University, South Korea

Abstract. We present a method for proving data refinement in the presence of low-level pointer operations, such as memory allocation and deallocation, and pointer arithmetic. Surprisingly, none of the existing methods for data refinement, including those specifically designed for pointers, are sound in the presence of low-level pointer operations. The reason is that the low-level pointer operations allow an additional potential for obtaining the information about the implementation details of the module: using memory allocation and pointer comparison, a client of a module can find out which cells are internally used by the module, even without dereferencing any pointers. The unsoundness of the existing methods comes from the failure of handling this potential. In the paper, we propose a novel method for proving data refinement, called power simulation, and show that power simulation is sound even with low-level pointer operations. Then, we identify special cases where power simulation has a more conventional-style representation. It turns out that the obtained representation for those special cases is an interesting combination of two well-known simulation methods, namely forward and backward simulation.

1 Introduction

Data refinement [7] is a process in which the concrete representation of some abstract module is formally derived. Viewed from outside, the more concrete representation behaves the same as (or better than) the given abstract module. Thus, data refinement ensures that for every program, we can replace a given abstract module by the concrete one, while preserving (or even improving) the observable behavior of the program.

Our aim here is to develop a method of data refinement in the presence of *low-level* pointer operations, such as memory allocation and deallocation, and pointer arithmetic. Developing such methods is challenging, because low-level pointer operations allow subtle ways for accessing the internals of a module; without protecting the module internals from these accessing mechanisms (and thus ensuring that the module internals are only accessed by the module operations), we cannot have a sound method of data refinement.

⋆ Yang was supported by R08-2003-000-10370-0 from the Basic Research Program of Korea Science & Engineering Foundation, and Brain Korea 21 project in 2005.

```
module counter1 {
  init() {∗1=allocCell2(); ∗∗1=0;}
  inc() {∗∗1=(∗∗1)+1;}
  read() {∗3=(∗∗1);}
  final() {free(∗1); ∗1=0;}  }
```

```
module counter2 {
  init() {∗1=0;}
  inc() {∗1=(∗1)+1;}
  read() {∗3=(∗1);}
  final() {∗1=0;}  }
```

```
module counter3 {
  init() {∗1=alloc(); ∗∗1=0;}
  inc() {∗∗1=(∗∗1)+1;}
  read() {∗3=(∗∗1);}
  final() {free(∗1); ∗1=0;}  }
```

Fig. 1. Counter Modules

The best-known accessing mechanism is the dereference of cross-boundary pointers. If a client program knows the location of some internal heap cell of a module, which we call a *cross-boundary pointer*, it can directly read or write that internal cell by dereferencing that location. Thus, such a program can detect the changes in the representation of a module, and invalidate the standard methods of data refinements. This problem of cross-boundary pointers is well known, and several methods for data refinement have been proposed specifically to solve this problem [12, 16, 1, 3, 2].

However, none of the existing data-refinement methods, including the ones designed for cross-boundary pointers, can handle another accessing mechanism, which we call *allocation-status testing.* This mechanism uses the memory allocator and pointer comparison (with specific integers) to find out which cells are used internally by a module. A representative example check2 that implements this mechanism is z=alloc(); if (z==2) then v=1 else v=2. Assume that the memory allocator alloc nondeterministically chooses one inactive cell, and allocates the chosen cell. Under this assumption, check2 can detect whether cell 2 is used internally by a module or not. If a module is currently using cell 2, the newly allocated cell in check2 has to be different from 2, so that check2 always assigns 2 to v. On the other hand, if a module is not using cell 2, so cell 2 is free, then the memory allocation in check2 may or may not choose cell 2, and so, the variable v nondeterministically has value 1 or 2. Thus, by changing its nondeterministic behavior, check2 "observes" the allocation status of cell 2.

Protecting the module internals from the allocation-status testing is crucial for sound data refinement; using the allocation-status testing, a client can detect space-optimizing data refinements. We explain the issue with the first two counter modules, counter1 and counter2, in Fig. 1. Both modules implement a counter "object" with operations for incrementing the counter (inc) or reading the value of the counter (read). The main difference is that the second module uses less space than the first module. Let allocCell2() be a memory allocator that always selects cell 2: if cell 2 is inactive, allocCell2() allocates the cell; otherwise, i.e., if 2 is already allocated, then allocCell2() diverges. The first module is initialized by allocating cell 2 (allocCell2()) and storing the value of the counter in the allocated cell 2. The address of this newly allocated cell, namely 2, is kept in cell 1. On the other hand, the second module uses only cell 1, and stores the counter value directly to cell 1. The space-saving optimization in counter2 can be detected by the command check2 in the previous paragraph. When check2

is run with counter1, it always assigns 1 to v, but when check2 is run with the other module counter2, it can nondeterministically assign 1 or 2 to v. Thus, the optimization in counter2 is not correct, because it generates a new behavior of the client program check2.

Here, we present a data-refinement method that handles both cross-boundary pointers and allocation-status testing. Our method is based on Mijajlović *et al.*'s technique [12], which ensures correct data refinement in the presence of cross-boundary pointers, but as stressed there, not with allocation status testing. We provide a more general method which can cope well with both problems. The key idea of our method is to restrict the space optimization of a concrete module to nondeterministically allocated cells only, in order to hide the identities of the optimized cells from a client program, by making all the allocation-status testing fail to give any useful information. For instance, our method allows counter3 in Fig. 1 to be optimized by counter2, because the internal cell in counter3 is allocated nondeterministically. Note that even with check2, a client cannot detect this optimization, e.g. when cell 2 is free initially, counter3.init(); detect2 nondeterministically assigns 1 or 2 to v, just as counter2.init(); detect2 does. The precise formulation of our method uses a new notion of simulation – *power simulation*, to express this restriction on space optimization.

Related Work and Motivation It has long been known that pointers cause great difficulties in the treatment of data abstraction [8, 9], and this has lead on to a non-trivial body of research [1, 3, 13, 11, 18, 16]. The focus of the present work (and [12]), on problems caused by low-level operations, sets it apart from all this other research.

Now, the reader might think that these problems arise only because of language bugs. Indeed, previous work has relied strongly on protection mechanisms of high-level, garbage collected languages. In such high-level languages, the nondeterministic memory allocation is harmless; it does not let one implement the allocation-status testing (because those languages forbid explicit deallocation and pointer arithmetic) and the nondeterministic allocation can even be treated deterministically using location renaming [18, 16]. Moreover, those high-level languages often have sophisticated type systems [3, 2] that limit cross-boundary pointers. However, we would counter that a comprehensive approach to abstraction cannot be based on linguistic restrictions. For, the fact of the existence of significant suites of infrastructure code – operating systems, database servers, network servers – argues against it. The architecture of this code is not enforced by linguistic mechanisms, and it is hard to see how it could be. Low-level code naturally uses cross-boundary pointers and address arithmetic. But it is a mistake to think that infrastructure code is unstructured; it often exhibits a large degree of pre-formal modularity. In this paper, we will demonstrate that there is no inherent reason why the *idea* of refinement of modules should not be applicable to it.

Outline We start the paper by defining the storage model and the programming language in Sec. 2 and 3. Then, in Sec. 4, we describe the problem of finding a sound data-refinement method, and show that the usual forward method of data refinement fails to be a solution for the problem. In Sec. 5, we introduce the notion of *power simulation*, and prove its soundness; so, power simulation is a solution for the problem. In Sec. 6, we study the special case of power simulation that can be expressed in a more conventional style. Finally, in Sec. 7, we conclude the paper.

2 Storage Model and Finite Local Action

Our storage model, St, is the RAM model in separation logic [17, 10]:

$$\mathsf{Loc} = \{1, 2, \ldots\} \qquad \mathsf{Int} = \{\ldots, -2, -1, 0, 1, \ldots\} \qquad \mathsf{St} = \mathsf{Loc} \rightharpoonup_{fin} \mathsf{Int}$$

A state $h \in \mathsf{St}$ in the model is a finite mapping from locations to integer values; the domain of h denotes the set of currently allocated memory cells, and the "action" of h the contents of those allocated cells. Note that addresses are positive natural numbers, and so, they can be manipulated by arithmetic operations. We recall the disjointness predicate $h\#h'$ and the (partial) heap combining operator $h{\cdot}h'$ from separation logic. The predicate $h\#h'$ means that $\mathsf{dom}(h) \cap \mathsf{dom}(h') \neq \emptyset$; and, $h{\cdot}h'$ is defined only for such disjoint heaps h and h', and in that case, it denotes the combined heap $h \cup h'$. We overload the disjointness predicate $\#$, and for states h and location sets L, we write $h\#L$ to mean that all locations in L are free in h (i.e., $\mathsf{dom}(h) \cap L = \emptyset$).

We specify a property of storage, using subsets of St directly, instead of syntactic formulas. We call such subsets of St *predicates*, and use semantic versions of separating conjunction $*$ and preciseness from separation logic:

$$p, q \in \mathsf{Pred} \stackrel{def}{=} \wp(\mathsf{St}) \qquad p * q \stackrel{def}{=} \{h_p{\cdot}h_q \mid h_p \in p \wedge h_q \in q\} \qquad \mathsf{true} \stackrel{def}{=} \mathsf{St}$$

p is precise $\stackrel{def}{\Leftrightarrow}$ for all h, there is at most one splitting $h_p{\cdot}h_0 = h$ of h s.t. $h_p \in p$.

An *action* r is a relation from St to $\mathsf{St} \cup \{\mathsf{av}, \mathsf{flt}\}$. Intuitively, it denotes a nondeterministic client program that uses a module. Action r can output two types of errors, access violation av and memory fault flt. The first error av means that a client attempts to break the boundary between the client and the module, by accessing the internals of the module directly without using module operations. The second one, flt, means that a client tries to dereference a null or a dangling pointer. Note that if $\neg h[r]\mathsf{flt}$, state h contains all the cells that r dereferences, except the newly allocated cells. As in separation logic, we write $\mathsf{safe}(r, h)$ to indicate this (i.e., $\neg h[r]\mathsf{flt}$).

A *finite local action* is an action that satisfies: *safety monotonicity*, *frame property*, *finite access property*, and *contents independence*. Intuitively, these four properties mean that each execution of the action accesses only finitely many heap cells. Some of the cells are accessed directly by pointer dereferencing, so that the contents of the cells affects the execution, while the other remaining

cells are accessed only indirectly by the allocation-status testing, so that the execution only depends on the allocation status of the cells, not their contents. More precisely, we define the four properties as follows:[1]

- **Safety Monotonicity**: if $h_0 \# h_1$ and $\mathsf{safe}(r, h_0)$, then $\mathsf{safe}(r, h_0 \cdot h_1)$.
- **Frame Property**: if $\mathsf{safe}(r, h_0)$ and $h_0 \cdot h_1[r]h'$, then $\exists h'_0.\ h' = h'_0 \cdot h_1 \wedge h_0[r]h'_0$.
- **Finite Access Property**: if $\mathsf{safe}(r, h_0)$ and $h_0[r]h'_0$, then

$$\exists L \subseteq_{fin} \mathsf{Loc}.\, \forall h_1.\, (h_1 \# h_0 \ \wedge\ h_1 \# h'_0 \ \wedge\ (\mathsf{dom}(h_1) \cap L = \emptyset)) \ \Rightarrow\ h_0 \cdot h_1[r]h'_0 \cdot h_1.$$

- **Contents Independence**: if $\mathsf{safe}(r, h_0)$ and $h_0 \cdot h_1[r]h'_0 \cdot h_1$, then $h_0 \cdot h_2[r]h'_0 \cdot h_2$ for all states h_2 with $\mathsf{dom}(h_1) = \mathsf{dom}(h_2)$.

The first two properties are well-known locality properties from separation logic, and mean that if h_0 contains all the directly accessed cells by a "command" r, every computation from a bigger state $h_0 \cdot h_1$ is safe, and it can be tracked by some computation from the smaller state h_0. The third condition expresses the converse; every computation from the smaller state h_0 can be extended to a computation from the bigger state $h_0 \cdot h_1$, as long as the extended part h_1 does not include directly accessed locations ($h_1 \# h_0 \wedge h_1 \# h'_0$) or indirectly accessed locations (i.e., $\mathsf{dom}(h_1) \cap L = \emptyset$). Note that the finite set L contains all the indirectly accessed locations by the computation $h_0[r]h'_0$. The last one, contents independence, expresses that if $\mathsf{safe}(r, h_0)$, the execution of r from a bigger state $h_0 \cdot h_1$ does not look at the contents of cells in h_1; it can only use the information that the locations in h_1 are allocated initially. At first glance, it may seem that contents independence follows from the frame property, but the following example suggests otherwise. Let $[]$ be the empty state, and let r be an action defined by $h[r]v \Leftrightarrow h = v \ \wedge\ (h = [] \ \vee\ (1 \in \mathsf{dom}(h) \wedge h(1) = 2))$. This "command" r satisfies both the safety monotonicity and the frame property, but not the contents independence; even though $\mathsf{safe}(r, [])$ and $1 \notin \mathsf{dom}([])$, "command" r behaves differently depending on the contents of cell 1. The finite access property and contents independence are new in this paper, and they play an important role in the soundness of our data-refinement method (Sect. 5.1).

Definition 1 (Finite Local Action). *A* finite local action*, in short* FLA*, is an action that satisfies safety monotonicity, frame property, finite access property, and contents independence. A finite local action is* av-free *iff it does not relate any state to* av.

The set of finite local actions has a structure rich enough to interpret programs with all the low-level pointer operations that have been considered in separation logic.[2] Let $\mathcal{F}$ be the poset of FLAs ordered by the "graph-subset"

[1] All the states free in the properties are universally quantified.

[2] Thus, the set of finite local actions, as a semantic domain, expresses the computational behavior of pointer programs more accurately than the set of local actions, just as the set of continuous functions is a more "accurate" semantic domain than that of monotone functions in the domain theory.

Let l be a location, i an integer, n a positive natural number, and I a set of integers.

$h[\mathsf{update}(l,i)]v \overset{def}{\Leftrightarrow}$ if $l \notin \mathsf{dom}(h)$ then $v{=}\mathsf{flt}$ else $v{=}h[l \mapsto i]$

$h[\mathsf{cons}(l,n)]v \overset{def}{\Leftrightarrow}$ if $l \notin \mathsf{dom}(h)$ then $v{=}\mathsf{flt}$ else $(\exists l'.\ v{=}(h[l \mapsto l'])\cdot[l'{\to}0,..,l'{+}n{-}1{\to}0])$

$h[\mathsf{dispose}(l)]v \overset{def}{\Leftrightarrow}$ if $l \notin \mathsf{dom}(h)$ then $v{=}\mathsf{flt}$ else $v\cdot[l{\to}h(l)]{=}h$

$h[\mathsf{test}(l,I)]v \overset{def}{\Leftrightarrow}$ if $l \notin \mathsf{dom}(h)$ then $v{=}\mathsf{flt}$ else $(v{=}h \wedge h(l){\in}I)$

Fig. 2. Semantic Low-level Pointer Operations

relation $\sqsubseteq$[3], and let $\mathcal{F}_{noav}$ be the sub-poset of $\mathcal{F}$ consisting of av-free FLAs. Particularly interesting are the low-level pointer operations, such as the memory update, allocation and deallocation of a cell, and a test "${*}l \in I$" for location l and integer set I: if l is allocated and it contains a value in I, the test skips; if l is allocated but its value is not in I, the test blocks; otherwise (i.e., if l is not allocated), the test generates the memory fault flt. For instance, $\mathsf{test}(1,\{3\})$ expresses the conditional statement $\mathsf{if}\ ({*}1{\neq}3)\{\mathsf{diverge}\}$. Note that $\mathsf{test}(1,\{3\})$ generates flt precisely when the boolean condition ${*}1{\neq}3$ dereferences an inactive cell.

Lemma 1. *The poset $\mathcal{F}_{noav}$ of av-free FLAs contains the operations in Fig. 2.*

Proof. None of the defined operations generate av. So, it suffices to show that the operations obey all four properties of finite local actions. We only prove that $\mathsf{update}(l,i)$ satisfies the finite access property; it is straightforward to prove the remaining cases. Suppose that $\neg h[\mathsf{update}(l,i)]\mathsf{flt}$ and $h[\mathsf{update}(l,i)]h'$. Then, l is allocated in both h and h', and $\mathsf{dom}(h) = \mathsf{dom}(h')$. Thus, for all states h_1 such that $h_1\#h$ and $h_1\#h$, location l is allocated in both $h{\cdot}h_1$ and $h'{\cdot}h_1$ and its value in $h'{\cdot}h_1$ is $h'(l) = i$. Hence, $h{\cdot}h_1[\mathsf{update}(l,i)]h'{\cdot}h_1$. Thus, the required set L of indirectly accessed locations in the finite access property is the empty set. □

Lemma 2. *Both $\mathcal{F}$ and $\mathcal{F}_{noav}$ are complete lattices that have the set union as their join operator: for every family $\{r_i\}_{i\in I}$ in each poset, $\bigsqcup_{i\in I} r_i$ is $\bigcup_{i\in I} r_i$.*

Proof. Since both $\mathcal{F}$ and $\mathcal{F}_{noav}$ are ordered by the graph-subset relation, we only need to show that they are closed under arbitrary union. Note that for every family $\{r_i\}_{i\in I}$ in $\mathcal{F}$, if each r_i is av-free, then $\bigcup_{i\in I} r_i$ is av-free as well. Thus, it suffices to show the closedness only for $\mathcal{F}$. It is well-known that the set of local actions are closed under union [19]. Thus, we focus on the finite access property and contents independence. Let $\{r_i\}_{i\in I}$ be a family of FLAs, and let r be its graph union $\bigcup_{i\in I} r_i$. We first show that r satisfies the finite access property. Suppose that $\neg h_0[r]\mathsf{flt}$ and $h_0[r]h_0'$. By the definition of r, there is some r_j such that $\neg h_0[r_j]\mathsf{flt}$ and $h_0[r_j]h_0'$. Since r_j satisfies the finite access property, there exists a finite set L of indirectly accessed locations for $h_0[r_j]h_0'$. We claim that L is the required set. To see why, consider a state h_1 such that $h_1\#h_0$, $h_1\#h_0'$ and

[3] $r \sqsubseteq r'$ iff $\forall h \in \mathsf{St}.\forall v \in \mathsf{St} \cup \{\mathsf{flt},\mathsf{av}\}.\ h[r]v \Rightarrow h[r']v$

$\mathsf{dom}(h_1) \cap L = \emptyset$. By the finite access property of r_j, we have that $h_0 \cdot h_1[r_j]h'_0 \cdot h_1$. Since r includes r_j, we also have $h_0 \cdot h_1[r]h'_0 \cdot h_1$.

For the contents independence, consider states h_0, h'_0, h_1, h_2 such that

$$h_1 \# h_0 \ \wedge \ h_1 \# h'_0 \ \wedge \ \neg h_0[r]\mathsf{flt} \ \wedge \ h_0 \cdot h_1[r]h'_0 \cdot h_1 \ \wedge \ \mathsf{dom}(h_1) = \mathsf{dom}(h_2).$$

Then, there exists r_j such that $\neg h_0[r_j]\mathsf{flt}$ and $h_0 \cdot h_1[r_j]h'_0 \cdot h_1$. By the contents independence of r_j, we have that $h_0 \cdot h_2[r_j]h'_0 \cdot h_2$. Since r includes r_j, we also have $h_0 \cdot h_2[r]h'_0 \cdot h_2$, as required. □

3 Programming Language

The programming language is Dijkstra's language of guarded commands [5] extended with low-level pointer operations and module operations. The syntax of the language is given by the grammar:

$$C ::= f \mid a \mid C; C \mid C [] C \mid P \mid \mathsf{fix}\, P.\, C$$

where f, a, P are, respectively, chosen from three disjoint sets $\mathsf{mop}, \mathsf{aop}, \mathsf{pid}$ of identifiers. The first construct f is a module operation declared in the "interface specification" mop. Before a command in our language gets executed, it is first "linked" to a specific module that implements the interface mop. This linked module provides the meaning of the command f. The second construct a is an atomic operation, which a client can execute without using the module operations. Usually, a denotes a low-level pointer operation. Note that the language does not provide a syntax for building specific pointer operations. Instead, we assume that the interpretation $[\![-]\!]_a$ of these atomic client operations as av-free FLAs is given along with aop, and that under this interpretation, aop includes at least all the pointer operations in Lemma 1, so that aop includes all the atomic pointer operations considered in separation logic. The remaining four constructs of the language are the usual compound commands from Dijkstra's language: sequential composition $C; C$, nondeterministic choice $C [] C$, the call of a parameterless procedure P, and the recursive definition $\mathsf{fix}\, P.\, C$ of a parameterless procedure. As in Dijkstra's language, the construct $\mathsf{fix}\, P.\, C$ not only defines a parameterless recursive procedure P, but also calls the defined procedure. We express that a command C does not have free procedure names, by calling C a *complete command*.

Note that all the usual constructs of the while language can be expressed in this language. For example, conditional statement $\mathsf{if}\ (*l \in I)\ \mathsf{then}\ C\ \mathsf{else}\ C'$ can be expressed as $(\mathsf{test}(l, I); C) [] (\mathsf{test}(l, \overline{I}); C')$, where $\overline{I}$ is the complement $\mathsf{Int} - I$ of I. And, the allocation-status testing $\mathsf{check2}$ can be expressed as:

$$\mathsf{cons}(3, 1); \Big(\big(\mathsf{test}(3, \{2\}); \mathsf{update}(3, 1)\big) [] \big(\mathsf{test}(3, \mathsf{Int} - \{2\}); \mathsf{update}(3, 2)\big) \Big).$$

We interpret commands using an instrumented denotational semantics; besides computing the usual state transformation, the semantics also checks whether

$$\mu \in \mathcal{E} \stackrel{def}{=} \mathsf{pid} \to \mathcal{F} \qquad [\![C]\!]_{(p,\eta)} : \mathcal{E} \to \mathcal{F} \qquad [\![C]\!]^c_{(p,\eta)} : \mathcal{F} \ \text{(for complete } C\text{)}$$

$$[\![a]\!]_{(p,\eta)}\mu \stackrel{def}{=} \mathsf{prot}([\![a]\!]_a, p) \qquad [\![C[]C']\!]_{(p,\eta)}\mu \stackrel{def}{=} [\![C]\!]_{(p,\eta)}\mu \cup [\![C]\!]_{(p,\eta)}\mu$$

$$[\![f]\!]_{(p,\eta)}\mu \stackrel{def}{=} \eta(f) \qquad [\![P]\!]_{(p,\eta)}\mu \stackrel{def}{=} \mu(P)$$

$$[\![C;C']\!]_{(p,\eta)}\mu \stackrel{def}{=} \mathsf{seq}([\![C]\!]_{(p,\eta)}\mu, [\![C']\!]_{(p,\eta)}\mu) \qquad [\![\mathsf{fix}\, P.\, C]\!]_{(p,\eta)}\mu \stackrel{def}{=} \mathsf{fix}\, \lambda r.\, [\![C]\!]_{(p,\eta)}(\mu[P{\to}r])$$

$$[\![C]\!]^c_{(p,\eta)} \stackrel{def}{=} \mathsf{seq}(\mathsf{seq}(\eta(\mathsf{init}), [\![C]\!]_{(p,\eta)}\bot), \eta(\mathsf{final})) \ \text{(for complete } C\text{)}$$

where $\mathsf{seq}\colon \mathcal{F}{\times}\mathcal{F}{\to}\mathcal{F}$ and $\mathsf{prot}\colon \mathcal{F}{\times}\mathsf{Pred}{\to}\mathcal{F}$ are defined as follows:

$$h[\mathsf{prot}(r,p)]v \stackrel{def}{\Leftrightarrow} h[r]v \vee (v{=}\mathsf{av} \wedge \neg h[r]\mathsf{flt} \wedge \exists h_p, h_0.\, h{=}h_p{\cdot}h_0 \wedge h_p \in p \wedge h_0[r]\mathsf{flt})$$

$$h[\mathsf{seq}(r,r')]v \stackrel{def}{\Leftrightarrow} (\exists h'.\, h[r]h' \wedge h'[r']v) \vee (h[r]\mathsf{flt} \wedge v{=}\mathsf{flt}) \vee (h[r]\mathsf{av} \wedge v{=}\mathsf{av})$$

Fig. 3. Semantics of Language

each atomic client operation accesses the internals of a module, and for such illegal accesses, the semantics generates an access violation av.

To implement the instrumentation, we parameterize the semantics by what we call a *semantic module*. Let init and final be identifiers that are not in mop. A semantic module is a pair of a predicate p and a function η from $\mathsf{mop}{\cup}\{\mathsf{init}, \mathsf{final}\}$ to $\mathcal{F}_{noav}$, such that (1) $\forall h, h'.\, (\mathsf{safe}(\eta(\mathsf{init}), h) \wedge h[\eta(\mathsf{init})]h' \Rightarrow h'{\in}p{*}\mathsf{true})$; (2) for all f in mop, $\forall h, h'.\, (\mathsf{safe}(\eta(f), h) \wedge h{\in}p{*}\mathsf{true} \wedge h[\eta(f)]h' \Rightarrow h'{\in}p{*}\mathsf{true})$; (3) p is precise. Intuitively, the predicate p in the semantic module denotes the resource invariant for the module internals, and function η specifies the meaning of the module operations, initialization init and finalization final. The first condition of the semantic module requires initialization to establish the resource invariant, and the second condition, that the established resource invariant be preserved by module operations. The last condition is more subtle. It ensures that using the invariant p, we can determine which part of each state belongs to the module. Recall that a predicate q is precise iff every state h in $q * \mathsf{true}$ has a unique splitting $h_q{\cdot}h_0 = h$ such that $h_q \in q$. Thus, if p is precise, then for every state h containing both the internals and externals of the module (i.e., $h \in p * \mathsf{true}$), we can unambiguously split h into module-owned part h_p and client-owned part h_0. This unambiguous splitting is used in the semantics to detect the access violation of the atomic client operations, and it also plays a crucial role in the soundness of our refinement method (Sect. 5.1). We remark that requiring the preciseness of the invariant p is not as restrictive as one might think, because most of the used resource invariants are precise; among the used resource invariants in separation logic, only one invariant is not precise, but even that invariant can safely be tightened to a precise one.[4]

Let $\mathcal{E}$ be the poset of all functions from pid to $\mathcal{F}$ ordered pointwise. Given semantic module (p, η), we interpret a command as a continuous function $[\![-]\!]_{(p,\eta)}$

[4] The only known unprecise invariant is $\mathsf{listseg}(x, y)$ in [17], which means the existence of a (possibly cyclic) linked list segment from x to y. However, even that invariant can be made precise, if it is restricted to forbid a cycle in the list segment [14].

from $\mathcal{E}$ to $\mathcal{F}$. For complete commands C, we consider an additional interpretation $[\![-]\!]^c_{(p,\eta)}$ that uses the least environment $\bot = \lambda P.\emptyset$, and runs the initialization and the finalization of the module (p, η) before and after $([\![C]\!]_{(p,\eta)}\bot)$, respectively. The details of these two interpretations are shown in Fig. 3.

The most interesting part of the semantics lies in the interpretation of the atomic client operations. For each atomic operation a, its interpretation first looks up the original meaning $[\![a]\!]_a \in \mathcal{F}_{noav}$, which is given when the syntax of the language is defined. Then, the interpretation transforms the meaning into $\mathsf{prot}([\![a]\!]_a, p)$, "the p-protected execution of $[\![a]\!]_a$." Intuitively, $\mathsf{prot}([\![a]\!]_a, p)$ behaves the same as $[\![a]\!]_a$, except that whenever $[\![a]\!]_a$ accesses the p-part of the input state, $\mathsf{prot}([\![a]\!]_a, p)$ generates av, thus indicating that there is an "access violation." Since p is the resource invariant of the module, $\mathsf{prot}([\![a]\!]_a, p)$ notifies all illegal accesses to the module internals, by generating av.

Lemma 3. *Function* seq *is a continuous map from* $\mathcal{F} \times \mathcal{F}$ *to* $\mathcal{F}$

Proof. Let r, r' be FLAs. We first prove that $\mathsf{seq}(r, r')$ is a FLA. Since it is well known that $\mathsf{seq}(r, r')$ satisfies the safety monotonicity and frame property [20], we focus on the finite access property and contents independence. To show that $\mathsf{seq}(r, r')$ satisfies the finite access property, consider states h_0, h'_0 such that $\neg h_0[\mathsf{seq}(r, r')]\mathsf{flt}$ and $h_0[\mathsf{seq}(r, r')]h'_0$. Then, there exists an intermediate state m_0 such that

$$h_0[r]m_0 \ \wedge \ m_0[r']h'_0.$$

Since $\neg h_0[\mathsf{seq}(r, r')]\mathsf{flt}$, by the definition of seq, we have that

$$(\neg h_0[r]\mathsf{flt}) \ \wedge \ (\neg m_0[r']\mathsf{flt}).$$

Thus, we can use the finite access property of r and r' for $h_0[r]m_0$ and $m_0[r']h'_0$. Let L, L' be the finite sets of indirectly accessed locations for $h_0[r]m_0$ and $m_0[r']h'_0$, respectively. We will show that the required set L'' for $h_0[\mathsf{seq}(r, r')]h'_0$ is $L \cup L' \cup \mathsf{dom}(m_0)$. For all states h_1 such that

$$\mathsf{dom}(h_1) \cap \big(\mathsf{dom}(h_0) \cup \mathsf{dom}(h'_0) \cup L \cup L' \cup \mathsf{dom}(m_0)\big) = \emptyset,$$

$\mathsf{dom}(h_1)$ is disjoint from $\mathsf{dom}(h_0) \cup \mathsf{dom}(m_0) \cup L$ and $\mathsf{dom}(m_0) \cup \mathsf{dom}(h'_0) \cup L'$. Thus,

$$h_0{\cdot}h_1[r]m_0{\cdot}h_1 \ \wedge \ m_0{\cdot}h_1[r']h'_0{\cdot}h_1.$$

This implies $h_0{\cdot}h_1[\mathsf{seq}(r, r')]h'_0{\cdot}h_1$, as required.

We now show that $\mathsf{seq}(r, r')$ satisfies the contents independence. Consider states h_0, h'_0, h_1, h_2 such that

$$h_1\#h_0 \wedge h_1\#h'_0 \wedge \neg h_0[\mathsf{seq}(r, r')]\mathsf{flt} \wedge h_0{\cdot}h_1[\mathsf{seq}(r, r')]h'_0{\cdot}h_1 \wedge \mathsf{dom}(h_1) = \mathsf{dom}(h_2)$$

Then, there exists an intermediate state m such that

$$h_0{\cdot}h_1[r]m \ \wedge \ m[r']h'_0{\cdot}h_1.$$

Since $\neg h_0[\mathsf{seq}(r, r')]\mathsf{flt}$, by the definition of seq, we have that

$$\neg h_0[r]\mathsf{flt}.$$

Thus, we can apply the frame property of r to $h_0{\cdot}h_1[r]m$. If we apply the frame property, then we get a substate m_0 of m such that

$$m = m_0{\cdot}h_1 \ \wedge \ h_0[r]m_0.$$

Since $\neg h_0[\mathsf{seq}(r, r')]\mathsf{flt}$, this substate m_0 should be a safe input for r': $\neg m_0[r']\mathsf{flt}$. We now use the contents independence of r and r'. We replace h_1 by h_2 in the computation $h_0{\cdot}h_1[r]m_0{\cdot}h_1$ and $m_0{\cdot}h_1[r']h_0'{\cdot}h_1$, and obtain the following new computations:

$$h_0{\cdot}h_2[r]m_0{\cdot}h_2 \ \wedge \ m_0{\cdot}h_2[r']h_0'{\cdot}h_2.$$

The obtained computations show that $h_0{\cdot}h_2[\mathsf{seq}(r, r')]h_0'{\cdot}h_2$.

Next, we prove that seq is continuous. Consider a chain $\{(r_i, r_i')\}_{i\in\omega}$ of FLA pairs. Then,

$$\begin{array}{ll}
& h[\mathsf{seq}(\bigcup_{i\in\omega} r_i, \bigcup_{i\in\omega} r_i')]v \\
\iff & \\
& \big(h[\bigcup_{i\in\omega} r_i]v \ \wedge \ (v = \mathsf{flt} \ \vee \ v = \mathsf{av})\big) \ \vee \ \big(\exists h'.\ h[\bigcup_{i\in\omega} r_i]h' \ \wedge \ h'[\bigcup_{i\in\omega} r_i']v\big) \\
\iff & (\because \{(r_i, r_i')\}_{i\in\omega} \text{ is a chain}) \\
& \big(\exists i.\ h[r_i]v \ \wedge \ (v = \mathsf{flt} \ \vee \ v = \mathsf{av})\big) \ \vee \ \big(\exists i.\ \exists h'.\ h[r_i]h' \ \wedge \ h'[r_i']v\big) \\
\iff & \\
& \exists i.\ \Big(\big(h[r_i]v \ \wedge \ (v = \mathsf{flt} \ \vee \ v = \mathsf{av})\big) \ \vee \ \big(\exists h'.\ h[r_i]h' \ \wedge \ h'[r_i']v\big)\Big) \\
\iff & \\
& \exists i.\ h[\mathsf{seq}(r_i, r_i')]v \\
\iff & \\
& h[\bigcup_{i\in\omega} \mathsf{seq}(r_i, r_i')]v
\end{array}$$

□

Lemma 4. *For every FLA $r \in \mathcal{F}$ and every precise predicate p, action $\mathsf{prot}(r, p)$ is a finite local action.*

Proof. Action $\mathsf{prot}(r, p)$ is identical to r when both are restricted to $\mathsf{St} \times (\mathsf{St} \cup \{\mathsf{flt}\})$. Note that all of the safety monotonicity, frame property, finite access property and contents independence only concern the state or flt outputs, and that those properties are satisfied by r. Thus, they are also satisfied by $\mathsf{prot}(r, p)$. □

Lemma 5. *The interpretation in Fig. 3 is well-defined.*

Proof. We use the induction on the structure of C. When C is either an atomic client operation a or a module operation f, $[\![C]\!]_{(p,\eta)}$ is a constant function from $\mathcal{E}$ to $\mathcal{F}$ (Lemma 4), and so, it is continuous. When C is a procedure name P, $[\![C]\!]_{(p,\eta)}$ is a projection map, and so, it is continuous as well. The cases of the sequential composition $C_1; C_2$ and the choice operator $C_1 [] C_2$ follow from the

fact that $\cup$ and seq are continuous operators from $\mathcal{F} \times \mathcal{F}$ to $\mathcal{F}$ (Lemma 3). In both cases, the semantics of C is given by the composition of some continuous function $k\colon \mathcal{F} \times \mathcal{F} \to \mathcal{F}$ with

$$k' = \lambda\mu.\, \langle [\![C_1]\!]_{(p,\eta)}\mu, [\![C_2]\!]_{(p,\eta)}\mu \rangle : \mathcal{E} \to \mathcal{F} \times \mathcal{F}.$$

By the induction hypothesis, k' is continuous, and so, the semantics $[\![C]\!]_{(p,\eta)}$, given by $k \circ k'$, is continuous as well. The final case is when C is $\mathsf{fix}\, x.C$. In this case, $[\![C]\!]_{(p,\eta)}$ is the composition of the continuous least-fixed-point operator $\mathsf{fix}\colon [\mathcal{F} \to \mathcal{F}] \to \mathcal{F}$ with the following function k':

$$k' = \lambda\mu \in \mathcal{E}.\, \lambda r \in \mathcal{F}.\, [\![C]\!]_{(p,\eta)}(\mu[x \to r]).$$

We will show that k' is a continuous function from $\mathcal{E}$ to $[\mathcal{F} \to \mathcal{F}]$. For all environments μ, and for all chains $\{r_i\}_{i\in\omega}$ of finite local actions, $\{\mu[x \to r_i]\}_{i\in\omega}$ is a chain whose least upper bound is $\mu[x \to \bigcup_{i\in\omega} r_i]$. So, by the induction hypothesis,

$$[\![C]\!]_{(p,\eta)}(\mu[x \to \bigcup_{i\in\omega} r_i]) = [\![C]\!]_{(p,\eta)}(\bigsqcup_{i\in\omega}(\mu[x \to r_i])) = \bigcup_{i\in\omega} [\![C]\!]_{(p,\eta)}(\mu[x \to r_i]).$$

Thus, k' is a well-defined function from $\mathcal{E}$ to $[\mathcal{F} \to \mathcal{F}]$. We now show that k' is indeed continuous. Let $\{\mu_i\}_{i\in\omega}$ be a chain of environments. Then, for all r in $\mathcal{F}$,

$$[\![C]\!]_{(p,\eta)}((\bigsqcup_{i\in\omega} \mu_i)[x \to r]) = [\![C]\!]_{(p,\eta)}(\bigsqcup_{i\in\omega}(\mu_i[x \to r])) = \bigcup_{i\in\omega}([\![C]\!]_{(p,\eta)}(\mu_i[x \to r])).$$

Thus, k' is continuous. □

4 Data Refinement

The goal of this paper is to find a method for proving that a "concrete" module (q, ϵ) data-refines an "abstract" module (p, η). In this section, we first formalize this goal by defining the notion of data refinement. Then, we demonstrate the difficulty of achieving the goal, by showing that the standard forward method is not sound in the presence of allocation-status testing.

We use the notion of data refinement that Mijajlović *et al.* devised in order to handle cross-boundary pointers. Usually, data refinement is a relation between modules defined by substitutability: a module (q, ϵ) data-refines another module (p, η) iff for all complete commands C using (p, η), substituting the concrete module (q, ϵ) for the abstract module (p, η) improves the behavior of C, i.e., C becomes more deterministic with the concrete module. Mijajlović *et al.* weakened this usual notion of data refinement, by dropping the requirement about improvement for error-generating input states: if C with the abstract module (p, η) generates an access violation av or a memory fault flt from an input state h, then for this input h, the data refinement does not constrain the execution of C with the concrete module (q, ϵ), and allows it to generate any outputs. In this paper, we use the following formalization of this weaker notion of data refinement:

Definition 2 (Data Refinement). *A module* (q, ϵ) data-refines *another module* (p, η) *iff for all complete commands* C *and all states* h*, if* $[\![C]\!]^c_{(p,\eta)}$ *does not generate an error from* h *(i.e.,* $\neg h[[\![C]\!]^c_{(p,\eta)}]\mathsf{av} \wedge \neg h[[\![C]\!]^c_{(p,\eta)}]\mathsf{flt}$*), then*

$$\big(\neg h[[\![C]\!]^c_{(q,\epsilon)}]\mathsf{av} \ \wedge \ \neg h[[\![C]\!]^c_{(q,\epsilon)}]\mathsf{flt}\big) \ \wedge \ \big(\forall h'.\ h[[\![C]\!]^c_{(q,\epsilon)}]h' \ \Rightarrow \ h[[\![C]\!]^c_{(p,\eta)}]h'\big).$$

The main benefit of considering this notion of data refinement is that a proof method for data refinement does not have to do anything special in order to handle the cross-boundary pointers. Recall that flt means that a command tries to dereference dangling pointers or nil, and av means that a command attempts to dereference the internal cells of a module without using module operations. Thus, if a command C does not generate an error from an input state h, then all the cells that C directly dereferences during execution must be allocated and belong to the "client" portion of the state; in particular, C does not dereference any cross-boundary pointers directly. Since the data refinement now asks for the improvement of only the error-free computations of C, a proof method for data refinement can ignore the "bad" computations where C dereferences cross-boundary pointers.

Unfortunately, even with this weaker notion of data refinement, standard proof methods for data refinement are not sound; they fail to deal with the allocation-status testing. We explain this soundness problem making use of the notion of the forward simulation in [12]. As pointed out in their work, while successfully dealing with the cross-boundary pointer dereferencing problem, the forward method is not sound for allocation-status testing.

The key concept of the forward simulation in [12] is an operator fsim that maps a pair (R_0, R_1) of state relations to a relation $\mathsf{fsim}(R_0, R_1)$ between FLAs. Intuitively, $r'[\mathsf{fsim}(R_0, R_1)]r$ means that given R_0-related input states h' and h, if r does not generate an error from h, then (1) r' does not generates an error from h' and (2) every output of r' from h' is R_1-related to some outcome of r. More precisely, $r'[\mathsf{fsim}(R_0, R_1)]r$ iff for all states h' and h, if $(h'[R_0]h \wedge \neg h[r]\mathsf{flt} \wedge \neg h[r]\mathsf{av})$, then

$$\big(\neg h'[r']\mathsf{flt} \ \wedge \ \neg h'[r']\mathsf{av}\big) \ \wedge \ \big(\forall h'_1.\ h'[r']h'_1 \ \Rightarrow \ \exists h_1.\, h[r]h_1 \wedge h'_1[R_1]h_1\big).$$

The condition about the absence of errors comes from the fact that the data refinement considers only error-free computations. Except this condition, the way of relating two actions (or commands) in $\mathsf{fsim}(R_0, R_1)$ is fairly standard in the work on data refinement [6, 4].

Let Δ be the diagonal relation on states[5], and for state relations R_0 and R_1, let $R_0 * R_1$ be their relational separating conjunction [16]: $h'[R_0 * R_1]h$ iff h' and h are, respectively, split into $h'_0 \cdot h'_1 = h'$ and $h_0 \cdot h_1 = h$ such that the first parts h'_0, h_0 are related by R_0 and the second parts h'_1, h_1 by R_1.[6] The formal definition of forward simulation is given below:

[5] $h'[\Delta]h \stackrel{def}{\Leftrightarrow} h' = h$

[6] $h'[R_0 * R_1]h \stackrel{def}{\Leftrightarrow} \exists h'_0, h'_1, h_0, h_1.\ h'_0 \cdot h'_1 = h' \ \wedge \ h_0 \cdot h_1 = h \ \wedge \ h'_0[R_0]h_0 \ \wedge \ h'_1[R_1]h_1.$

Definition 3 (Forward Simulation). *Let* $(q,\epsilon),(p,\eta)$ *be semantic modules, and* R *a relation s.t.* $R \subseteq q \times p$. Module (q,ϵ) forward-simulates (p,η) by R *iff*

1. $\epsilon(\mathsf{init})[\mathsf{fsim}(\Delta, R*\Delta)]\eta(\mathsf{init})$ *and* $\epsilon(\mathsf{final})[\mathsf{fsim}(R*\Delta,\Delta)]\eta(\mathsf{final})$;
2. $\forall f \in \mathsf{mop}.\ \epsilon(f)[\mathsf{fsim}(R*\Delta, R*\Delta)]\eta(f)$.

The relation $R*\Delta$ here expresses that the corresponding states of r' and r can, respectively, be partitioned into the module and client parts; the module parts of r' and r are related by R, but the client parts of r' and r are the same.

The forward simulation is not sound: there are modules $(q,\epsilon),(p,\eta)$ such that the concrete module (q,ϵ) forward-simulates the abstract module(p,η) by some $R \subseteq q \times p$, but it does not data-refine it. The main reason of this unsoundness is that the low-level pointer operations in our language, especially those implementing allocation-status testing, break the underlying assumption of the forward simulation. The forward simulation assumes a language where if a command C does not call module operations, then for all relations $R \subseteq q \times p$, the command "forward-simulates" itself by R: $[\![C]\!]_{(q,\epsilon)}\mu'[\mathsf{fsim}(R*\Delta, R*\Delta)][\![C]\!]_{(p,\eta)}\mu$ for all μ', μ that define $\mathsf{fsim}(R*\Delta, R*\Delta)$-related "procedures". Our language, however, does not satisfy this assumption; if an atomic client command a implements the allocation-status testing, it is not related to itself by $\mathsf{fsim}(R*\Delta, R*\Delta)$ in general. For instance, having a concrete module (q,ϵ) and the abstract one (p,η) and a relation R between them, consider an atomic command $\mathsf{cons}(2,1)$ that allocates one new cell initialized to 0 and assigns its address to cell 2; in case that cell 2 is not allocated initially, $\mathsf{cons}(2,1)$ generates flt.[7] Let R be defined by $h_0'[R]h_0 \Leftrightarrow h_0' = [] \wedge h_0 = [1{\to}2]$. Then, $h'[R{*}\Delta]h$ iff there is some state h_1 such that $1 \notin \mathsf{dom}(h_1) \wedge h' = []{\cdot}h_1 \wedge h = [1{\to}2]{\cdot}h_1$. Thus, states $h' = [2{\to}0]$ and $h = [1{\to}2, 2{\to}0]$ are $R{*}\Delta$-related. We will now consider the execution of $\mathsf{cons}(2,1)$ from these $R{*}\Delta$-related states h' and h. When $\mathsf{cons}(2,1)$ is run from h' (with the concrete module (q,ϵ)), it can allocate cell 1 and give the output state $h_1' = [1{\to}0, 2{\to}1]$ (i.e., $h_1[[\![\mathsf{cons}(2,1)]\!]_{(q,\epsilon)}\mu']h_1'$), because cell 1 is free initially (i.e., $1 \notin \mathsf{dom}(h')$). However, when the same command is run from h (with the abstract module (p,η)), it cannot allocate cell 1, because 1 is already active in h (i.e., $1 \in \mathsf{dom}(h)$). In this case, all the output states of $\mathsf{cons}(2,1)$ have the form $[1{\to}0, 2{\to}n, n{\to}0]$ for some $n \in \mathsf{Nats}{-}\{1,2\}$. Note that the state $h_1' = [1{\to}0, 2{\to}1]$ is not $R{*}\Delta$-related to any such outputs $[1{\to}0, 2{\to}n, n{\to}0]$. Thus, we cannot have that $\big([\![\mathsf{cons}(2,1)]\!]_{(q,\epsilon)}\mu'\big)[\mathsf{fsim}(R{*}\Delta, R{*}\Delta)]\big([\![\mathsf{cons}(2,1)]\!]_{(p,\eta)}\mu\big)$. In Appendix A, we use these R and cons to construct a counter example for the soundness of the forward simulation.

5 Power Simulation

We now present the main result of this paper: a new method for data refinement, called power simulation, and its soundness proof.

[7] $h[[\![\mathsf{cons}(2,1)]\!]_a]v \overset{def}{\Leftrightarrow} \mathsf{if}\ 2 \notin \mathsf{dom}(h)\ \mathsf{then}\ v{=}\mathsf{flt}\ \mathsf{else}\ \exists n.\, v{=}h[2{\to}n]{\cdot}[n{\to}0]$.

The key idea of power simulation is to use the state-set lifting $\mathsf{lft}(r)$ of a FLA:

$$\begin{array}{l} \mathsf{lft}(r) \ : \ \wp(\mathsf{St}) \leftrightarrow (\wp(\mathsf{St}) \cup \{\mathsf{flt}, \mathsf{av}\}) \\ H[\mathsf{lft}(r)]V \stackrel{def}{\Leftrightarrow} (V \subseteq \mathsf{St} \wedge \forall h' \in V. \exists h \in H.\, h[r]h') \vee ((V = \mathsf{av} \vee V = \mathsf{flt}) \wedge \exists h \in H.\, h[r]V). \end{array}$$

Given an input state set H, the "lifted command" $\mathsf{lft}(r)$ runs r for all the states in H, chooses some states among the results, and returns the set V of the chosen states. Note that V might not contain some possible outputs from H; so, $\mathsf{lft}(r)$ is different from the usual direct image map of r, and in general, it is a relation rather than a function. For each module (p, η), we write $\mathsf{lft}(\eta)$ for the lifting of all module operations (i.e., $\forall f \in \mathsf{mop}.\ \mathsf{lft}(\eta)(f) = \mathsf{lft}(\eta(f))$), and call $(p, \mathsf{lft}(\eta))$ the *lifting* of (p, η).

The power simulation is the usual forward simulation of a *lifted* "abstract" module by a *normal* "concrete" module. Suppose that we want to show that a concrete module (q, ϵ) data-refines an abstract module (p, η). Define a power relation to be a relation between states and state sets. Intuitively, the power simulation says that to prove this data refinement, we only need to find a "good" power relation $\mathcal{R} \subseteq \mathsf{St} \times \wp(\mathsf{St})$ such that every concrete-module operation $\epsilon(k)$ "forward-simulates" the corresponding lifted abstract-module operation $\mathsf{lft}(\eta(k))$ by $\mathcal{R}$. The official definition of power simulation formalizes this intuition by specifying (1) which power relation should be considered good for given modules (q, ϵ) and (p, η), and (2) what it means that a normal command "forward-simulates" a lifted command. For the first, we use the *expansion* operator and *admissibility* condition for power relations. For the second, we use the operator psim that maps a power-relation pair to a relation on FLAs. We will now define these subcomponents of power simulation, and use them to give the formal definition of power simulation.

We explain operator psim first. For power relations $\mathcal{R}_0$ and $\mathcal{R}_1$, $\mathsf{psim}(\mathcal{R}_0, \mathcal{R}_1)$ relates a "concrete" FLA r' with an "abstract" r iff for every $\mathcal{R}_0$-related input state h' and state set H, if $\mathsf{lft}(r)$ does not generate an error from H, then all the outputs of r' from h' are $\mathcal{R}_1$-related to some output state sets of $\mathsf{lft}(r)$ from H. More precisely, $r'[\mathsf{psim}(\mathcal{R}_0, \mathcal{R}_1)]r$ iff for all h' and H, if $h'[\mathcal{R}_0]H$ and neither $H[\mathsf{lft}(r)]\mathsf{flt}$ nor $H[\mathsf{lft}(r)]\mathsf{av}$, then

$$\big(\neg h'[r']\mathsf{flt} \ \wedge \ \neg h'[r']\mathsf{av}\big) \ \wedge \ \big(\forall h'_1.\, h'[r']h'_1 \ \Rightarrow \ \exists H_1.\, H[\mathsf{lft}(r)]H_1 \wedge h'_1[\mathcal{R}_1]H_1\big).$$

Note that this definition is the lifted version of fsim in Sec. 4; except that it considers the lifted computation $\mathsf{lft}(r)$, instead of the usual computation r, it coincides with the definition of fsim. In the definition of power simulation, we will use this psim to express the "forward-simulation" of a lifted command by a normal command.

Next, we define the expansion operator $-\otimes\Delta$ for power relations. The expansion $\mathcal{R}\otimes\Delta$ of a power relation $\mathcal{R}$ is a power relation defined as follows:

$$h[\mathcal{R}\otimes\Delta]H \ \stackrel{def}{\Leftrightarrow} \ \exists h_r, h_0, H_r.\big(h = h_r \cdot h_0 \ \wedge \ h_r[\mathcal{R}]H_r \ \wedge \ H = H_r * \{h_0\}\big).$$

Intuitively, the definition means that h and H are obtained by extending $\mathcal{R}$-related state h_r and state sets H_r by the same state h_0. Usually, $\mathcal{R}$ is a "coupling" power relation that connects the internals of two modules, and $\mathcal{R}\otimes\Delta$ expands this coupling relation to the relation for the entire memory, by asking that the added client parts must be identical.

The final subcomponent of power simulation is the admissibility condition for power relations. A power relation $\mathcal{R}$ is *admissible* iff for every $\mathcal{R}$-related state h and state set H (i.e., $h[\mathcal{R}]H$), we have that[8]

$$H \neq \emptyset \ \wedge \ \big(\forall L{\subseteq_{fin}}\mathsf{Loc}{-}\mathsf{dom}(h).\ \exists H_1{\subseteq}H.\ \big(H_1{\neq}\emptyset \wedge h[\mathcal{R}]H_1 \wedge \forall h_1{\in}H_1.\ h_1\#L\big)\big).$$

The first conjunct in the admissibility condition means that all related state sets must contain at least one state. The second conjunct is about the "free cells" in these related state sets. It means that if $h[\mathcal{R}]H$, state set H collectively has at least as many free cells as h: for every finite collection L of free cells in h, set H contains states that do not have any of the cells in L, and, moreover, the set H_1 of such states itself collectively has as many free cells as h. To understand the second conjunct more clearly, consider power relations $\mathcal{R}_0, \mathcal{R}_1, \mathcal{R}_2$ defined as follows:

$$h[\mathcal{R}_0]H \overset{def}{\Leftrightarrow} h{=}[3{\to}1] \wedge H{=}\{[3{\to}5]\} \qquad h[\mathcal{R}_1]H \overset{def}{\Leftrightarrow} h{=}[3{\to}1] \wedge H{=}\{[3{\to}5, 4{\to}5]\}$$
$$h[\mathcal{R}_2]H \ \overset{def}{\Leftrightarrow}\ h{=}[3{\to}1] \wedge \exists L{\subseteq_{fin}}\mathsf{Loc}.\ H{=}\{[3{\to}5, n{\to}5] \mid n \notin L \cup \{3\}\}$$

The first power relation $\mathcal{R}_0$ is admissible, because set $\{[3{\to}5]\}$ has only one state $[3{\to}5]$ that has the exactly same free cells, namely all cells other than 3, as state $[3{\to}1]$. On the other hand, $\mathcal{R}_1$ is not admissible, because the (unique) state in $\{[3{\to}5, 4{\to}5]\}$ has an active cell 4 that is not free in $[3{\to}1]$. The last relation $\mathcal{R}_2$ is tricky; relation $\mathcal{R}_2$ is admissible, even though for all $\mathcal{R}_2$-related h and H, every state in H has more active cells than h. The intuitive reason for this is that for every free cell in $[3{\to}1]$, set H contains a state that does not contain the cell, and so, it collectively has as many free cells as $[3{\to}1]$; in a sense, by having sufficiently many states, H hides the identity of the additional cell n. The formal proof that $\mathcal{R}_2$ satisfies the second conjunct of the admissibility condition proceeds as follows. Consider H, h', L_1 such that $h'[\mathcal{R}_2]H$ and $L_1 \subseteq_{fin} (\mathsf{Loc}{-}\mathsf{dom}(h))$. By the definition of $\mathcal{R}_2$, there exists a finite location set L such that $H = \{[3{\to}5, n{\to}5] \mid n \notin L \cup \{3\}\}$. Let $H_1 = \{[3{\to}5, n{\to}5] \mid n \notin L \cup L_1 \cup \{3\}\}$. The defined set H_1 is a nonempty subset of H. We now prove that H_1 is in fact the required subset of H in the admissibility condition. Since $h'[\mathcal{R}_2]H_1$, $h'[\mathcal{R}_2]H_1$ follows from the definition of $\mathcal{R}_2$ and H_1. We also have that $\forall h_1 \in H_1.\ \mathsf{dom}(h_1) \cap L_1 = \emptyset$, because $\mathsf{dom}(h_1) \cap L_1 \subseteq \{3\}$ but L_1 does not contain 3 ($h'{=}[3{\to}1]\#L_1$).

Using the expansion operator and admissibility condition, we can define the criteria for deciding which power relation should be considered "good" for given modules (q, ϵ) and (p, η). The criteria is: a power relation should be the expansion $\mathcal{R}\otimes\Delta$ of an admissible $\mathcal{R}$ for the module internals (i.e., $\mathcal{R} \subseteq q \times \wp(p)$). The following lemma, which we will prove later in Sec. 5.1, provides the justification of this criteria:

[8] Recall that $h_1\#L$ iff $\mathsf{dom}(h_1) \cap L = \emptyset$.

LEMMA 6: For all q, p, and all power relations $\mathcal{R} \subseteq q \times \wp(p)$, if $\mathcal{R}$ is admissible and q is precise, then $\forall r \in \mathcal{F}_{noav}.\ \mathsf{prot}(r, q)[\mathsf{psim}(\mathcal{R}\otimes\Delta, \mathcal{R}\otimes\Delta)]\mathsf{prot}(r, p)$.

To see the significance of this lemma, recall that the forward simulation in Sec. 4 failed to be sound mainly because some atomic client operations are not related to themselves by fsim. The lemma indicates that as long as we are using admissible power relation $\mathcal{R}$, we do not have such a problem for psim: if $\mathcal{R}$ is admissible, then for all atomic client operations a and all environment pairs (μ', μ) with $\mathsf{psim}(\mathcal{R}\otimes\Delta, \mathcal{R}\otimes\Delta)$-related procedures, we have that $[\![a]\!]_{(q,\epsilon)}\mu'[\mathsf{psim}(\mathcal{R}\otimes\Delta, \mathcal{R}\otimes\Delta)][\![a]\!]_{(p,\eta)}\mu$.

We now define the power simulation of an abstract module (p, η) by a concrete module (q, ϵ). Let $\mathcal{R}$ be an admissible power relation such that $\mathcal{R} \subseteq q \times \wp(p)$, and let ID be the "identity" power relation defined by: $h[\mathsf{ID}]H \stackrel{def}{\Leftrightarrow} \{h\} = H$.

Definition 4 (Power Simulation). Module (q, ϵ) power-simulates (p, η) by $\mathcal{R}$ *iff*

1. $\epsilon(\mathsf{init})[\mathsf{psim}(\mathsf{ID}, \mathcal{R}\otimes\Delta)]\eta(\mathsf{init})$ *and* $\epsilon(\mathsf{final})[\mathsf{psim}(\mathcal{R}\otimes\Delta, \mathsf{ID})]\eta(\mathsf{final})$;
2. $\forall f \in \mathsf{mop}.\ \epsilon(f)[\mathsf{psim}(\mathcal{R}\otimes\Delta, \mathcal{R}\otimes\Delta)]\eta(f)$.

Example 1. We demonstrate power simulation using the semantic modules (q, ϵ) and (p, η) that, respectively, correspond to counter2 and counter3 in Fig. 1. Recall that both counter2 and counter3 implement a counter "object" with two operations, inc for incrementing the counter and read for reading the value of the counter; the main difference is that counter3 uses two cells, namely cell 1 and a newly allocated one, to track the value of the counter, while counter2 uses only cell 1 for the same purpose. The corresponding semantic modules, (q, ϵ) for counter2 and (p, η) for counter3, are defined in Fig. 4. Note that the resource invariant p indicates that counter3 uses two cells 1 and n internally, and the invariant q that counter2 uses only one cell 1 internally. We will now show that the space saving in counter2 is correct, by proving that (q, ϵ) power-simulates (p, η).

The first step of power simulation is to find an admissible power relation that couples the internals of (q, ϵ) and (p, η). For this, we use the following $\mathcal{R}$:

$$h[\mathcal{R}]H \stackrel{def}{\Leftrightarrow} \exists L, n.\ L \subseteq_{fin} \mathsf{Loc} \wedge n \geq 0 \wedge h = [1 \rightarrow n] \wedge H = \{[1 \rightarrow n', n' \rightarrow n] \mid n' \notin L \cup \{1\}\}.$$

Intuitively, $h[\mathcal{R}]H$ means that all the states in H and state h represent the same counter having the value $h(1)$, and moreover, H collectively has as many free cells as h.

The next step is to show that all the corresponding module operations of (q, ϵ) and (p, η) are related by psim. Here we only show that $\epsilon(\mathsf{init})$ and $\eta(\mathsf{init})$ are $\mathsf{psim}(\mathsf{ID}, \mathcal{R}\otimes\Delta)$-related. Consider h and H related by the "identity relation" ID. Then, by the definition of ID, set H must be the singleton set containing the heap h. Thus, it suffices to show that if $\mathsf{lft}(\eta(\mathsf{init}))$ does not generate an error from $\{h\}$, all the outputs of $\epsilon(\mathsf{init})$ from h are $\mathcal{R}\otimes\Delta$-related to some output state sets of $\mathsf{lft}(\eta(\mathsf{init}))$ from $\{h\}$. Suppose that $\mathsf{lft}(\eta(\mathsf{init}))$ does not generate an

$$
\begin{array}{rl}
h \in p & \stackrel{def}{\Leftrightarrow} \exists n, n'.\ n' \neq 1 \wedge n \geq 0 \wedge h = [1{\rightarrow}n', n'{\rightarrow}n] \\
h[\eta(\mathsf{init})]v & \stackrel{def}{\Leftrightarrow} \mathsf{if}\ (1{\notin}\mathsf{dom}(h))\ \mathsf{then}\ v{=}\mathsf{flt}\ \mathsf{else}\ \exists n.\ n{\notin}\mathsf{dom}(h) \wedge v{=}h[1{\rightarrow}n]{\cdot}[n{\rightarrow}0] \\
h[\eta(\mathsf{inc})]v & \stackrel{def}{\Leftrightarrow} \mathsf{if}\ (1{\notin}\mathsf{dom}(h) \vee h(1){\notin}\mathsf{dom}(h))\ \mathsf{then}\ v{=}\mathsf{flt}\ \mathsf{else}\ v{=}h[h(1){\rightarrow}(h(h(1)){+}1)]) \\
h[\eta(\mathsf{read})]v & \stackrel{def}{\Leftrightarrow} \mathsf{if}\ (1{\notin}\mathsf{dom}(h) \vee h(1){\notin}\mathsf{dom}(h) \vee 3{\notin}\mathsf{dom}(h))\ \mathsf{then}\ v{=}\mathsf{flt}\ \mathsf{else}\ v{=}h[3{\rightarrow}h(h(1))] \\
h[\eta(\mathsf{final})]v & \stackrel{def}{\Leftrightarrow} \mathsf{if}\ (1{\notin}\mathsf{dom}(h) \vee h(1){\notin}\mathsf{dom}(h))\ \mathsf{then}\ v{=}\mathsf{flt} \\
 & \quad \mathsf{else}\ \exists h_0.\ v{=}h_0[1{\rightarrow}0] \wedge h{=}h_0{\cdot}[h(1){\rightarrow}h(h(1))] \\
h \in q & \stackrel{def}{\Leftrightarrow} \exists n.\ n \geq 0 \wedge h = [1{\rightarrow}n] \\
h[\epsilon(\mathsf{init})]v & \stackrel{def}{\Leftrightarrow} \mathsf{if}\ (1{\notin}\mathsf{dom}(h))\ \mathsf{then}\ v{=}\mathsf{flt}\ \mathsf{else}\ v{=}h[1{\rightarrow}0] \\
h[\epsilon(\mathsf{inc})]v & \stackrel{def}{\Leftrightarrow} \mathsf{if}\ (1 \notin \mathsf{dom}(h))\ \mathsf{then}\ v{=}\mathsf{flt}\ \mathsf{else}\ v{=}h[1{\rightarrow}(h(1){+}1)] \\
h[\epsilon(\mathsf{read})]v & \stackrel{def}{\Leftrightarrow} \mathsf{if}\ (1{\notin}\mathsf{dom}(h) \vee 3{\notin}\mathsf{dom}(h))\ \mathsf{then}\ v{=}\mathsf{flt}\ \mathsf{else}\ v{=}h[3{\rightarrow}h(1)] \\
h[\epsilon(\mathsf{final})]v & \stackrel{def}{\Leftrightarrow} \mathsf{if}\ (1 \notin \mathsf{dom}(h))\ \mathsf{then}\ v{=}\mathsf{flt}\ \mathsf{else}\ v{=}h[1{\rightarrow}0]
\end{array}
$$

Fig. 4. Definition of Module (p, η) and (q, ϵ)

error from $\{h\}$. Then, $\eta(\mathsf{init})$ cannot output flt from h, and so, cell 1 should be in $\mathsf{dom}(h)$. From this, it follows that the concrete initialization $\epsilon(\mathsf{init})$ does not generate an error from h. We now check the non-error outputs of $\epsilon(\mathsf{init})$. When started from h, the concrete initialization $\epsilon(\mathsf{init})$ has only one non-error output, namely state $h[1{\rightarrow}0]$. We split this output state $h[1{\rightarrow}0]$ into $[1{\rightarrow}0]$ and the remainder h_0. By the definition of $\mathcal{R}$, the first part $[1{\rightarrow}0]$ of the splitting is $\mathcal{R}$-related to $H_r = \{[1{\rightarrow}n', n'{\rightarrow}0] \mid n' \notin \mathsf{dom}(h)\}$. Thus, extending $[1{\rightarrow}0]$ and H_r by the remainder h_0 gives $\mathcal{R}{\otimes}\Delta$-related state $[1{\rightarrow}0]{\cdot}h_0 = h[1{\rightarrow}0]$ and state set $H_r{*}\{h_0\}$. The state set $H_r{*}\{h_0\}$ is equal to $\{h[1{\rightarrow}n']{\cdot}[n'{\rightarrow}0] \mid n' \notin \mathsf{dom}(h)\}$, and so, it is a possible output of $\mathsf{lft}(\eta(\mathsf{init}))$ from $\{h\}$ by the definition of $\mathsf{lft}(\eta(\mathsf{init}))$. We have just shown that the output $h[1{\rightarrow}0]$ is $\mathcal{R}{\otimes}\Delta$-related to some output of $\mathsf{lft}(\eta(\mathsf{init}))$, as required.

The nondeterministic allocation in the abstract initialization $\eta(\mathsf{init})$ is crucial for the correctness of data refinement. Suppose that we change the initialization of the abstract module such that it allocates a specific cell 2:

$$h[\eta(\mathsf{init})]v \stackrel{def}{\Leftrightarrow} \mathsf{if}\ (1 \notin \mathsf{dom}(h))\ \mathsf{then}\ (v{=}\mathsf{flt})\ \mathsf{else}\ (2 \notin \mathsf{dom}(h)\ \wedge\ v{=}h[1{\rightarrow}2]{\cdot}[2{\rightarrow}0])$$

Then, (q, ϵ) no longer data-refines (p, η);[9] by testing the allocation status of cell 2 using memory allocation and pointer comparison, a client command can detect the replacement of (p, η) by (q, ϵ), and exhibit a behavior that is only possible with (q, ϵ), but not with (p, η). Power simulation correctly captures this failure of data refinement. More specifically, for all power relations $\mathcal{R} \subseteq q \times \wp(p)$ if $\epsilon(\mathsf{init})[\mathsf{psim}(\mathsf{ID}, \mathcal{R}{\otimes}\Delta)]\eta(\mathsf{init})$, then $\mathcal{R}$ cannot be admissible. To see the reason, suppose that $\epsilon(\mathsf{init})[\mathsf{psim}(\mathsf{ID}, \mathcal{R}{\otimes}\Delta)]\eta(\mathsf{init})$. When $\epsilon(\mathsf{init})$ and $\mathsf{lft}(\eta(\mathsf{init}))$ are run from ID-related $[1{\rightarrow}0]$ and $\{[1{\rightarrow}0]\}$, $\epsilon(\mathsf{init})$ outputs $[1{\rightarrow}0]$ and $\mathsf{lft}(\eta(\mathsf{init}))$ outputs $\{[1{\rightarrow}2, 2{\rightarrow}0]\}$ or $\emptyset$. Thus, by the definition of $\mathsf{psim}(\mathsf{ID}, \mathcal{R}{\otimes}\Delta)$, $[1{\rightarrow}0]$ should be

[9] Even when we replace p by a more precise invariant $\{[1{\rightarrow}2, 2{\rightarrow}n] \mid n \geq 0\}$, module (q, ϵ) does not data-refine (p, η).

$\mathcal{R}\otimes\Delta$-related to $\{[1\to 2, 2\to 0]\}$ or $\emptyset$. Then, by the definition of $\mathcal{R}\otimes\Delta$, state $[1\to 0]$ is $\mathcal{R}$-related to $\{[1\to 2, 2\to 0]\}$ or $\emptyset$. In either case, $\mathcal{R}$ is not admissible; the first case violates the second conjunct about the free cells in the admissibility condition, and the second case violates the first conjunct about the nonemptiness. □

5.1 Soundness of Power Simulation

The soundness of power simulation follows from the fact that every atomic client operation is related to itself by psim (Lemma 6), all language constructs preserve psim (Lemma 7,8) and $\mathsf{psim}(\mathsf{ID}, \mathsf{ID})$ is precisely the improvement requirement in the definition of data refinement (Lemma 9). In this section, we prove these lemmas, and show how the lemmas give the soundness of power simulation.

Lemma 6. *For all predicates q, p, and all power relations $\mathcal{R} \subseteq q \times \wp(p)$, if q is precise and $\mathcal{R}$ is admissible, then $\forall r{\in}\mathcal{F}_{noav}.\ \mathsf{prot}(r, q)[\mathsf{psim}(\mathcal{R}\otimes\Delta, \mathcal{R}\otimes\Delta)]\mathsf{prot}(r, p)$.*

Note that the lemma requires that the "resource invariant" q for the "concrete module" be precise, and that r be a av-free finite local action. None of these requirements can be omitted, because the requirements are used crucially in the proof of the lemma.

Proof. Let r_q be $\mathsf{prot}(r, q)$ and let r_p be $\mathsf{prot}(r, p)$. Pick arbitrary $[\mathcal{R}\otimes\Delta]$-related h and H such that $\mathsf{lft}(r_p)$ does not generate an error from H. Since $h[\mathcal{R}\otimes\Delta]H$, state h and state set H can, respectively, be split into $h_q{\cdot}h_0 = h$ and $H = H_p * \{h_0\}$ for some h_q, h_0, H_p such that $h_q[\mathcal{R}]H_p$. We note two facts about these splittings. First, set H_p contains a state that is disjoint from h_0. Since h_q and H_p are related by the admissible relation $\mathcal{R}$ and $\mathsf{dom}(h_q)$ is disjoint from $\mathsf{dom}(h_0)$, there is a nonempty subset of H_p such that every h_1 in the subset satisfies $h_1\#\mathsf{dom}(h_0)$. We pick one state from this subset, and call it h_p. Second, the state h_p in H_p and the part h_q of the splitting of h, respectively, belong to p and q. This second fact follows since $h_q[\mathcal{R}]H_p$ and $\mathcal{R} \subseteq q \times \wp(p)$. We sum up the obtained properties about H_p, h_0, h_q, h_p below:

$$H = H_p * \{h_0\} \ \wedge\ h = h_q{\cdot}h_0 \ \wedge\ h_q[\mathcal{R}]H_p \ \wedge\ h_p\#h_0 \ \wedge\ h_p \in p \ \wedge\ h_q \in q.$$

We now prove that r_q does not generate an error from h. Since the lifted command $\mathsf{lft}(r_p)$ does not generate an error from H and state $h_p{\cdot}h_0$ is in this input state set H, we have that $\neg h_p{\cdot}h_0[r_p]\mathsf{flt} \ \wedge\ \neg h_p{\cdot}h_0[r_p]\mathsf{av}$. This absence of errors of r_p ensures one important property of r: r cannot generate flt from h_0. To see the reason, note that h_p is in p, and that $\neg h_p{\cdot}h_0[r]\mathsf{flt}$ since $\neg h_p{\cdot}h_0[r_p]\mathsf{flt}$. So, if $h_0[r]\mathsf{flt}$, then by the definition of prot, we have that $h_p{\cdot}h_0[r_p]\mathsf{av}$, which contradicts $\neg h_p{\cdot}h_0[r_p]\mathsf{av}$. We will use this property of r to show $\neg h[r_q]\mathsf{flt}$ and $\neg h[r_q]\mathsf{av}$. Since $h = h_0{\cdot}h_q$ and $\neg h_0[r]\mathsf{flt}$, by the safety monotonicity of r, we have that $\neg h[r]\mathsf{flt}$. Thus, $\neg h[r_q]\mathsf{flt}$ by the definition of prot. For $\neg h[r_q]\mathsf{av}$, we have to show that

$$\neg h[r]\mathsf{av} \wedge \big(h[r]\mathsf{flt} \vee \big(\forall m_q, m_0 \in \mathsf{St}.\ (m_q{\cdot}m_0{=}h \wedge m_q \in q) \ \Rightarrow\ \neg m_0[r]\mathsf{flt}\big)\big).$$

Since r is av-free, it does not output av for any input states. For the second conjunct, consider a splitting $m_q{\cdot}m_0$ of h such that $m_q \in q$. Then, since $h = h_q{\cdot}h_0$, $h_q \in q$ and q is precise, we should have that $m_q = h_q$ and $m_0 = h_0$. Since $\neg h_0[r]\mathsf{flt}$, it follows that $\neg m_0[r]\mathsf{flt}$.

Finally, we prove that every output state of r_q from h is $\mathcal{R}{\otimes}\Delta$-related to some output state set of $\mathsf{lft}(r_p)$ from H. In the proof, we will use $\neg h_0[r]\mathsf{flt}$, which we have shown in the previous paragraph. Consider a state h' such that $h[r_q]h'$. Since $h = h_0{\cdot}h_q$, by the definition of $\mathsf{prot}(r, q)$, we have that $h_0{\cdot}h_q[r]h'$. Since $\neg h_0[r]\mathsf{flt}$, we can apply the frame property of r to this computation, and obtain a substate h'_0 of h' such that $h' = h'_0{\cdot}h_q$. Let L_0 be the finite set that includes all the indirectly accessed locations by the "computation" $h_0{\cdot}h_q[r]h'_0{\cdot}h_q$; L_0 is guaranteed to exist by the finite access property of r. Let L be the location set $\big(L_0 \cup \mathsf{dom}(h_0) \cup \mathsf{dom}(h'_0)\big) - \mathsf{dom}(h_q)$. Since $h_q[\mathcal{R}]H_p$ and $\mathcal{R}$ is admissible, there is a subset H_1 of H_p such that

$$H_1 \subseteq H_p \ \wedge\ H_1[\mathcal{R}]h_q \ \wedge\ \forall h_1 \in H_1.\ h_1 \# L.$$

We will show that $H_1 * \{h'_0\}$ is the required output state set. Since h_q and H_1 are $\mathcal{R}$-related, their h'_0-extensions, $h_q{\cdot}h'_0$ and $H_1 * \{h'_0\}$, have to be $\mathcal{R}{\otimes}\Delta$-related. Thus, it remains to show that $H = H_p * \{h_0\}[\mathsf{lft}(r_p)]H_1 * \{h'_0\}$. Instead of proving this relationship directly, we will prove that

$$H_1 * \{h_0\}[\mathsf{lft}(r_p)]H_1 * \{h'_0\}.$$

Because, then, the definition of $\mathsf{lft}(r_p)$ will ensure that we also have the required computation. For every m in $H_1 * \{h'_0\}$, there is a state $m_1 \in H_1$ such that $m = m_1{\cdot}h'_0$. By the choice of H_1, we have $m_1 \in H_p \wedge m_1 \# L$. Then, there exist splitting $n_1{\cdot}n_2 = m_1$ of m_1 and splitting $o_2{\cdot}o_3 = h_q$ of h_q with the property that $n_1 \# h_q$ and $\mathsf{dom}(n_2) = \mathsf{dom}(o_2)$. Note that $n_1 \# h_q$ implies $n_1 \# L_0$, because $L_0 \subseteq L \cup \mathsf{dom}(h_q)$ and $n_1{\cdot}n_2 \# L$. We obtain a new computation of r_p as follows:

$$\begin{array}{lll}
h_q{\cdot}h_0[r]h_q{\cdot}h'_0 & \Longrightarrow & n_1{\cdot}h_q{\cdot}h_0[r]n_1{\cdot}h_q{\cdot}h'_0 \qquad (\because \text{the finite access property of } r) \\
& \Longrightarrow & n_1{\cdot}o_2{\cdot}o_3{\cdot}h_0[r]n_1{\cdot}o_2{\cdot}o_3{\cdot}h'_0 \qquad (\because h_q = o_2{\cdot}o_3) \\
& \Longrightarrow & n_1{\cdot}n_2{\cdot}o_3{\cdot}h_0[r]n_1{\cdot}n_2{\cdot}o_3{\cdot}h'_0 \quad (\because \text{the contents independence of } r) \\
& \Longrightarrow & n_1{\cdot}n_2{\cdot}h_0[r]n_1{\cdot}n_2{\cdot}h'_0 \qquad (\because \text{the frame property of } r) \\
& \Longrightarrow & m_1{\cdot}h_0[r]m \qquad (\because m_1 = n_1{\cdot}n_2 \wedge m_1{\cdot}h'_0 = m) \\
& \Longrightarrow & m_1{\cdot}h_0[r_p]m \qquad (\because \text{the definition of } \mathsf{prot}(r, p))
\end{array}$$

Note that the input $m_1{\cdot}h_0$ of the obtained computation belongs to the state set $H_1 * \{h_0\}$. We just have shown $H_1 * \{h_0\}[\mathsf{lft}(r_p)]H_1 * \{h'_0\}$. □

Lemma 7. *For all power relations $\mathcal{R}_0, \mathcal{R}_1, \mathcal{R}_2$ and all FLAs r_0, r'_0, r_1, r'_1, if $r'_0[\mathsf{psim}(\mathcal{R}_0, \mathcal{R}_1)]r_0 \wedge r'_1[\mathsf{psim}(\mathcal{R}_1, \mathcal{R}_2)]r_1$, then $\mathsf{seq}(r'_0, r'_1)[\mathsf{psim}(\mathcal{R}_0, \mathcal{R}_2)]\mathsf{seq}(r_0, r_1)$.*

Proof. Let r_0, r_1, r'_0, r'_1 be FLAs such that $r'_0[\mathsf{psim}(\mathcal{R}_0, \mathcal{R}_1)]r_0$ and $r'_1[\mathsf{psim}(\mathcal{R}_1, \mathcal{R}_2)]r_1$. Consider $\mathcal{R}_0$-related state h and state set H such that $\mathsf{lft}(\mathsf{seq}(r_0, r_1))$ does not generate an error from H. We first prove that $\mathsf{seq}(r'_0, r'_1)$ does not generate an error from h:

$$
\begin{array}{l}
\quad \neg H[\mathsf{lft}(\mathsf{seq}(r_0, r_1))]\mathsf{flt} \wedge \neg H[\mathsf{lft}(\mathsf{seq}(r_0, r_1))]\mathsf{av} \\
\Longrightarrow \ (\because \text{the definition of } \mathsf{lft}(\mathsf{seq}(r_0, r_1))) \\
\quad \neg H[\mathsf{lft}(r_0)]\mathsf{flt} \wedge \neg H[\mathsf{lft}(r_0)]\mathsf{av} \\
\quad \wedge \ (\forall H'.\ H[\mathsf{lft}(r_0)]H' \Rightarrow \neg H'[\mathsf{lft}(r_1)]\mathsf{flt} \wedge \neg H'[\mathsf{lft}(r_1)]\mathsf{av}) \\
\Longrightarrow \ (\because h[\mathcal{R}_0]H \wedge r_0'[\mathsf{psim}(\mathcal{R}_0, \mathcal{R}_1)]r_0) \\
\quad \neg h[r_0']\mathsf{flt} \wedge \neg h[r_0']\mathsf{av} \wedge (\forall h'.\ h[r_0']h' \Rightarrow \exists H'.\ h'[\mathcal{R}_1]H' \wedge H[\mathsf{lft}(r_0)]H') \\
\quad \wedge \ (\forall H'.\ H[\mathsf{lft}(r_0)]H' \Rightarrow \neg H'[\mathsf{lft}(r_1)]\mathsf{flt} \wedge \neg H'[\mathsf{lft}(r_1)]\mathsf{av}) \\
\Longrightarrow \\
\quad \neg h[r_0']\mathsf{flt} \wedge \neg h[r_0']\mathsf{av} \\
\quad \wedge \ (\forall h'.\ h[r_0']h' \Rightarrow \exists H'.\ h'[\mathcal{R}_1]H' \wedge \neg H'[\mathsf{lft}(r_1)]\mathsf{flt} \wedge \neg H'[\mathsf{lft}(r_1)]\mathsf{av}) \\
\Longrightarrow \ (\because h'[\mathcal{R}_1]H' \wedge r_1'[\mathsf{psim}(\mathcal{R}_1, \mathcal{R}_2)]r_1) \\
\quad \neg h[r_0']\mathsf{flt} \wedge \neg h[r_0']\mathsf{av} \wedge (\forall h'.\ h[r_0']h' \Rightarrow \neg h'[r_1']\mathsf{flt} \wedge \neg h'[r_1']\mathsf{av}) \\
\Longrightarrow \\
\quad \neg h[\mathsf{seq}(r_0', r_1')]\mathsf{flt} \wedge \neg h[\mathsf{seq}(r_0', r_1')]\mathsf{av}
\end{array}
$$

Next we prove that all the output states of $\mathsf{seq}(r_0', r_1')$ from h are $\mathcal{R}_2$-related to some output state sets of $\mathsf{lft}(\mathsf{seq}(r_0, r_1))$ from H:

$$
\begin{array}{l}
\quad \neg H[\mathsf{lft}(\mathsf{seq}(r_0, r_1))]\mathsf{flt} \wedge \neg H[\mathsf{lft}(\mathsf{seq}(r_0, r_1))]\mathsf{av} \wedge h[\mathsf{seq}(r_0', r_1')]h'' \\
\Longrightarrow \ (\because \text{the definition of } \mathsf{seq}(r_0, r_1)) \\
\quad \neg H[\mathsf{lft}(\mathsf{seq}(r_0, r_1))]\mathsf{flt} \wedge \neg H[\mathsf{lft}(\mathsf{seq}(r_0, r_1))]\mathsf{av} \wedge (\exists h'.\ h[r_0']h' \ \wedge \ h'[r_1']h'') \\
\Longrightarrow \ (\because \text{the definition of } \mathsf{lft}(\mathsf{seq}(r_0, r_1))) \\
\quad \neg H[\mathsf{lft}(r_0)]\mathsf{flt} \wedge \neg H[\mathsf{lft}(r_0)]\mathsf{av} \wedge \neg H[\mathsf{lft}(\mathsf{seq}(r_0, r_1))]\mathsf{flt} \\
\quad \wedge \ \neg H[\mathsf{lft}(\mathsf{seq}(r_0, r_1))]\mathsf{av} \wedge (\exists h'.\ h[r_0']h' \ \wedge \ h'[r_1']h'') \\
\Longrightarrow \ (\because r_0'[\mathsf{psim}(\mathcal{R}_0, \mathcal{R}_1)]r_0 \wedge h[\mathcal{R}_0]H) \\
\quad \exists H', h'.\ H[\mathsf{lft}(r_0)]H' \wedge h'[\mathcal{R}_1]H' \wedge \neg H[\mathsf{lft}(\mathsf{seq}(r_0, r_1))]\mathsf{flt} \\
\qquad\qquad \wedge \neg H[\mathsf{lft}(\mathsf{seq}(r_0, r_1))]\mathsf{av} \wedge h'[r_1']h'' \\
\Longrightarrow \ (\because \text{the definition of } \mathsf{lft}(\mathsf{seq}(r_0, r_1))) \\
\quad \exists H', h'.\ H[\mathsf{lft}(r_0)]H' \wedge h'[\mathcal{R}_1]H' \wedge \neg H'[\mathsf{lft}(r_1)]\mathsf{flt} \wedge \neg H'[\mathsf{lft}(r_1)]\mathsf{av} \wedge h'[r_1']h'' \\
\Longrightarrow \ (\because r_1'[\mathsf{psim}(\mathcal{R}_1, \mathcal{R}_2)]r_1 \wedge h'[\mathcal{R}_1]H') \\
\quad \exists H', H''.\ H[\mathsf{lft}(r_0)]H' \wedge H'[\mathsf{lft}(r_1)]H'' \wedge h''[\mathcal{R}_2]H'' \\
\Longrightarrow \ (\because \text{the definition of } \mathsf{lft}(\mathsf{seq}(r_0, r_1))) \\
\quad \exists H''.\ H[\mathsf{lft}(\mathsf{seq}(r_0, r_1))]H'' \wedge h''[\mathcal{R}_2]H''.
\end{array}
$$

□

Lemma 8. *For all power relations $\mathcal{R}_0, \mathcal{R}_1$, sets I and I-indexed families $\{r_i'\}_{i\in I}$, $\{r_i\}_{i\in I}$ of FLAs, if $\forall i{\in}I.\, r_i'[\mathsf{psim}(\mathcal{R}_0, \mathcal{R}_1)]r_i$, then $\bigcup_{i\in I} r_i'[\mathsf{psim}(\mathcal{R}_0, \mathcal{R}_1)] \bigcup_{i\in I} r_i$.*

Proof. Let $\{r_i\}_{i\in I}$ and $\{r_i'\}_{i\in I}$ be (possibly empty) families of FLAs such that $r_i'[\mathsf{psim}(\mathcal{R}_0, \mathcal{R}_1)]r_i$ for all indices i in I. We need to show that

$$(\bigcup_{i\in I} r_i')[\mathsf{psim}(\mathcal{R}_0, \mathcal{R}_1)](\bigcup_{i\in I} r_i).$$

Consider $\mathcal{R}_0$-related state h and state set H such that $\mathsf{lft}(\bigcup_{i\in I} r_i)$ does not generate an error from H. We first show that $\bigcup_{i\in I} r_i'$ does not generate an error from h:

$$
\begin{array}{l}
\quad \neg H[\mathsf{lft}(\bigcup_{i\in I} r_i)]\mathsf{flt} \ \wedge \ \neg H[\mathsf{lft}(\bigcup_{i\in I} r_i)]\mathsf{av} \\
\Longrightarrow \ (\because \forall j \in I.\, \mathsf{lft}(\bigcup_{i\in I} r_i) \supseteq \mathsf{lft}(r_j)) \\
\quad \forall i \in I.\, \neg H[\mathsf{lft}(r_i)]\mathsf{flt} \ \wedge \ \neg H[\mathsf{lft}(r_i)]\mathsf{av} \\
\Longrightarrow \ (\because h[\mathcal{R}_0]H \wedge \forall i \in I.\, r'_i[\mathsf{psim}(\mathcal{R}_0, \mathcal{R}_1)]r_i) \\
\quad \forall i \in I.\, \neg h[r'_i]\mathsf{flt} \ \wedge \ \neg h[r'_i]\mathsf{av} \\
\Longrightarrow \\
\quad \neg h[\bigcup_{i\in I} r'_i]\mathsf{flt} \ \wedge \ \neg h[\bigcup_{i\in I} r'_i]\mathsf{av}
\end{array}
$$

Next, we prove that all the output states of $\bigcup_{i\in I} r'_i$ from h are $\mathcal{R}_1$-related to some output state sets of $\mathsf{lft}(\bigcup_{i\in I} r_i)$ from H:

$$
\begin{array}{l}
\quad h[\bigcup_{i\in I} r'_i]h' \ \wedge \ \neg H[\mathsf{lft}(\bigcup_{i\in I} r_i)]\mathsf{flt} \ \wedge \ \neg H[\mathsf{lft}(\bigcup_{i\in I} r_i)]\mathsf{av} \\
\Longrightarrow \ (\because \forall j \in I.\, \mathsf{lft}(\bigcup_{i\in I} r_i) \supseteq \mathsf{lft}(r_j)) \\
\quad h[\bigcup_{i\in I} r'_i]h' \ \wedge \ (\forall i \in I.\, \neg H[\mathsf{lft}(r_i)]\mathsf{flt} \ \wedge \ \neg H[\mathsf{lft}(r_i)]\mathsf{av}) \\
\Longrightarrow \\
\quad \exists i \in I.\, h[r'_i]h' \ \wedge \ \neg H[r_i]\mathsf{flt} \ \wedge \ \neg H[r_i]\mathsf{av} \\
\Longrightarrow \ (\because h[\mathcal{R}_0]H \wedge \forall i \in I.\, r'_i[\mathsf{psim}(\mathcal{R}_0, \mathcal{R}_1)]r_i) \\
\quad \exists i \in I.\, \exists H'.\, H[\mathsf{lft}(r_i)]H' \ \wedge \ h'[\mathcal{R}_1]H' \\
\Longrightarrow \ (\because \forall j \in I.\, \mathsf{lft}(\bigcup_{i\in I} r_i) \supseteq \mathsf{lft}(r_j)) \\
\quad \exists H'.\, H[\mathsf{lft}(\bigcup_{i\in I} r_i)]H' \ \wedge \ h'[\mathcal{R}_1]H'.
\end{array}
$$

□

Theorem 1 (Abstraction). *Let $(q, \epsilon), (p, \eta)$ be semantic modules, and $\mathcal{R}$ be an admissible power relation s.t. $\mathcal{R} \subseteq q \times \wp(p)$. If (q, ϵ) power-simulates (p, η) by $\mathcal{R}$, then for all commands C and all environments μ, μ', we have that*

$$(\forall P.\, \mu'(P)[\mathsf{psim}(\mathcal{R}\otimes\Delta, \mathcal{R}\otimes\Delta)]\mu(P)) \ \Rightarrow \ [\![C]\!]_{(q,\epsilon)}\mu'[\mathsf{psim}(\mathcal{R}\otimes\Delta, \mathcal{R}\otimes\Delta)][\![C]\!]_{(p,\eta)}\mu.$$

Proof. We use induction on the structure of C. When C is a module operation f or a procedure name P, the theorem follows from the assumption: (q, ϵ) power-simulates (p, η) by $\mathcal{R}$, and for all P, $\mu'(P)[\mathsf{psim}(\mathcal{R}\otimes\Delta, \mathcal{R}\otimes\Delta)]\mu(P)$. When C is an atomic client operation a, the theorem holds because of Lemma 6. The remaining three cases follow from the closedness of $\mathsf{psim}(\mathcal{R}\otimes\Delta, \mathcal{R}\otimes\Delta)$ in Lemma 7 and 8: $\mathsf{psim}(\mathcal{R}\otimes\Delta, \mathcal{R}\otimes\Delta)$ is closed under arbitrary union and seq. This closedness property directly implies that the induction step goes through for the cases of $C_1[]C_2$ and $C_1; C_2$. For $\mathsf{fix}\, P.C'$, we note that the closedness under arbitrary union implies that $\mathsf{psim}(\mathcal{R}\otimes\Delta, \mathcal{R}\otimes\Delta)$ is complete,[10] and that this completeness is what we need to prove the induction step for $\mathsf{fix}\, P.C'$. □

Lemma 9 (Identity Extension). *A module (q, ϵ) data-refines another module (p, η) iff for all complete commands C, we have that $[\![C]\!]^c_{(q,\epsilon)}[\mathsf{psim}(\mathsf{ID}, \mathsf{ID})][\![C]\!]^c_{(p,\eta)}$.*

Proof. We prove this lemma by transforming $\mathsf{psim}(\mathsf{ID}, \mathsf{ID})$ to an equivalent simpler assertion; unrolling $\mathsf{psim}(\mathsf{ID}, \mathsf{ID})$ in the lemma by this assertion then gives the claimed equivalence. Power relation ID relates state h and state set H iff

[10] $\mathsf{psim}(\mathcal{R}_1, \mathcal{R}_2)$ relates the least FLA to itself, and is chain-complete.

$H = \{h\}$. Therefore, $\mathsf{psim}(\mathsf{ID},\mathsf{ID})$ can be simplified as follows: $r'[\mathsf{psim}(\mathsf{ID},\mathsf{ID})]r$ iff for all states h, if $\mathsf{lft}(r)$ does not generate an error from $\{h\}$, then

$$\big(\neg h[r']\mathsf{flt} \ \wedge \ \neg h[r']\mathsf{av}\big) \ \wedge \ \big(\forall h'.\, h[r']h' \implies \{h\}[\mathsf{lft}(r)]\{h'\}\big).$$

Note that in this simplified assertion, the lifted command $\mathsf{lft}(r)$ is run only for a singleton input set $\{h\}$. For such special inputs, $\mathsf{lft}(r)$ behaves the same as r: $\mathsf{lft}(r)$ does not generate an error from $\{h\}$ iff r does not generate an error from h; $\mathsf{lft}(r)$ can produce $\{h'\}$ from $\{h\}$ iff r can produce h' from h. Thus, the definition of $r'[\mathsf{psim}(\mathsf{ID},\mathsf{ID})]r$ can be further simplified to the following assertion: for all states h, if r does not generate an error from h, then r' does not generate an error and all the output states of r' are also possible outcomes of r. Now, using this assertion, we unroll $\mathsf{psim}(\mathsf{ID},\mathsf{ID})$ in the lemma. The resulting unrolled statement proves the lemma, because both sides of "iff" in the statement are the same. □

Theorem 2 (Soundness). *If a module (q,ϵ) power-simulates another module (p,η) by an admissible power relation $\mathcal{R} \subseteq q \times \wp(p)$, then (q,ϵ) data-refines (p,η).*

Proof. Suppose that a module (q,ϵ) power-simulates another module (p,η) by an admissible power relation $\mathcal{R} \subseteq q \times \wp(p)$. We will show that for all complete commands C, $[\![C]\!]_{(q,\epsilon)}[\mathsf{psim}(\mathsf{ID},\mathsf{ID})][\![C]\!]_{(p,\eta)}$, because, then, module (q,ϵ) should data-refine (p,η) (Lemma 9). Pick an arbitrary complete program C. Let μ be an environment that maps all program identifiers to the empty relation. By Lemma 8, $\mu(P)[\mathsf{psim}(\mathcal{R}\otimes\Delta,\mathcal{R}\otimes\Delta)]\mu(P)$ for all P in pid. From this, we derive the required relationship as follows:

$\forall P \in \mathsf{pid}.\ \mu(P)[\mathsf{psim}(\mathcal{R}\otimes\Delta,\mathcal{R}\otimes\Delta)]\mu(P)$
$\implies$ ($\because$ Theorem 1)
$[\![C]\!]_{(q,\epsilon)}\mu[\mathsf{psim}(\mathcal{R}\otimes\Delta,\mathcal{R}\otimes\Delta)][\![C]\!]_{(p,\eta)}\mu$
$\implies$ ($\because$ Lemma 7)
$\mathsf{seq}(\mathsf{seq}(\epsilon(\mathsf{init}),[\![C]\!]_{(q,\epsilon)}\mu),\epsilon(\mathsf{final}))[\mathsf{psim}(\mathsf{ID},\mathsf{ID})]\mathsf{seq}(\mathsf{seq}(\eta(\mathsf{init}),[\![C]\!]_{(p,\eta)}\mu)),\eta(\mathsf{final}))$
$\implies$ ($\because$ the definition of $[\![-]\!]^c$)
$[\![C]\!]^c_{(q,\epsilon)}[\mathsf{psim}(\mathsf{ID},\mathsf{ID})][\![C]\!]^c_{(p,\eta)}$

□

6 State-based Representation of Power Simulation

Our soundness result is rather technical, and does not provide a computational intuition about why power simulation is sound. Giving one such intuition is the goal of this section. We will consider a special case of power simulation where the coupling *power* relation arises from a standard coupling relation on *states*, and show that in such a case, the power simulation is forward simulation modified with backtracking; thus, it is this backtracking that makes power simulation sound. Throughout the section, we assume fixed modules (p,η) and (q,ϵ), and consider the power simulation of (p,η) by (q,ϵ).

A standard coupling relation $R \subseteq q \times p$ can generate an admissible power relation $\mathcal{R} \subseteq q \times \wp(p)$, when it is given two additional data: an equivalence relation E on p, and an operator rs, which imposes certain restrictions on relations on states. Equivalence relation E denotes which "abstract" states can be considered the same. The other data, rs, restricts a relation S, using pre-given finite location set L and state h; intuitively, it imposes an additional requirement on S using L and h. We will require that for all L and h, the restriction $\mathsf{rs}(S, L, h)$ of relation S satisfies (1) $(\forall L'.\, \mathsf{rs}(S, L \cup L', h) \subseteq \mathsf{rs}(S, L, h))$, and (2) $(\forall h', h'_0.\, h'[\mathsf{rs}(S, L, h)]h'_0 \Rightarrow h'[S]h'_0 \wedge h'_0 \# (L - \mathsf{dom}(h)))$. From (R, E, rs), we define a power relation as follows:

$$h'[\mathsf{pw}(R, E, \mathsf{rs})]H \stackrel{def}{\Leftrightarrow} \exists h \in \mathsf{St}. \exists L \subseteq_{fin} \mathsf{Loc}.\, h'[R]h \wedge H = \{h_0 \mid h[\mathsf{rs}(E, L, h')]h_0\}.$$

This definition means that H is obtained from h' in three steps: first, to find some h such that $h'[R]h$; then, to collect all the states that are E-equivalent to h; finally, to extract the states among the collected ones that "satisfy" the additional requirement in $\mathsf{rs}(E, L, h')$. The last extracting step is crucial in this construction; it ensures that the constructed relation is admissible.

Lemma 10. *If* $(\forall h, H. h[\mathsf{pw}(R, E, \mathsf{rs})]H \Rightarrow H \neq \emptyset)$, *then* $\mathsf{pw}(R, E, \mathsf{rs})$ *is admissible.*

Proof. In order to prove that $\mathsf{pw}(R, E, \mathsf{rs})$ is admissible, we need to prove that for all states h' and state sets H such that $h'[\mathsf{pw}(R, E, \mathsf{rs})]H$, the following two conditions hold:

- $H \neq \emptyset$; and
- for all finite sets L of locations, there exists a subset H_1 of H such that

$$H_1 \neq \emptyset \ \wedge \ h'[\mathsf{pw}(R, E, \mathsf{rs})]H_1 \ \wedge \ \forall h_1 \in H_1.\, h_1 \# (L - \mathsf{dom}(h')).$$

Consider $\mathsf{pw}(R, E, \mathsf{rs})$-related state h' and state set H. By the definition of $\mathsf{pw}(R, E, \mathsf{rs})$, there exist $L \subseteq_{fin} \mathsf{Loc}$ and $h \in \mathsf{St}$ such that

$$h'[R]h \ \wedge \ H = \{h_0 \mid h[\mathsf{rs}(E, L, h')]h_0\}.$$

The first condition of the admissibility follows from the assumption of this lemma. To prove the second condition of the admissibility, consider a finite set L_1 of locations. The second condition requires a subset of H with certain properties. We show that the following H_1 is such a subset.

$$H_1 = \{h_1 \mid h[\mathsf{rs}(E, L \cup L_1, h')]h_1\}.$$

Set H_1 is included in H, because $\mathsf{rs}(E, -, h')$ relates more states as its second parameter gets smaller. Moreover, by the definition of pw, state h' is $\mathsf{pw}(R, E, \mathsf{rs})$-related to set H_1. Note that by the assumption of the lemma, this relationship also ensures that H_1 is not empty. Finally, for every state h_1 in H_1, we have $h[\mathsf{rs}(E, L \cup L_1, h')]h_1$, and so, $h \# (L \cup L_1 - \mathsf{dom}(h_1))$. Hence, $h \# (L_1 - \mathsf{dom}(h_1))$.

□

Suppose that $\mathsf{pw}(R, E, \mathsf{rs}) \otimes \Delta = \mathsf{pw}(R * \Delta, E * \Delta, \mathsf{rs})$. Under this supposition, we give the state-based representation of $\mathsf{psim}(\mathsf{pw}(R, E, \mathsf{rs}) \otimes \Delta, \mathsf{pw}(R, E, \mathsf{rs}) \otimes \Delta)$.

Proposition 1. *We have that* $r'[\mathsf{psim}(\mathsf{pw}(R, E, \mathsf{rs}) \otimes \Delta, \mathsf{pw}(R, E, \mathsf{rs}) \otimes \Delta)]r$*, iff for all* h', h, L_0*, if* $h'[R * \Delta]h \wedge (\forall h_0.\ h[\mathsf{rs}(E * \Delta, L_0, h')]h_0 \Rightarrow \neg h_0[r]\mathsf{av} \wedge \neg h_0[r]\mathsf{flt})$*, then*

1. $\neg h'[r']\mathsf{flt} \wedge \neg h'[r']\mathsf{av}$*, and*
2. *for all output states* m' *of* r' *from* h' *(i.e.,* $h'[r']m'$*), there exist* m, L_1 *s.t.*

$$m'[R * \Delta]m \wedge (\forall m_0.\ m[\mathsf{rs}(E * \Delta, L_1, m')]m_0 \Rightarrow \exists h_0.\ h[\mathsf{rs}(E * \Delta, L_0, h')]h_0 \wedge h_0[r]m_0).$$

The main message of the state-based characterization lies in the second condition, which is about the output states of the concrete "command" r'. The condition means that every such output state m' from the given concrete input h' should be $R * \Delta$-related to some "backtrackable output" m of the abstract r from h: for some L_1, abstract command r can *backtrack* every $\mathsf{rs}(E * \Delta, L_1, m')$-equivalent state m_0 of m, to some $\mathsf{rs}(E * \Delta, L_0, h')$-equivalent state h_0 of h. Thus, the condition mimics the usual tracking requirement in forward simulation, but it requires that every normal concrete computation should be tracked by some imaginary "backtrackable abstract computation," instead of a normal abstract computation.

Proof (Proposition 1). We will prove two facts about r and r', which together imply the proposition. Note that, when both the definition γ_0 of psim and its new characterization γ_1 given here are viewed syntactically, they are universally quantified formulas whose bodies are $\varphi_0 \Rightarrow \psi_0 \wedge \psi_0'$ and $\varphi_1 \Rightarrow \psi_1 \wedge \psi_1'$, respectively. Consider the splittings of γ_i into $\alpha_i = \forall ... \varphi_i \Rightarrow \psi_i$ and $\beta_i = \forall ... \varphi_i \Rightarrow \psi_i'$ where the universal quantifications are precisely the ones in the original formula γ_i. Then, the original formula is equivalent to the conjunction of the split pieces. The first fact about r and r', which we will prove shortly, expresses the equivalence between the first pieces α_0, α_1 of the two splittings, and the second fact the equivalence between the second parts β_0, β_1 of the splittings. Thus, these two facts together give the equivalence between γ_0 and γ_1 as required. We prove the first fact below:

$$
\begin{array}{l}
\forall h', H.\ (h'[\mathsf{pw}(R, E, \mathsf{rs}) \otimes \Delta]H \wedge \neg H[\mathsf{lft}(r)]\mathsf{av} \wedge \neg H[\mathsf{lft}(r)]\mathsf{flt}) \\
\qquad \Rightarrow (\neg h'[r']\mathsf{av} \wedge \neg h'[r']\mathsf{flt}) \\
\Longleftrightarrow\ (\because \mathsf{pw}(R, E, \mathsf{rs}) \otimes \Delta = \mathsf{pw}(R * \Delta, E * \Delta, \mathsf{rs})) \\
\forall h', H.\ (h'[\mathsf{pw}(R * \Delta, E * \Delta, \mathsf{rs})]H \wedge \neg H[\mathsf{lft}(r)]\mathsf{av} \wedge \neg H[\mathsf{lft}(r)]\mathsf{flt}) \\
\qquad \Rightarrow (\neg h'[r']\mathsf{av} \wedge \neg h'[r']\mathsf{flt}) \\
\Longleftrightarrow\ (\because \text{the definition of } \mathsf{pw}(R * \Delta, E * \Delta, \mathsf{rs})) \\
\forall h', H. \\
\quad (\exists h, L_0.\ h'[R * \Delta]h \wedge H = \{h_0 \mid h[\mathsf{rs}(E * \Delta, L_0, h')]h_0\} \wedge \neg H[\mathsf{lft}(r)]\mathsf{av} \wedge \neg H[\mathsf{lft}(r)]\mathsf{flt}) \\
\quad \Rightarrow (\neg h'[r']\mathsf{av} \wedge \neg h'[r']\mathsf{flt}) \\
\Longleftrightarrow \\
\forall h', H, h, L_0. \\
\quad (h'[R * \Delta]h \wedge H = \{h_0 \mid h[\mathsf{rs}(E * \Delta, L_0, h')]h_0\} \wedge \neg H[\mathsf{lft}(r)]\mathsf{av} \wedge \neg H[\mathsf{lft}(r)]\mathsf{flt}) \\
\quad \Rightarrow (\neg h'[r']\mathsf{av} \wedge \neg h'[r']\mathsf{flt})
\end{array}
$$

$$
\begin{array}{l}
\Longleftrightarrow \\
\quad \forall h', h, L_0. \\
\quad \left(\begin{array}{l} h'[R{*}\Delta]h \wedge \neg\{h_0 \mid h[\mathsf{rs}(E{*}\Delta, L_0, h')]h_0\}[\mathsf{lft}(r)]\mathsf{av} \\ \wedge \neg\{h_0 \mid h[\mathsf{rs}(E{*}\Delta, L_0, h')]h_0\}[\mathsf{lft}(r)]\mathsf{flt} \end{array}\right) \Rightarrow (\neg h'[r']\mathsf{av} \wedge \neg h'[r']\mathsf{flt}) \\
\Longleftrightarrow \ (\because \text{the definition of } \mathsf{lft}(r)) \\
\quad \forall h', h, L_0. \\
\quad \big(h'[R{*}\Delta]h \wedge (\forall h_0.\, h[\mathsf{rs}(E{*}\Delta, L_0, h')]h_0 \Rightarrow \neg h_0[r]\mathsf{av} \wedge \neg h_0[r]\mathsf{flt})\big) \\
\quad \Rightarrow (\neg h'[r']\mathsf{av} \wedge \neg h'[r']\mathsf{flt})
\end{array}
$$

The second fact can be proved as follows:

$$
\begin{array}{l}
\quad \forall h', H, m'.\ \big(h'[\mathsf{pw}(R, E, \mathsf{rs})\otimes\Delta]H \wedge \neg H[\mathsf{lft}(r)]\mathsf{av} \wedge \neg H[\mathsf{lft}(r)]\mathsf{flt} \wedge h'[r']m'\big) \\
\qquad\qquad \Rightarrow (\exists M.\ m'[\mathsf{pw}(R, E, \mathsf{rs})\otimes\Delta]M \wedge H[\mathsf{lft}(r)]M) \\
\Longleftrightarrow \ (\because \mathsf{pw}(R, E, \mathsf{rs})\otimes\Delta = \mathsf{pw}(R{*}\Delta, E{*}\Delta, \mathsf{rs})) \\
\quad \forall h', H, m'.\ \big(h'[\mathsf{pw}(R{*}\Delta, E{*}\Delta, \mathsf{rs})]H \wedge \neg H[\mathsf{lft}(r)]\mathsf{av} \wedge \neg H[\mathsf{lft}(r)]\mathsf{flt} \wedge h'[r']m'\big) \\
\qquad\qquad \Rightarrow (\exists M.\ m'[\mathsf{pw}(R{*}\Delta, E{*}\Delta, \mathsf{rs})]M \wedge H[\mathsf{lft}(r)]M) \\
\Longleftrightarrow \ (\because \text{the definition of } \mathsf{pw}(R{*}\Delta, E{*}\Delta, \mathsf{rs})) \\
\quad \forall h', H, m'. \\
\quad \left(\begin{array}{l} \exists h, L_0.\ h'[R{*}\Delta]h \wedge H{=}\{h_0 \mid h[\mathsf{rs}(E{*}\Delta, L_0, h')]h_0\} \\ \qquad \wedge \neg H[\mathsf{lft}(r)]\mathsf{av} \wedge \neg H[\mathsf{lft}(r)]\mathsf{flt} \wedge h'[r']m' \end{array}\right) \\
\quad \Rightarrow (\exists M, m, L_1.\ m'[R{*}\Delta]m \wedge M{=}\{m_0 \mid m[\mathsf{rs}(E{*}\Delta, L_1, m')]m_0\} \wedge H[\mathsf{lft}(r)]M) \\
\Longleftrightarrow \\
\quad \forall h', H, m', h, L_0. \\
\quad \left(\begin{array}{l} h'[R{*}\Delta]h \wedge H{=}\{h_0 \mid h[\mathsf{rs}(E{*}\Delta, L_0, h')]h_0\} \\ \wedge \neg H[\mathsf{lft}(r)]\mathsf{av} \wedge \neg H[\mathsf{lft}(r)]\mathsf{flt} \wedge h'[r']m' \end{array}\right) \\
\quad \Rightarrow (\exists M, m, L_1.\ m'[R{*}\Delta]m \wedge M{=}\{m_0 \mid m[\mathsf{rs}(E{*}\Delta, L_1, m')]m_0\} \wedge H[\mathsf{lft}(r)]M) \\
\Longleftrightarrow \\
\quad \forall h', m', h, L_0. \\
\quad \left(\begin{array}{l} h'[R{*}\Delta]h \wedge h'[r']m' \wedge \neg\{h_0 \mid h[\mathsf{rs}(E{*}\Delta, L_0, h')]h\}[\mathsf{lft}(r)]\mathsf{av} \\ \wedge \neg\{h_0 \mid h[\mathsf{rs}(E{*}\Delta, L_0, h')]h\}[\mathsf{lft}(r)]\mathsf{av} \end{array}\right) \\
\quad \Rightarrow \left(\begin{array}{l} \exists m, L_1.\ \{h_0 \mid h[\mathsf{rs}(E{*}\Delta, L_0, h')]h_0\}[\mathsf{lft}(r)]\{m_0 \mid m[\mathsf{rs}(E{*}\Delta, L_1, m')]m_0\} \\ \qquad\qquad \wedge m'[R{*}\Delta]m \end{array}\right) \\
\Longleftrightarrow \ (\because \text{the definition of } \mathsf{lft}(r)) \\
\quad \forall h', m', h, L_0. \\
\quad \big(h'[R{*}\Delta]h \wedge h'[r']m' \wedge \big(\forall h_0.\, h[\mathsf{rs}(E{*}\Delta, L_0, h')]h_0 \Rightarrow \neg h_0[r]\mathsf{av} \wedge \neg h_0[r]\mathsf{flt}\big)\big) \\
\quad \Rightarrow \left(\begin{array}{l} \exists m, L_1.\ m'[R{*}\Delta]m \\ \quad \wedge \forall m_0.\, m[\mathsf{rs}(E{*}\Delta, L_1, m')]m_0 \Rightarrow \exists h_0.\, h[\mathsf{rs}(E{*}\Delta, L_0, h')]h_0 \wedge h_0[r]m_0 \end{array}\right) \\
\Longleftrightarrow \\
\quad \forall h', h, L_0. \\
\quad \big(h'[R{*}\Delta]h \wedge \big(\forall h_0.\, h[\mathsf{rs}(E{*}\Delta, L_0, h')]h_0 \Rightarrow \neg h_0[r]\mathsf{av} \wedge \neg h_0[r]\mathsf{flt}\big)\big) \\
\quad \Rightarrow \\
\quad \left(\begin{array}{l} \forall m'.\, h'[r']m' \Rightarrow \\ \quad \left(\begin{array}{l} \exists m, L_1. \\ \quad m'[R{*}\Delta]m \\ \quad \wedge \forall m_0.\, m[\mathsf{rs}(E{*}\Delta, L_1, m')]m_0 \Rightarrow \exists h_0.\, h[\mathsf{rs}(E{*}\Delta, L_0, h')]h_0 \wedge h_0[r]m_0 \end{array}\right) \end{array}\right)
\end{array}
$$

□

7 Conclusion

In this paper, we have proposed a new data-refinement method, called power simulation, for programs with low-level pointer operations, and provided a non-trivial soundness proof of the method.

The very idea of relating a state to a state set in power simulation comes from Reddy's method for data refinement [15]. In order to have a single complete data-refinement method for a language *without pointers*, he lifted forward simulation such that all the components of the simulation become about state sets, instead of states. However, the details of the two methods, such as the admissibility condition for coupling relations and the lifting operator for commands, are completely different.

References

1. A. Banerjee and D. A. Naumann. Representation independence, confinement and access control (extended abstract). In *POPL'02*, pages 166–177. ACM, 2002.
2. C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *POPL'03*, pages 213–223. ACM, 2003.
3. D. G. Clarke, J. Noble, and J. M. Potter. Simple ownership types for object containment. In *ECOOP'01*, volume 2072 of *Lecture Notes in Computer Science*, pages 53–76. Springer-Verlag, 2001.
4. W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison.* Number 47 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, UK, 1998.
5. E. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, 1976.
6. J. He, C. A. R. Hoare, and J. W. Sanders. Data refinement refined. In *ESOP'86*, volume 213 of *Lecture Notes in Computer Science*, pages 187–196. Springer-Verlag, 1986.
7. C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.
8. J. Hogg. Islands: Aliasing protection in object-oriented languages. In *OOPLA'91*, pages 271–285. ACM, 1991.
9. J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt. The geneva convention on the treatment of object aliasing. *OOPS Messenger*, 3(2):11–16, 1992.
10. S. Istiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL'01*, pages 14–26, London, 2001. ACM.
11. K. R. M. Leino and G. Nelson. Data abstraction and information hiding. *ACM Trans. on Program. Lang. and Syst.*, 24(5):491–553, 2002.
12. I. Mijajlovic, N. Torp-Smith, and P. O'Hearn. Refinement and separation contexts. In *FSTTCS'04*, volume 3328 of *Lecture Notes in Computer Science*, pages 421–433. Springer-Verlag, 2004.
13. D. Naumann and M. Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state. In *LICS'04*, pages 313–323. IEEE, 2004.
14. P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *POPL'04*, pages 268–280, Venice, 2004. ACM.
15. U. S. Reddy. Talk at MFPS'00, Hokoken, New Jersey, USA, 2000.

16. U. S. Reddy and H. Yang. Correctness of data representations involving heap data structures. *Science of Computer Programming*, 50(1):257–305, 2004.
17. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS'02*, volume 17, pages 55 – 74, Copenhagen, 2002. IEEE.
18. I. Stark. Categorical models for local names. *Lisp and Symbolic Comput.*, 9(1):77–107, 1996.
19. H. Yang. *Local Reasoning for Stateful Programs*. PhD thesis, University of Illinois at Urbana-Champaign, 2001. (Technical Report UIUCDCS-R-2001-2227).
20. H. Yang and P. W. O'Hearn. A semantic basis for local reasoning. In *FSTTCS'02*, volume 2303 of *Lecture Notes in Computer Science*, pages 402–416. Springer-Verlag, 2002.

A Counter example for the Soundness of the Forward Simulation

Suppose that the module interface mop is $\{\mathsf{init}, \mathsf{final}, f\}$. Let $(p, \eta), (q, \epsilon)$ be semantic modules for mop defined as follows:

$$\begin{array}{rcl} h \in p & \stackrel{def}{\Leftrightarrow} & h = [1{\to}2] \\ h[\eta(\mathsf{init})]v & \stackrel{def}{\Leftrightarrow} & 1 \notin \mathsf{dom}(h) \ \wedge \ v = h{\cdot}[1{\to}2] \\ h[\eta(f)]v & \stackrel{def}{\Leftrightarrow} & v = h \\ h[\eta(\mathsf{final})]v & \stackrel{def}{\Leftrightarrow} & \mathsf{if}\ (1 \notin h)\ \mathsf{then}\ (v = \mathsf{flt})\ \mathsf{else}\ (\exists i{\in}\mathsf{Int}.\, v{\cdot}[1{\to}i] = h) \end{array}$$

$$\begin{array}{rcl} h \in q & \stackrel{def}{\Leftrightarrow} & h = [] \\ h[\epsilon(\mathsf{init})]v & \stackrel{def}{\Leftrightarrow} & 1 \notin \mathsf{dom}(h) \ \wedge \ v = h \\ h[\epsilon(f)]v & \stackrel{def}{\Leftrightarrow} & v = h \\ h[\epsilon(\mathsf{final})]v & \stackrel{def}{\Leftrightarrow} & v = h \end{array}$$

Module (q, ϵ) does not data-refine (p, η). To see the reason, consider a complete command that consists of a single atomic operation $\mathsf{cons}(2, 1)$ in Sec. 4. When this complete command $\mathsf{cons}(2, 1)$ is run with (q, ϵ) from $[2{\to}0]$, it can output $[1{\to}0, 2{\to}1]$:

$$[2{\to}0]\left[[\![\mathsf{cons}(2,1)]\!]^{c}_{(q,\epsilon)}\right][1{\to}0, 2{\to}1].$$

However, if the command is run with the other module (p, η) from $[2{\to}0]$, it cannot produce the same output state; all of its output states have the form of $[2{\to}n', n'{\to}0]$ for some n' different from 1 and 2, because the initialization $\eta(\mathsf{init})$ takes cell 1 before $\mathsf{cons}(2, 1)$. Since the command with (p, η) does not generate an error from the input $[2{\to}0]$, this failure of producing the same output shows that module (q, ϵ) does not data-refine (p, η).

However, the forward simulation incorrectly claims the opposite. It is because, when R is a relation defined by

$$h'[R]h \ \stackrel{def}{\Leftrightarrow} \ h' = [] \wedge h = [1{\to}2],$$

module (q, ϵ) forward-simulates (p, η) by R.