Separation and Data Refinement

Ivana Mijajlović

Submitted for the degree of Doctor of Philosophy

Queen Mary, University of London

2007

Declaration

I hereby declare that this thesis

- is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text;
- the work in this thesis is not substantially the same as any that I have submitted for a degree or diploma or other qualification at any other university.

Ivana Mijajlović

22 October 2007

Author: Ivana Mijajlović (1975-)

Title of the PhD Thesis: Separation and Data Refinement

Keywords: data refinement, semantics of programming languages, separation logic, separation context, forward simulation, lifting theorem, data abstraction, power simulation

Area: data refinement, formal methods, separation logic

Advisor: Peter O'Hearn

PhD Committee: Uday Reddy, Ian Mackie

Faculty: Queen Mary, University of London

Date: June 8 2007

Language: English

References: 81

Pages: 157

Abstract

The thesis develops two sound methods for proving data refinement in the presence of the low level pointer operations. Central to the methods is a notion of a *separation context*, which ensures that a simulation between two modules can be lifted to the whole programming language. Such a "lifting theorem" is not true for arbitrary programming language contexts, in the presence of pointers.

Separation contexts are based on a demand that a client program in an interaction with a module, while respecting constraints imposed by its precondition, also complies with the requirement not to interfere with the module's internal representation directly, but only through the provided module interface. The concept of a separation context is introduced semantically, but a more pragmatic way of checking whether a certain program is a separation context is also provided. Namely, from the fact that a program satisfies certain specification in an environment which contains a module, it can be inferred that the program is then a separation context for the relevant precondition and assumed module. The proof that a program meets given specification can be conducted in Separation logic.

The forward simulation method requires fixing a binary relation between the states of concrete and abstract modules and proving that the relation is preserved by the corresponding pairs of abstract and concrete operations. The lifting and the soundness results depend on a restriction on binary relations to be only the growing ones, where the abstract state is in a certain sense smaller than the concrete one.

The power simulation method is a more general method than forward simulation. It works in a similar fashion to forward simulation, but involves power relations, relations between a state and a state set, rather than the ordinary relations between pairs of states. The lifting and the soundness theorems for the method are the ultimate results, which we achieve by setting a requirement that all the involved power relations are admissible, in a sense to be defined.

In some special, but still very important cases, the power simulation method has a state-based representation. The thesis also explores this aspect of power simulation method and its relation to the forward simulation method.

Acknowledgements

I owe a great debt of gratitude to many people who have helped me complete the research for my thesis. Above all, I am deeply grateful to my supervisor and friend Peter O'Hearn, who has given me the right guidance throughout the course of my studies. His kindness and patience encouraged me greatly in going through this difficult task. Peter is a genuinely gifted scientist and a wise teacher, and it is very difficult to stay unaffected by his infectious enthusiasm. I just hope I have learned from him enough to become a better person and a good scientist.

I would like to express my gratitude to Noah Torp-Smith and Hongseok Yang who, apart from Peter, I have worked with on some of the research presented in this thesis [52, 53]. They have been great collaborators and friends and I hope I will have a chance to work with them in the future.

This research was financially supported by Queen Mary, University of London, and I am very grateful for that.

I owe special thanks to my chums Josh Berdine and Dino Distefano. They have been there for me in the best and the worst, giving me support both as colleagues and as friends. I am very proud to have a privilege to work with them and to be their friend.

My studies would not have been as exciting and enlightening without the remaining members of the East London Massive: Richard Bornat who is always full of ideas and equally excited about each one of them, Cristiano Calcagno with his quiet sharpness, Matthew Parkinson with the true talent and Aziem Chawdhary, a promising future computer scientist. The East London Massive has provided many, many inspiring and stimulating discussions on our regular weekly meetings.

Among others, I owe a very special thanks to John Reynolds, Uday Reddy, Nick Benton, Philippa Gardner, Uri Zarfaty, Lars Birkedal and Jeff Sanders for providing inspirational discussions, giving invaluable advice and letting me learn from them.

I would like to thank Max Kanovich and Gianluigi Bellin who have shown a lot of good will and patience for us, PhD students at Queen Mary, in an effort to broaden our general computer science knowledge in the reading group meetings they regularly held. I am also grateful to my fellow PhD students, especially Corrado Biasi, Kurt Renelter, Mike Samuels, Olga Lightfoot, Alessio Del Bue and Jose Galan and all the staff at the Department of Computer Science of Queen Mary, who have made my studies enjoyable and amusing.

I am in great debt to all of my friends here in UK and in my homeland Serbia, especially Vesna Damnjanović who has proved to be as good friend as one can get, many times during the long course of (almost) twenty years.

I am deeply grateful to my parents Svetlana and Žarko who, working in related fields and being great parents, gave me exactly the right support and believed in me from the very beginning. I am equally grateful to my brother Aleksandar and his family for all the love and support.

Finally, there are no words to express how grateful I am to my husband and my best friend Zlatko Filipović, for giving all it takes in showing his love, care and support.

Contents

List of Figures

List of Tables

1

The Introduction

A computer program often implements an abstract idea. For example, searching and sorting algorithms originate from operations on sequences and sets – mathematical objects, which are conveniently represented as arrays or lists for the manipulation by a computer. Graph algorithms, used for solving a diversity of problems, from finding the shortest path from place A to place B to problems of scheduling or security – execute on some representation of a mathematical graph. A variety of such examples can be found in any book on data structures and algorithms, for instance [50, 28]. Data refinement examines the relationship between the abstract, mathematical objects and their representations. In particular, using the techniques of data refinement, one can derive a concrete representation of an abstract data type, or prove that a given concrete object is indeed a representation of some abstract object. For the pairs of objects for which a refinement relationship is established, the concrete object always behaves at least as well as the abstract one, and it can replace it in any computation.

The early ideas of data refinement come from Wirth [79] and Hoare in his paper on the correctness of data representations [35], where also an early suggestion for the method for proving data refinement was illustrated. The method was then used and developed in the VDM technique of data refinement [42] and generalized in a paper by Hoare et al [33]. While it has become very influential, certain limitations to the "traditional" method have been recognized. In this introductory chapter we shed some light on the problems with the traditional method and give an overview of what we have done in order to solve them. We then discuss work related to ours and finally give an outline of the rest of the thesis.

1.1 The Traditional Method

In his paper on proving correctness of data representations [35], Hoare gave the first ideas of data refinement and an informal account of how to prove it. The gist of the method is defining the "abstract space" on which the abstract program operates, and the "concrete space" on which the concrete program executes, and finding a suitable relationship between the two. This relationship is usually accompanied by an invariant condition, which must be respected by the operations; i.e., having that the condition holds before the execution of the operation, it also holds upon its completion. In the original paper [35], the method is illustrated by an example, which we revisit here. The example examines the relationship between the mathematical representation of a set of at most hundred integers and its more concrete representation – an array also declared to have at most a hundred elements. The abstract specification provides operations for insertion, removal and testing whether a particular element is in the set. The concrete representation needs to supply the client program with the same services, so it needs to have the same operations, only implemented to work on an array instead of a set. The concrete implementation is given in Table 1.1. We can view the array together with the operations as one entity, and call it a *module*. To change a value of the set, the client program interacting with the module should do so only through the operations provided by the module. For instance, if the client program interacts with the concrete implementation, then it is allowed to read and change the values in the array only through the operations that the concrete module provides.

We now prove, somewhat informally, that the array representation simulates the abstract specification of data type Set given in Table 1.2, by defining a relation between the two and showing that each of the concrete operations mimics the behavior of the corresponding abstract one. The abstract specification can be given using mathematical language. The definition of the abstract data structure assumes that the current value of the set is held in a set-variable s. The parameter of the operations, *n*, denotes an integer.

First, we define a refinement relation between the two representations. The relation

Table 1.1: Concrete implementation of the data type Set

```
int a[100]; int m;  /* m denotes the actual  size  of the array */

void insert(int i);
   {int j;
   for (j = 0; j < m && a[j]! = i; j++);
    if (j == m) then
        {a[m] = i; m = m + 1
    else skip}

void remove(int i);
   {int j;
   for (j = 0; j < m && a[j]! = i; j++);
    if (j < m) then
      {for (int k = j + 1; k < m; k++)
          a[k − 1] = a[k]; m = m − 1}
    else skip}

int has(int i);
   {for (int j = 0; j < m && a[j]! = i; j++;);
    if (j < m) then return 1
    else return 0}
```

between two different implementations connects pairs of program states, the concrete state and the abstract state. A state assigns values to variables. We designate state by σ. We say that two states σ_1 and σ_2 are related if the abstract one contains a set s and the concrete one contains an array a, such that they hold exactly the same elements, while imposing on all other variables to have identical values.

$$\sigma_1[R]\sigma_2 \iff \{\sigma_1(a[i]) \mid 0 \leq i < m\} = \sigma_2(s) \text{ and for all other variables x } \sigma_1(x) = \sigma_2(x).$$

For each concrete operation we need to prove that when we run the operation on a state σ_1 that contains an array a which is related to some state σ_2 that contains set s, it produces an output state which is related to the output state of the corresponding abstract operation when executed on state σ_2. So, if we have $\sigma_1[R]\sigma_2$, and run the $\mathsf{insert}(\mathsf{n})$ operation on σ_1, there are two possible outcomes. One possibility is that n is already in the array. In that case we get the same array with which we started. Since the initial states are related by R, n is in the set s too, and so the $s \cdot \mathsf{insert}(n)$ returns the state containing the initial set s, just as in the concrete case. Therefore the output states are related as required. The other possibility is that n is not already in the array; it is not in the set s either because the states containing them are related by R. After running $\mathsf{insert}(\mathsf{n})$ on the array, the set of values held in the output array is going to be $\{a[i]) \mid 0 \leq i < m\} \cup \{a[m] | a[m] = n\}$. This is exactly the value of the set s in the output state of $s \cdot \mathsf{insert}(n)$, and therefore the output states are again related by relation R, bearing in mind that no other variables have been changed. We can similarly argue that the operations $\mathsf{remove}(n)$ and $\mathsf{has}(n)$ satisfy the relation preservation requirements.

The traditional method presupposes a static separation between the variables of the module and the variables of the *client* – a program that initiates the calls to the operations of the module. It can be easily seen that a client program is allowed to make changes to the module representing a set only through the provided operations. All the variables involved in the representation of the data type Set are local to this representation and the client has no knowledge of them. Consequently, the client cannot access the internal representation of the module directly, through variables, but only through the module operations. It is very important for this requirement to be met, because otherwise the client program might change directly the representation of the module, not respecting the conditions necessary for module's proper functioning. If we allow the client to access the module's part of the state directly, it becomes sensitive to the changes in the representation of the abstract specification. For instance, if there was no separation between the client program variables and the module variables, in our example of data type Set and its array representation, the client could access directly the internal representation of the module; it could directly read from or write to the individual elements of the array which represents the set. One possible consequence of that would be, if the client program changed the contents of one of the elements of the array, the array would no longer be a representation of the intended abstract set. Or, if the client program reads directly the value of an element of the array expecting one representation of the data type set, its behavior might be completely different if another representation was supplied; for instance, if an ascending ordered array was supplied instead of a descending ordered array.

Table 1.2: Abstract specification of the data type Set

$\mathsf{s} \cdot \mathsf{insert}(\mathsf{n}) = \textbf{if } \mathsf{n} \notin \mathsf{s} \textbf{ then } \mathsf{s} = \mathsf{s} \cup \{\mathsf{n}\} \textbf{ else } \mathsf{skip}.$

$\mathsf{s} \cdot \mathsf{remove}(\mathsf{n}) = \textbf{if } \mathsf{n} \in \mathsf{s} \textbf{ then } \mathsf{s} = \mathsf{s} \setminus \{\mathsf{n}\} \textbf{ else } \mathsf{skip}.$

$\mathsf{s} \cdot \mathsf{has}(\mathsf{n}) = \textbf{if } \mathsf{n} \in \mathsf{s} \textbf{ then } \mathsf{return\ true} \textbf{ else } \mathsf{return\ false}.$

1.2 Pointer Problems

The major assumption of the traditional method for proving data refinement, as we have explained above, is the ability to divide the client variables from the module variables. Then, the client program can access the internal representation of the module *only* through the provided operations; it *cannot* directly access the internal representation of the module because it does not have a direct access to the module variables, i.e. it has no knowledge of their existence. However, a broad range of programs make extensive use of pointers. For instance, infrastructure code such as operating systems kernel routines, device drivers, database servers – they all manipulate pointers. Unfortunately, the mere presence of pointers violates the assumption that a program state can be divided statically. With variables only, the separation is easy – one can simply say variables $x, y, z, ...$ are to be visible only to the client program, and variables $m, n, p, ...$ are to be local to the module. Then, simple static checking can enforce the required separation. This is not the case in presence of pointers. Because, with pointers there are locations not named by variables, so simple scoping constraints do not ensure separation.

In what follows, we describe the three most common problems which arise in the presence of pointers and explain the effect they have on data refinement. As we will see, the problem is not to find a relation between the abstract and concrete modules, when considering them in isolation. Often, that can be done without difficulties. The problems arise when we want to “lift” this relation to the whole language. In many cases the lifting part does not work, but without lifting, simulation between the modules is useless.

1.2.1 Information Leaking

A common problem that arises in presence of pointers in a setting of data refinement is *information leaking*. In an environment where a client program interacts with a module, this happens when the internal representation of the module is unintentionally (or even intentionally) exposed to the client program. To understand the problem better and to shed some light on what consequences it has for data refinement, we consider the following example. We give the abstract specification of a LIFO Stack. The stack is represented as a sequence of integers, and operations push and pop which, respectively, add an integer to the beginning of the sequence and take an integer off the beginning of the sequence. For its concrete implementation we choose a linked-list representation, where the push operation adds a new element holding an integer value to the beginning of the list, and the pop operation takes an element from the beginning of the list, reads the integer kept in it and passes it over to the client. The abstract specification and its concrete implementation are given in Table 1.3. Symbol ε represents the empty sequence and $\cdot$ denotes the concatenation operation.

We first informally "prove" that the linked list representation simulates the sequence representation of the Stack. Note that here we have pointers. This means that, apart from assigning values to variables, a program state also assigns values to (unnamed) memory locations. We assume this for the rest of this chapter and denote state with σ. Relation between two implementations of the Stack can be given as a relation between the concrete and abstract states.

$$\begin{aligned}\sigma_1[R]\sigma_2 \iff \quad & \sigma_2(a) = \alpha, \\ & \text{in state } \sigma_1 \text{ variable } l \text{ points to a list that holds sequence } \alpha \text{ and} \\ & \text{for all other variables and locations, } \sigma_1 \text{ and } \sigma_2 \text{ have the same values.}\end{aligned}$$

Suppose that we are given R-related states σ_2 containing sequence α and σ_1 containing a list pointed to by l. We need to prove that after running $\mathsf{push}(i)$ on the state σ_1, the newly obtained state is related by R to the output state of the abstract operation $\mathsf{push}(i)$ when run on the state σ_2. Since σ_1 and σ_2 are related, l points to a list which holds sequence α, and after a call to $\mathsf{push}(i)$ it points to the sequence $i \cdot \alpha$ in the output state. Note that this sequence is exactly the one that is obtained after running abstract operation $\mathsf{push}(i)$ on state σ_2. To argue that the concrete $\mathsf{pop}()$ operation simulates the

Table 1.3: Stack – sequence (left) and linked list (right) representations

```
intsequence $\alpha$;
void push(int i);
    $\{\alpha = \mathrm{i} \cdot \alpha\}$

int pop();
    if $(\alpha \neq \varepsilon)$ then
        {return car($\alpha$)}
    else
        {return $-1$}
```

```
Link l;
void push(int i);
    {Link t;
    t = l;
    l = malloc(sizeof(Element));
    l $\rightarrow$ data = i; l $\rightarrow$ next = t}
int pop();
    {Link t;
    if (l ! = NULL) then;
        {t = l; l = l $\rightarrow$ next;
         return t $\rightarrow$ data}
    else
        return $-1$}
```

abstract one, we need to inspect two cases. The first case is when the sequence in the abstract state σ_2 is empty. Then, the variable l is a null pointer in state σ_1 and the list holds the empty sequence. In this case both the abstract and concrete operations return -1. The other possibility is that the sequence is not empty and then we can represent it as $i \cdot \alpha$. The linked list in the related state then holds this abstract sequence and the variable l points to such list. When we run operation pop() on the concrete state containing the linked list, the resulting list holds the sequence α. On the other hand, the state containing the sequence α is exactly the one obtained as the output of the abstract operation pop(). Note that these operations do not change the value of any other variables or memory locations. Hence, the output states are in both cases related.

We argued using the simulation method that the operations of the linked list representation simulate the corresponding operations of the abstract one, and that should imply, by the soundness of the method, that the concrete representation can replace the abstract one in any computation. Now suppose that there is an additional operation called bad(), which as a result returns a pointer. In the abstract representation it returns

Table 1.4: The bad operation

Abstract	Concrete
void bad(Element $*$x);	**void** bad(Element $*$x);
{x = malloc(sizeof(Element));}	{x = l;}

Table 1.5: Client using bad() method

```
main(){
   Element *x;
   push(17);push(2);push(42);
   bad(x);
   x → next = x;}
```

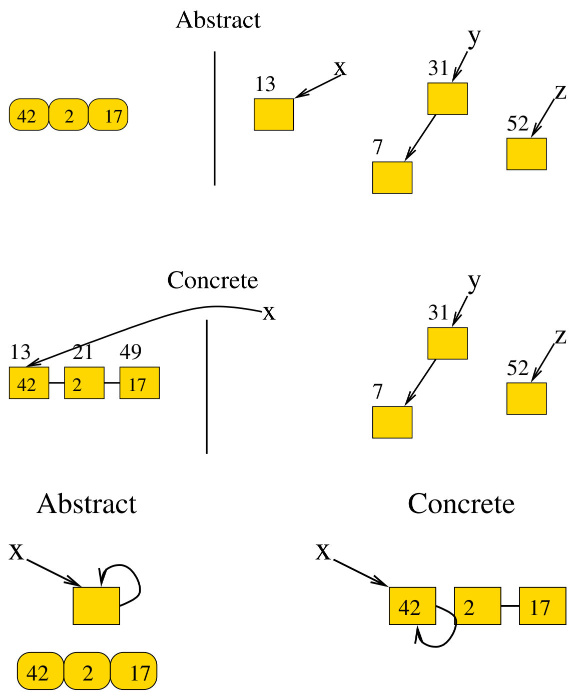
just any old pointer, as we assume a completely nondeterministic allocator, while in the linked list representation it returns the pointer to the first element of the linked list. The code is given in Table 1.4.

Using the refinement relation R defined above, we contend that the operation bad() in the linked list representation simulates the abstract bad() operation. Suppose we have a state σ_2 containing a sequence α and a state σ_1 containing a list which holds sequence α pointed to by l, which are related by the relation R. If we run the concrete operation on state σ_1, the resulting list still holds the initial sequence, but we have a client variable x pointing to the beginning of the list. On the other hand, running the abstract implementation on the state σ_2 leaves us also with the initial sequence and an extra location with a client variable x pointing to it. Relation R requires that the linked list holds the corresponding sequence in order for states holding them to be related, and this is indeed the case. In both concrete and abstract output states, variable x points to a pointer. In fact, since the initial states are related by R, all other variables and memory locations have identical values. This implies that the address of the pointer l is available in the abstract state (it can be easily be seen that it is not allocated in state σ_2) and can be allocated during the execution of bad(), so the identity can be reestablished on the remaining (client) part of the state.

However, if we look at the code in Table 1.5, we can see that in this case the concrete representation cannot replace the abstract one.

If we run this program with the linked list representation, after the third call to the push() operation, the linked list holds a sequence which contains elements 17, 2 and 42. The statement bad(x) places a pointer to the first element of the linked list into the variable x, and the final statement ties a knot in the linked list, which has as a consequence that the resulting linked list no longer holds the intended sequence. On the other hand, if we run the program presuming the abstract representation, the returned pointer after the bad(x) statement is a newly allocated one and the last statement is harmless for the internal representation of the module resources, and so after its execution we still have the same sequence, unlike in the concrete case.

Figure 1.1: Information leaking



Even though our example relies on the fact that a nondeterministic allocator is used, and that it might not work for some other allocators, it shows the point of information leakage. For instance, instead of a nondeterministic allocator, we could have used a nondeterministic assignment operator in the code for the abstract bad operation, and still would have gained the same effect.

This shows that, even though the linked list representation of Stack simulates the sequence representation, they cannot be interchanged in all computations. This example discloses the inadequacy of the traditional method for proving data refinement. A more formal account of the breakdown of the traditional method will be given in Chapter 2.

1.2.2 Ownership Transfer

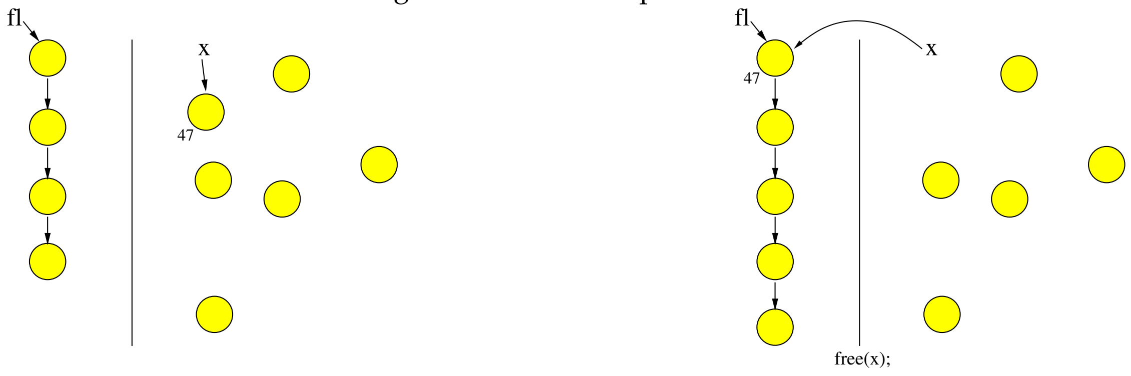
An inevitable issue that emerges in presence of pointers, where an environment in which client program is interacting with a module is considered, is the so-called "ownership transfer". Unlike in cases where only variables are used in the computation, when heap memory is manipulated, it is impossible to divide the resources between the client and the module once and for all in the beginning of the computation. Throughout the process of computation, memory locations are "transferred" back and forth between a client and a module, and so, any initial division would be violated. A consequence of ownership transfer is exposing the internal data representation of the module. So, it is technically the same phenomenon as information leakage. However, it is conceptually different as information leakage is unintentional and is a result of a bug, while ownership transfer is intentional and is not a bug.

To understand the issue better, we consider a toy Memory Manager. The memory manager maintains a list of locations available for allocation – the "free list". The free list is manipulated through the operations for allocation and deallocation. The code for these operations is given in Table 1.6, and the free list is declared to be pointed to by a variable fl, visible only to the mentioned operations.

Operation alloc() works by taking the first element off the free list and returning it to the client program. In case the free list is empty, the operation calls cons, an idealized version of the Unix system routine sbrk that never runs out of the memory. Operation free puts a disposed location on the front of the free list. Note that the free operation does not break any existing links to the disposed location. For example, if a client variable holds a pointer to the location that is to be disposed, it will still point to it after the disposal.

The Toy memory manager is a concrete implementation of the Magical memory manager, described also in Table 1.6. The Magical memory manager does not own any memory, but instead when the alloc() operation is called, the system routine cons() is

Figure 1.2: Ownership transfer



initiated to provide a new location. Similarly, when the free(x) operation is called, the location pointed to by x is returned to the system.

To show that the toy memory manager is an implementation of the Magical one, we employ the following relation between concrete and abstract states:

$$\sigma_1[R]\sigma_2 \iff \begin{array}{l} fl \text{ points to a non-circular linked list and} \\ \text{the rest of the state } \sigma_1 \text{ is identical to whole of } \sigma_2 \end{array}$$

This relation reflects the fact that the internal representation of the abstract memory manager owns no memory, and imposes the requirement that "client" parts of the abstract and concrete states are equal.

We now argue that the concrete alloc() simulates the abstract one. Starting from the concrete and abstract states σ_1 and σ_2 respectively, when the concrete alloc() is called, the first element in the list is taken and returned to the client, if the list is non-empty in state σ_1. Then, we are left with the list which has one element less then the initial list in the output state. If the list is initially empty, then the system routine is called to provide a new location. In both cases we are left with either fl being a null pointer, in which instance the list is empty, or pointing to some non empty list in the concrete output state. In the abstract execution the magical alloc() is executed on state σ_2 and only the value of variable x is changed, now holding the address of the newly allocated memory location. We now need to prove that the value of all variables and memory locations in the abstract output state are identical to their values in the concrete output state. Since the value of all "client" variables (all the variables except from fl) in the initial related states are identical and the abstract module has no heap nor variables playing part in its representation, the location that is taken from the list and given to the client must

Table 1.6: Memory manager

Magical	**Linked-list**
*int alloc(){	**static** Link fl;
return(cons());}	Link alloc();
void free(int *x){	{Link res;
dispose(x);}	**if** (fl = NULL) **then**
	{res = cons(sizeof(Element));
User	return res}
main(){	**else**
int *x;	{res = fl; fl = fl → next;
x = alloc();	return res}}
x → data = 5;	**void** free(Link x);
free(x);	{Link t;
free(x)}	t = fl; fl = x; x → next = t}

be unallocated in the abstract state, and so, available for allocation. This means that there exists an execution of the abstract operation which would maintain the identity between the values of variable x in abstract and concrete client states. Note that values of all other variables and memory locations stay unchanged after the execution of the allocation operations. Hence, the discussed output states are related by relation R.

To argue that the concrete free() simulates the magical one, note that after calling the concrete operation to dispose a location pointed to by the client variable, the free list grows by one element, and retains the structure of the linked list. Again, the "client" states stay identical, since the same location is deallocated and values of all other variables and memory locations in both concrete and abstract states remain the same.

Now, consider the client program of the memory manager module, given in Table 1.6.

The client program allocates a new location, dereferences it and then deallocates it. Initially, the client knows only about the declared pointer variable x. If the client interacts with the toy memory manager, before the allocation, the module variable fl points to the free list. After the allocation, the first element is taken off the linked list and a pointer to it returned to the client. After manipulating the newly allocated memory and freeing it, the location is handed back to the free list, but the client variable x still points to it. When the client frees the pointer in the variable x for the second time, it ties a knot in the free list of the memory manager, and the memory manager variable fl no longer points to a non-circular linked list, but rather a circular one with only one element. Now, the resource no longer satisfies the invariant, and since the refinement relation imposes the same invariant condition, the refinement is also broken. The interesting point here is that the module invariant is broken and so, the problem exists prior to the refinement, but this is also reflected in the refinement reasoning.

We suggest that it is helpful to think of this problem in terms of "ownership". The allocated location is initially in the free list of the memory manager, that is, it is initially *owned* by the module – it *belongs* to it. After the allocation, the ownership of the location is *transferred* to the client program, and the client program is now entitled to use it as it likes, until it deallocates it, when the ownership of the location is transferred back to the free list of the memory manager. Since the deallocated location is now owned by the module, the client should not manipulate it whatsoever; as a matter of fact, if it does, the program might crash and the data abstraction is violated. The concept of "ownership" here is simple: at any program point we regard the state as being separated into two parts, and ownership is just membership in one or the other component of the separated states.

As we have already mentioned, even though they both lead to exposing the internal data representation and potential violation of encapsulation, information leaking and ownership transfer are different in their nature. While ownership transfer can often be useful, information leaking is normally considered an undesirable phenomenon. Ownership transfer gives means for moving data between client and module, or even between modules. In our example, the location is transfered between a client and the memory manager module; this is necessary for good memory management. On the other hand, unless malicious, the client does not benefit from information leaking. It gains access to the internal representation of the module which should not be used in any circumstances. Ownership transfer is a way to conceptualize how exposing internal representation in certain circumstances is actually not a bug.

Table 1.7: Allocation-status testing examples

module counter1 {	module counter2{	module counter3{
init(){	init(){	init(){
*1=allocCell2();**1=0;}	*1=0;}	*1=alloc();**1=0;}
inc(){	inc(){	inc(){
1=(1)+1;}	*1=(*1)+1;}	**1=(**1)+1;}
read(){	read(){	read(){
*3=(**1);}	*3=(*1);}	*3=(**1);}
final(){	final(){	final(){
free(*1);*1=0;}	*1=0;}}	free(*1);*1=0;}}

check2 ≡ z=alloc(); if (z==2) then v=1 else v=2.

Interestingly enough, apart from corroborating that pointers wreak havoc with data abstraction, the memory manager is a canonical and natural example, and very often a stumbling point for many proposed solutions to the problem that pointers cause for data abstraction.

1.2.3 Allocation-status testing

We have so far explored the difficulties which arise from the use of pointers, which can be easily understood and demonstrated. Still, there are other, more subtle accessing mechanisms, which are not so apparent. One such known mechanism is *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. We again illustrate the problem with an example.

In Table 1.7 there are three different implementations of a Counter module, with operations for initializing and finalizing the module, and operations for incrementing the counter and reading from it. Module counter1 initialization is done in couple of steps. Firstly, the location 2 is allocated. This is done by calling the operation allocCell2() which returns location 2 if it is available and it diverges otherwise. The address of the newly allocated location, namely 2, is kept in location 1, which is assumed to be part of the module prior to initialization. Secondly, the value of the counter is then initialized to zero and stored in the allocated location 2. Module counter2 uses only one location – location 1, to store the value of the counter. The main difference between counter1 and counter2 modules is that the second module uses less space than the first module. The third implementation of the counter, namely counter3, uses two locations and, unlike counter1, picks *nondeterministically* the location in which the value of the counter is actually kept.

Program check2 is also given in Table 1.7. It "observes" the allocation-status of location 2 by changing its nondeterministic behavior. Suppose that we run program check2, together with the module counter1. Then, upon execution of check2, variable v always has value 2. This is the case because prior to the execution of check2, the module is initialized and, as we have already explained, location 2 is allocated. Then location 2 is no longer available for allocation in the client program check2, and so the test of the client program always fails, and therefore the v is always assigned value 2. On the other hand, if we run check2 together with the module counter2, variable v can have either value 1 or value 2 in the end. Notice that in counter2, initialization does not allocate location 2, and so when allocation is initiated by the client program, it is available. This means that the address in variable z might be either 2 or something else, and so the test in the client program might fail or succeed, which as a consequence has that the variable v might have one or the other value. This implies that the optimized module introduces new behavior to the client check2, which is unacceptable for sound method for proving data refinement.

1.2.4 Summary of problems

In this section, we have described the following phenomena which contribute to the unsoundness of the existing method for proving data refinement:

- information leaking – a module leaks information about its internal representation to a client,
- ownership transfer – in the interaction of a module and a client the internal representation of the module is exposed to the client, and

- allocation-status testing – using dirty techniques, a client may discover information about the module's internal representation.

All these concepts make reasoning about data refinement difficult. The examples that we presented show how easy it is to "prove wrongly" data refinement between two implementations. By "prove wrongly" we mean that the simulation between the modules can be easily proved, but because of the failure of lifting theorem, it does not lead to data refinement. In fact, the first requirement of data refinement, that there is no client program which can observe the difference when the abstract module is replaced by the concrete one, is not met.

Our solution to the problems that arise when data abstraction is faced with pointers focuses on those client programs that behave "well".

1.3 Contributions of the thesis

The main aim of the thesis is to develop sound methods for proving data refinement and to ensure that in spite of problems raised by pointers, which are thoroughly described in Section 1.2, the abstraction results can be achieved.

We present the list of the contributions of the thesis in what follows.

- **Separation contexts**
  - One of the main assumptions of the traditional method is that the resources of a module and a client program are separate. Unfortunately, when pointers are considered it is impossible to separate these resources once and for all before running the program. However, as all the considered examples suggest, tracking in each step of the computation which resources belong to whom is realistic. We use this knowledge to semantically impose such tracking. In our approach to the problems illustrated in previous sections, we identify the class of "well behaved" client programs, *separation contexts*. A separation context is a program which respects the internal representation of a module with which it interacts. It never reads or writes any memory that "belongs" to the module directly, but only through the operations provided by the mod-

ule. To illustrate the idea, we revisit the example of a Toy memory manager given in Table 1.6.

– In our theory, we consider only separation contexts. They are defined semantically, but we also provide a more practical way of checking whether a given program is a separation context for a specific module. Namely, proving a certain specification of a client program using separation logic ensures that the program is a separation context for the module assumed in the proof.

Example. To begin with, we regard the program state as being separated into two parts, one of which belongs to a client, and the other which belongs to the module. The module's part always contains a free list, whether it be an empty list or a list with a certain number of available locations. When a new location is allocated, it is taken off the free list and put into a client variable. At this point we regard a boundary between the client and the module states as shifting: the ownership of the location has transferred from the module to the client, so that the separation between client and module states is dynamic rather than determined once-and-for-all before a program runs. Similarly, when a client disposes a location we regard the ownership of that location as transferring from the client to the module.

Now, the program in Table 1.6, as we have shown, tramples on the free list of the memory manager, and so contradicts our assumption of separation: the second free(x) statement accesses a cell which the client does not own, since it was previously transferred to the module. In fact, *any* attempt to use the location after first free() will contradict separation, say, if we replace the second free() by a statement x → link := x. But, for instance, the following program *is* a separation context.

```
main(){
  int *x;
  x = alloc();
  x → data = 5;
  free(x); }
```

If we consider client programs which are not necessarily separation contexts, we get into a situation where the programs contradict abstraction. For instance, if the memory manager uses the $\mathsf{x} \rightarrow \mathsf{link}$ field as a pointer to the next node in the free list, then in the following code

```
main(){
  int *x;
  x = alloc();
  x → data = 5;
  free(x);
  x → link = x}
```

the execution of the $\mathsf{x} \rightarrow \mathsf{link} := \mathsf{x}$ will corrupt the free list. However, if the memory manager uses a different representation of the free list, corruption might not occur: client becomes representation-dependent. This shows the importance of separation contexts with regards to data abstraction.

- **Data refinement**
  - In our approach to data refinement, we disregard "badly behaved" client programs, we consider only separation contexts. In other words, in computations generated by non-separation contexts, we do not require that the concrete representation of the module can replace the abstract one, even when the simulation between the different implementations of the module exists. One of the conditions for our methods for proving data refinement to be sound is considering only separation contexts.
  - The forward simulation method involves refinement relation between the states of an abstract module and the states of a concrete module. In order to prove that one operation forward simulates another, one must ensure that starting from related states, whatever step the concrete operation makes, the abstract one is able to perform the same step in such a manner that the refinement relation between the output states still holds. The refinement relations have to be restricted in a certain way in order for the method to be sound. This requirement narrows a range of the examples that can be tackled with forward simulation method. However, it can be used to prove the refinement for most of the canonical examples.

– We introduce a power simulation method which is more general method for proving data refinement. As described in Section 1.2.3, there are certain client programs which can detect space-optimizations, and forward simulation method is not strong enough to deal with such problems. This also has an impact on proving the equivalence between two different implementations of some abstract specification. A good example where one would like to prove the equivalence is the two implementations of the doubly ended queues: one with doubly-linked lists, and the other one with the space-saving XOR-linked lists. The code for these two implementations is given in Table 1.8. In most cases, we will be able to prove data refinement in one direction – as we can prove that doubly-linked lists forward-simulate the XOR-linked list representation – but for the equivalence we need the other direction, too. The problems may arise if the other implementation is a space optimization of the first one. Power simulation method successfully deals with space-optimizing problems. Power simulation method allows only space optimizations of the nondeterministically allocated memory. That way, the identity of the memory space to be optimized is hidden from a client program, and all the allocation-status testing fails to give any useful information. For instance, the power-simulation method allows counter3 in Table 1.7 to be optimized by counter2 from Table 1.7, because the internal cell in counter3 is allocated nondeterministically. Note that even with check2 given in the same table, a client cannot detect this optimization, e.g. when cell 2 is free initially, $\mathsf{counter3.init}(); \mathsf{check2}$ nondeterministically assigns 1 or 2 to v, just as $\mathsf{counter2.init}(); \mathsf{check2}$ does.

– As already mentioned, the power-simulation method is more general then the forward simulation method. In fact, in certain circumstances power-simulation can be reduced to forward-simulation, which is easier to use.

It may seem that our solution is narrow and targets only C-like programming languages. We would like to emphasize here that the issue raised by the pointers when considering data abstraction is not exclusive to the low-level programming languages.

For example, in a garbage collected languages thread and connection pools are sometimes used to avoid the overhead of creating and destroying threads and database connections (such as in a web server). Then, a thread or connection id should not be used after it has been returned to a pool, until it has been doled out again.

1.4 Related Work

The initial work on data refinement dates back to early 1970s. Wirth first introduced the idea of gradual refinement of programs and data specifications in parallel in his paper on stepwise refinement [79]. Hoare expressed, in an informal way, his suggestion of how to prove a connection between two different implementations of a data type in order to ensure the correctness of the more concrete one with respect to the abstract one, in his 1972 paper [35]. This suggestion was embraced and developed in the VDM model based formal method for description and development of computer systems [42]. A more formal account of data refinement was presented in paper by Hoare et al. [33]. The operations of the data type were allowed to be total relations and the connection between the abstract and concrete data types no longer had to be functional. They introduced two kinds of simulation: upward and downward, which were together necessary and sufficient for sound and complete data refinement. Hoare and He generalized Dijkstra's weakest precondition [30] introducing the notion of the weakest prespecification in [37]. They showed how it can be used in the VDM framework for derivation using the simulation method. In their technical monograph [38], Hoare and He gave an account of data refinement in a categorical setting, where they explicitly state the assumption that there is a static separation between the client and the module. They also consider lifting of data refinement from module to the whole language. All the work mentioned so far, considers data abstraction and data refinement with regard to languages which do not utilize pointers.

A systematic study of model-oriented proof methods of data refinement can be found in De Roever's et al. book [29]. Nancy Lynch et al. also studied simulation techniques, for instance in [48, 49, 47].

Pointers wreak havoc with data abstraction and module encapsulation. There are a number of early documents which confirm and try to resolve this problem. One of them is the Geneva convention on aliasing problems [40]. The document defines and explains aliasing in the object-oriented context and gives a categorization of the approaches to the problem. In his paper [39], Hogg introduced a concept of islands which prevent problems caused by aliasing. Islands are used to isolate a group of related objects. An island is a completely encapsulated unit, within which any system of aliasing control can be used. Capsules are another idea introduced by Wills [77]. The system Fresco based on capsules is used for program verification in Smalltalk. Systems are built by composing capsules, which contain both code and specification, including the assertions about aliasing. Similar concepts are confined types [76] and balloons [4].

Another approach to encapsulation of data representation is work on ownership types. Ownership types impose an ownership hierarchy on objects – an object can own other objects which are involved in its internal representation. References into a certain object are allowed only through the owner of the object and so, no internal representation objects are accessible directly from the outside. Ownership types were first proposed by Clarke et al. [26] and they formalized this idea in their later work [25]. Here, they enforce strict encapsulation but on account of expressiveness. In work by Boyapati et al. [17, 19] and Clarke et al. [24] ownership types were extended to support a natural form of subtyping, but to allow iterators and similar constructs the encapsulation was allowed to be temporarily broken, and hence local reasoning was not supported. In [18] the ownership types system was proposed which is both expressive and supports local reasoning. There is a lot of research done in the area of ownership and encapsulation, and here we mention only some of them [43, 66, 65].

Work of Banerjee and Naumann on confinement [10] also imposes typing restrictions to ensure representation independence. Namely, they introduce a notion of "confinement" which requires a heap to consist of three parts: client, class interface and its internal representation. Only links between a client and a class interface, and between the class interface and its internal representation may exist. All other references are forbidden. In their following paper [8], Banerjee and Naumann remove the restriction that pointers from data representation to outside of it may not exist; these pointers may now be fully used. They also prove generalized abstraction theorem and identity-extension lemma. In their work on ownership [9], they use assertions and auxiliary fields to enforce ownership relations, heap encapsulation and control of reentrant callbacks.

Reddy and Yang, in their paper [67], consider correctness of data representations that involve heap data structures. Unlike Naumann et al., they do not base their work on explicit confinement conditions. Instead, their semantics reveals the breach in data encapsulation in cases such as information leaking. The semantics is founded on reachability and information hiding of the module's internal state is captured only if it is not reachable from client variables. Their notion of parametricity does not view programs with different internal representations as being equivalent, when there are cross-boundary pointers. In fact, such programs are not equivalent in all program contexts. By focussing on fewer contexts, just the separation contexts, we are able to reason about such data abstractions. A representative example is a memory manager with malloc() and free() operations.

Verification methodology for model fields [44], based on the Boogie Methodology for object invariants [12, 45], uses specification only fields to enable the abstraction of the concrete state of a data structure. This work addresses the problems caused by the mutable objects and aliasing when dealing with modularity and data abstraction. Parkinson's abstract predicates [61] present another approach to dealing with frame properties and abstraction.

Back is one of the first promoters of the weakest precondition predicate transformer semantics in program development and data refinement [6]. Recently, he has addressed the problems that arise from pointers [7], by converting all the pointer operations into assignment statement and then applying the rules of refinement calculus to construct programs. Butler also used techniques of refinement to focus on the derivation of tree-based pointer algorithms [20]. These approaches, however, aim at the program development problems rather than those of data refinement. Influenced by predicate transformer semantics [31], Morgan et al. introduced a single complete rule for data refinement [32]. This is similar to our notion of power simulations. However, this work, as their other work on data refinement [54], does not address problems raised by pointers. Power simulation is also closely related to Reddy's method for data refinement [68]. In order to have a single complete data-refinement method for a language, again *without* pointers, he lifted forward simulation such that all the components of the simulation become about state sets, instead of states.

Work of Abrial on the event based sequential program development [3], gives means for constructing a program from its abstract specification. His approach, based on the *B method* [2], allows refinement of pointer programs, but is concerned with program development rather then data refinement. However, the importance of the B method is huge, as there is a lot of research based around the B method, for instance [21, 5], as well as the fascinating industrial applications [63].

What sets our work apart from all this other work? First of all, there are two separate issues regarding data abstraction. One of them focuses on mechanisms for checking data abstraction, and the second one on what is its meaning. In the thesis we are concerned with the second issue, the meaning of data abstraction, which we try to get at by defining the notion of separation context. The reason abstraction is broken by using pointers is that pointers are, among other things, used to dereference the internal representation of a module from the outside. Now, islands, balloons, confined types, ownership types and heap confinement all provide *mechanisms* for checking data abstraction based on restricting cross-boundary pointers. However, they are not very clear on what do these mechanisms achieve or imply. In our opinion, they achieve that pointers into module internals will not be dereferenced; this implies that you can refine one module with another, without having to re-check the clients. Banerjee et al. explicitly consider this aspect, too. We wanted to understand the reasons for why language restrictions might work when approaching data abstraction for pointers. Our suggestion is that separation between a module's internal representation and a client's state is the key idea. By taking this perspective, new possibilities open up, as we show in this thesis, with regard to ownership transfer and embracement of low-level features.

Although work in the thesis presents a semantic advance, separation contexts are a semantic concept, and consequently, they are harder to check then whether imposed language restrictions are respected. Our results on the connection between separation contexts and logic give some useful information and possibilities, but we have taken semantics, rather than static checkability, as our main aim.

The work of Leino et al. is much more flexible and closer to ideas in this thesis. They use abstract variables for specifying modules and develop "modular soundness" to establish in which circumstances clients cannot access the internal representation of a module. Hence, they do not experience problems with cross-boundary pointers and modular reasoning about frame properties. However, they enforce their modules to preserve data abstraction in order to be able to verify them and use them in larger systems; they do not present results on a question of when can one representation be replaced by another.

Parkinson's abstract predicates are also tailored to deal well with modularity and abstraction, but his work does not consider data refinement. Work of O'Hearn et al. [58] on information hiding, where modules are represented using internal resource invariants, was our main inspiration. The resource invariants fit naturally with refinement. Parkinson does not have internal invariants. Instead, the resource of a module is explicitly exposed but the way it is manipulated is constrained. It would be interesting to work out an alternative refinement theory based on it.

A lot of the work on program development in presence of pointers, like that of Abrial and Back, as explained earlier, focuses on program refinement rather than data refinement. The majority of program refinement techniques do not support local reasoning about pointers and often, certain properties of a data structure (that corresponds to a module in our setting) once proved in the development of one program, must be reproved if they are to be used in the development of another. Our approach supports local reasoning about pointers and once data refinement is proved between two modules, the more concrete one can replace the more abstract one in all separation contexts. Some of the program refinement techniques provide theories for a particular data structure (like those of Butler), such as a list theory, or a tree theory, but once a new data structure is introduced, a new theory has to be built. Our precise predicates ensure that we can deal with virtually any data structure.

Finally, since the work in this thesis has advanced reasoning about ownership transfer, we discuss this issue in the light of related work. Ownership transfer is one of the stumbling points of the work on reasoning about programs with pointers. Some approaches simply cannot handle ownership transfer [39, 4, 10], while admitting that this is a limitation. However, some solutions have been presented in order to solve this issue. Work of Clarke et al. [23] proposes a concept of external uniqueness, while that of

Naumann et al. [11] builds on it. External uniqueness requires that there is a unique reference from the outside to the aggregate, while there can be many from the inside. Naumann et al., apart from requiring that there is a unique pointer from the sending owner to the object o which is to be transfered, demand that either there are no references back from the object o to the owner, or all reachable objects from o are transfered together with it. They cannot ensure that imposed uniqueness conditions are satisfied, but rather assume they are. On the other hand, our work does not suffer from such restrictions, i.e. we do not need to impose external uniqueness conditions. Our notion of separation context is defined semantically, and however many aliases might be introduced by transfering a location from client to a module, if any of them is dereferenced, the semantics will detect that. There is also an ongoing work on ownership transfer based on universe types, by Müller [55].

The Boogie methodology can deal with ownership transfer in a more flexible way than the type-based approaches. In comparison to the work here, Boogie does not encompass low-level features such as address arithmetic (like one might find in a memory manager), and they have not proven an analogue of the lifting theorem for data refinement as far as we are aware. But, we acknowledge that the Boogie approach is powerful, and promising. Also, we emphasize that the ideas in that work were arrived at independently of, and virtually concurrently to, the approach here.

Parkinson can successfully deal with ownership transfer, but as we already pointed out, our main goals somewhat differ.

We found great motivation for our work on data refinement in Hoare's work [35]. However, the intuitive and technical inspiration comes from ideas in Separation logic [72, 41]. Namely, in their work on information hiding, O'Hearn et al. consider modules that are represented by their resource invariants and a set of module operations. They use separating conjunction to provide separation between the internal states of the modules and the client.

Our approach to data refinement is modular in the following sense. We allow for simulation to be proved independently of any external users or other modules. Then, by lifting the simulation to the whole language, we enable our module to be incorporated into a bigger system. In our work, we also consider behaviors that are present in “dirty”, low-level programming, such as cross-boundary pointers and ownership transfer, which are not manageable by linguistic restrictions, in particular, by any of the mentioned theories.

1.5 Overview of the Thesis

The thesis proposes a theory which gives rise to a sound method of data refinement in presence of low-level pointer operations.

We give a detailed discussion of the classical theory of data refinement and a more formal account of the shortcomings of the method in Chapter 2. We first show how the method works in a setting in which it is sound, and then gradually introduce conditions inspired by the real-world programming techniques which invalidate the traditional method.

The notion of a *separation context* is central to our theory of data refinement. The most intuitive explanation of a separation context is that it is a well-behaved client program that does not interfere with the internal representation of a module with which it interacts. We introduce separation contexts in Chapter 3 and prove some important properties of them. We also illustrate separation contexts with several examples and non-examples.

In Chapter 4, we present the technical core of our method. We introduce our forward simulation method and prove the “abstraction” or “simulation” theorems, which also imply the soundness of the method. The simulation theorem states that a simulation relation between two modules can be lifted to all separation contexts. This is particularly important feature, as otherwise an existence of a simulation between two modules is useless. We also illustrate the method with two examples. We formally prove successive refinements between linked-list and set representations, and set and “magical” representations of a memory manager module.

Separation contexts are defined semantically in Chapter 4. Chapter 5 gives a more practical way of deciding whether a certain program is a separation context. Namely, using the forward simulation method, we show that it is enough to prove that a program meets a certain specification in separation logic.

In order to prove soundness of the forward simulation method, we had to impose certain restrictions to the refinement relations to be allowed in the method. This leads to the narrowing of the range of the problems that forward simulation can solve. A more general method for proving data refinement is a power simulation method, which we present in Chapter 6. We also prove the "abstraction" theorem and soundness of the power simulation method. This method enables us to handle more demanding examples, such as equivalence between the doubly-linked list and XOR-linked list representations of queues.

Power-simulation method is more general then the forward-simulation method and can be reduced to it in certain circumstances. In Chapter 7 we give conditions under which this can be done. In this chapter, we also illustrate the power simulation method with a detailed example, the equivalence of the doubly-linked and the XOR-linked list representation of the doubly-ended queues.

We conclude and give the directions for future work in Chapter 8.

Table 1.8: Doubly-linked list and XOR-linked list implementations

```
struct dnode{
    int data;
    struct dnode *left
    struct dnode *right}
typedef struct dnode Delement;
typedef Delement *Dlink;
static Dlink f, b;

void insert(int i; Dlink m, n, k; );
    {Dlink t1;
     t1 = malloc(sizeof(Delement));
     t1 → data = i;
     t1 → left = m; t1 → right = n;
     if (m = NULL) then f = k
     else m → right = k;
     if (n = NULL) then b = k
     else n → left = k; }

void delete(Dlinkk; );
     m = k → left;
     n = k → right; free(k);
     if (m = NULL) then f = n
     else m → right = n;
     if (n = NULL) then b = m
     else n → left = m}
```

```
static Link fl, bl;

void insert(int i; Link m, n; );
    {Link t1, t2;
     t1 = malloc(sizeof(Element));
     t1 → data = i;
     t1 → next = m^n;
     if (m = NULL) then f = t1
     else
         {t2 = m → next;
          m → next = t2^n^t1};
     if (n = NULL) then b = t1
     else
         {t2 = n → tail;
          n → tail = t2^m^t1}}.

void delete(Link k, m; );
    {Link n, t2;
     n = (k → next)^m;
     free(k);
     if (m = NULL) then f = n
     else
         {t2 = m → next;
          m → next = t2^n^k; };
     if (n = NULL) then b = m
     else
         {t2 = n → tail;
          n → tail = t2^m^k; }}.
```

2

The Classical Theory of Data Refinement

Central to the classical theory of data refinement is the simulation method and the "lifting theorem" [29, 33, 38]. The lifting theorem ensures a sound way of extending a simulation between modules to a simulation between client programs which interact with the modules; the relation preservation property then *lifts* from the operations of the modules to the whole language. Without this property, simulation between just the modules is useless. The client language for which the lifting theorem can be proved is the simple while language, which uses only variables to store values. The problems arise when the memory model is extended or new features, such as direct memory access, added to the programming language.

In this chapter we give a formal account of the simulation method and prove the lifting theorem in a classical setting. We then introduce the extensions to the memory model and the programming language and provide simple examples which expose the failure of the lifting theorem.

2.1 The programming language

The underlying assumption which we will carry throughout the thesis, as already suggested in the Introduction, is that the environment in which our programs are executed consists of a module (one, if not stated differently) and a client program. We can see the client as a program that uses the services of the module.

For the memory model, we assume that a set of variables Var is given, and moreover, that this set consists of two disjoint parts: module variables Var_m and client variables Var_c. In accordance with the splitting of the set of variables, in any state of execution there are two substates – the module and the client (sub)states. The states assign values to the variables, so we have

$$S_m = \mathsf{Var}_m \to \mathsf{Val}, \qquad S_c = \mathsf{Var}_c \to \mathsf{Val}, \qquad S = \{s_m \uplus s_c \mid s_m \in S_m \ \wedge \ s_c \in S_c\}$$
$$\mathsf{Val} = \ldots, -1, 0, 1, \ldots$$

A module is a collection of operations, together with a set of declared module variables. We denote a module by $M = (V, f_1, ..., f_n)$. Module operations $f_1, \ldots, f_n$ are defined as relations on states, i.e. $f_i \subseteq S \times S$. We can view a client program as using the services of the module, so the module operations must be able to change the client state. For example, client may call an operation of the module giving a variable as a parameter. Because of this, we let the module operation execute on the whole state (i.e. combined module and client state).

The client programming language is a standard while language with assignment and module operations as basic commands. Note that the client may use (read or write directly) only client variables.

$$const \ ::= \ 1 \mid 2 \mid 3 \mid \ldots$$
$$var_m \ ::= \ x_m \mid y_m \mid z_m \mid \ldots$$
$$var_c \ ::= \ x_c \mid y_c \mid z_c \mid \ldots$$
$$e \ ::= \ const \mid var_c \mid e+e \mid e-e \mid e \cdot e \mid e/e$$
$$c \ ::= \ var_c = e \mid f_i, \ i = 1, \ldots, n \mid c;c \mid \textbf{if } e \textbf{ then } c \textbf{ else } c \mid \textbf{while } e \textbf{ do } c$$

The semantics of the language is standard. When a module is defined, the meaning of its operations is provided and given as relations. Then, the module operations execute in agreement with this meaning. The other commands have the standard meaning.

In work on refinement, the client language is often left implicit. We are being clear about it because we want to highlight problems with lifting simulation relations, this being the area where pointer problems surface.

In Table 2.1, $s[f_i]s'$ denotes two states which are related by relation f_i. Recall that the module operations f_i are defined as relations $f_i \subseteq S \times S$.

Notation remark. We will use the notation $a[r]b$ for any two objects related by some

Table 2.1: The semantics of the language

$$\frac{}{x = e, s \rightsquigarrow s[x \mapsto [\![e]\!]_s]} \qquad \frac{s[f_i]s'}{f_i, s \rightsquigarrow s'}\ i = 1, \ldots, n \qquad \frac{c_1, s \rightsquigarrow s'' \quad c_2, s'' \rightsquigarrow s'}{c_1; c_2, s \rightsquigarrow s'}$$

$$\frac{[\![e]\!]_s \neq 0 \quad c_1, s \rightsquigarrow s'}{\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, s \rightsquigarrow s'} \qquad \frac{[\![e]\!]_s = 0 \quad c_2, s \rightsquigarrow s'}{\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, s \rightsquigarrow s'} \qquad \frac{[\![e]\!]_s = 0}{\textbf{while } e \textbf{ do } c, s \rightsquigarrow s}$$

$$\frac{[\![e]\!]_s \neq 0 \quad c, s \rightsquigarrow s'' \quad \textbf{while } e \textbf{ do } c, s'' \rightsquigarrow s'}{\textbf{while } e \textbf{ do } c, s \rightsquigarrow s'}$$

$[\![-]\!]_s \in \mathsf{Int}$

relation r throughout the thesis. Also, for any function s, by $s[x \mapsto e]$ we denote the same function as s except that it maps x to e.

2.2 The Simulation Method

The simulation method is means of proving data refinement. Intuitively, operation a simulates operation b if whatever step operation a can make in its execution, operation b can perform the same step. This method is also known in literature as the downward simulation [33, 36]. The method is based on the preservation of the "refinement relation" between the operations involved in the simulation. In this section, we first introduce the refinement relation and then formally define the simulation method. We also prove the *lifting theorem* and illustrate it with an example.

A refinement relation $R \subseteq S_m \times S_m$ is a binary relation defined between the abstract and concrete modules. A client program uses services of a module by executing in an environment in which the module is defined and calling operations of the module. This can be regarded as the module interacting with a client program. This suggests that it is not sufficient to observe relation between the modules in isolation, but we also need to reason about the behavior of the modules when they are interacting with a client program. In order to do that we have to extend the refinement relation to also encompass the client states. In fact, we are interested in a situation where there is one client program; we compare its behavior when it is using abstract module to its behavior when it is using the concrete one, and we expect these behaviors to be identical. Hence, we extend the module refinement relation with the identity relation. If we denote by $R \subseteq S_m \times S_m$ the module refinement relation and by $Id \subseteq S_c \times S_c$ the identity relation, then the overall refinement relation is $R \times Id \subseteq S \times S$, where $S = S_m \times S_c$.

We now define the simulation method.

Definition 1. *Let $f \subseteq S \times S$ and $g \subseteq S \times S$ be operations and let $Z \subseteq S \times S$ be a refinement relation. Then f simulates g with respect to Z if and only if for all states s_1, s_2 and s'_2, such that $s_1[Z]s_2$ and $s_2[f]s'_2$, there exists a state s'_1 such that $s_1[g]s'_1$ and $s'_1[Z]s'_2$.*

The simulation method describes the requirements for the behavior of two operations in order to have one simulating the other. Having two states – the concrete and the abstract, related by some relation Z, we have to make sure that the abstract operation g, executing on the abstract state, can do whatever the concrete one, f, can do when executed on the concrete state, up to the relation Z.

The method, in this set-up, allows for a command which always diverges to simulate any abstract command. This is because, if the semantics of the commands is given by partial relations, the condition of the method is vacuously true. But even if we restrict the relations that play a role in the semantics of the commands to be total (which was done in [33]) , the situation does not change, because an empty relation simulates all abstract operations. This problem is discussed in [36], and a suggestion for a possible solution is given. However, in our work we only consider data refinement in a partial correctness setting and hence, we do not have to face this problem.

Having defined what means for an operation to simulate another, we now define what it means for a module to simulate another module.

Definition 2. *Let $R \subseteq S_m \times S_m$ be a relation. Module $M = (V, f_1, \ldots, f_n)$ simulates module $N = (U, g_1, \ldots, g_n)$ with respect to the relation $R \times Id$ if and only if for all $i = 1, \ldots, n$ f_i simulates g_i with respect to $R \times Id$.*

In order to have simulation between two modules, we instantiate the refinement relation to $R \times Id$. Then if each operation of the concrete module simulates the corresponding operation of the abstract module, we say that the concrete module simulates the abstract one. Here, we require that apart from preserving the module refinement relation R, module operations either do not touch the rest of the state, or if they do, they change it in the same way; this is imposed by the Id part.

The lifting theorem is expressing abstraction. It connects the meanings of different representations of the same data type, reminiscent of Reynolds' Abstraction theorem [69] and Plotkin's Lemma of logical relations [64]. Having simulation between the abstract data structure and its concrete representation, the lifting theorem ensures that the client program using the concrete representation simulates the same program when using the abstract data structure. If the lifting theorem did not hold, that would mean that there are commands in the client programming language which can differentiate between the two representations of the abstract specification of the data type.

We can view a client as a context $c[]$ with holes, into which the appropriate operations of some module can be plugged in. Once the client is linked with a module, in place of a hole, the appropriate implementation of a function for which that hole stands is filled. So, for example, if we have a client interacting with a linked list memory manager, then in place of $alloc()$, the linked list implementation of this operation will be called, while if the same client interacts with the magical implementation of memory manager, then the implementation which works on sets will be placed in a hole.

Theorem 1 (Lifting theorem)**.** *Let $R \subseteq S_m \times S_m$ be a relation. Let module $M = (V, f_1, \ldots, f_n)$ simulate module $N = (U, g_1, \ldots, g_n)$ with respect to the relation $R \times Id$, and let $c[]$ be a client program. Then $c[f_1, \ldots, f_n]$ simulates $c[g_1, \ldots, g_n]$ with respect to $R \times Id$.*

Proof. The proof is by induction on the structure of a client program c. Let s_1, s_2, s_2' be states such that $s_1[R \times Id]s_2$ and $c[f_1, \ldots, f_n], s_2 \rightsquigarrow s_2'$.

Let $c \equiv x_c = e$. By the semantics of the command $x_c = e$, we have that $s_2 = s_1[x_c \mapsto e]$. If we run $x_c = e[g_1, \ldots, g_n]$ in state s_1, the output state is $s_1' = s_1[x_c \mapsto e]$, and it is related to s_2' by $R \times Id$, because $s_1[R \times Id]s_2$ and in both states s_1 and s_2 after running the command, client variable x is set to value e, which maintains the identity relation for the client parts. The module parts are unchanged. So, $s_1'[R \times Id]s_2'$.

If $c \equiv f_i$, then by the assumption of the theorem, there exists a state s_1', such that $f_i, s_1 \rightsquigarrow s_1'$ and $s_1'[R \times Id]s_2'$.

Let $c \equiv c_1; c_2$. By the semantics of a command $c_1; c_2[f_1, \ldots, f_n]$ there exists a state s_2''

such that $c_1[f_1,\dots,f_n],s_2 \rightsquigarrow s_2''$ and $c_2[f_1,\dots,f_n],s_2'' \rightsquigarrow s_2'$. Then, by the induction hypothesis, we have that there exists a state s_1'' such that $c_1[g_1,\dots,g_n],s_1 \rightsquigarrow s_1''$ and $s_1''[R\times Id]s_2''$. Applying the induction hypothesis once again, we have that there exists a state s_1' such that $c_2[g_1,\dots,g_n],s_1'' \rightsquigarrow s_1'$ and $s_1'[R\times Id]s_2'$.

Let $c \equiv \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2$. Depending on the value of e in states s_1 and s_2 (the value of e is the same in these two states because the client parts on which the expression depends are related by the identity relation), the theorem follows by the induction hypothesis for c_1, i.e. c_2.

Let $c \equiv \textbf{while}\ e\ \textbf{do}\ c$. We do the inner induction on the length of the computation. If $s_2(e) = 0$, then also $s_1(e) = 0$ and by the semantics of the **while**-statement, we have that

$$\textbf{while}\ e\ \textbf{do}\ c[g_1,\dots,g_n],s_1 \rightsquigarrow s_1 \text{ and } \textbf{while}\ e\ \textbf{do}\ c[f_1,\dots,f_n],s_2 \rightsquigarrow s_2$$

and they are related by $R\times Id$. Now suppose that the theorem is true for all the computations of length n, and let $\textbf{while}\ e\ \textbf{do}\ c[f_1,\dots,f_n],\ s_2 \rightsquigarrow s_2'$ be a computation of length $n+1$. Then, by the semantics of the **while**-loop, we have that there exists a state s_2'' such that $c[f_1,\dots,f_n],s_2 \rightsquigarrow s_2''$ and $\textbf{while}\ e\ \textbf{do}\ c[f_1,\dots,f_n],s_2'' \rightsquigarrow s_2'$. By the outer induction hypothesis, there exists a state s_1'', such that $c[g_1,\dots,g_n],s_1 \rightsquigarrow s_1''$ and $s_1''[R\times Id]s_2''$. By the inner induction hypothesis, since the remaining computation is of length n, we have that there exists a state s_1', such that $\textbf{while}\ e\ \textbf{do}\ c[g_1,\dots,g_n],s_1'' \rightsquigarrow s_1'$ and $s_1'[R\times Id]s_2'$. □

We now illustrate the simulation method with the example we already looked at in the Introduction – the example of a data type Set, implemented by an array. Recall that the data type set has operations for insertion, removal and testing whether a particular element is in the set. For simplicity, it is assumed that the set has at most hundred elements. The concrete implementation is given in Table 1.1 and the abstract one in Table 1.2.

We now formally prove that the concrete array representation simulates the abstract specification of Set.

Lemma 1. *Array representation of Set, given in Table 1.1, simulates the abstract specification given in Table 1.2.*

Proof. First, we give a refinement relation. We say that two states s_1 and s_2 are related by refinement relation R

$$s_1[R]s_2 \iff \{s_2(a[i]) \mid 0 \leq i < s_2(m)\} = s_1(s).$$

We need to prove that each concrete operation simulates the corresponding abstract one, i.e. for instance,

$$\begin{tikzcd}
s_1 \arrow[r, dashed, "insert(n)"] \arrow[d, leftrightarrow, "R\times Id"'] & \exists s_1' \arrow[d, leftrightarrow, dashed, "R\times Id"] \\
s_2 \arrow[r, "insert(n)"'] & s_2'
\end{tikzcd}$$

For the insert() operation, if $s_1[R \times Id]s_2$, and we execute insert(n) on state s_2, there are two possible outcomes. One possibility is to get the same state with which we started – if n is already in the array, in which case because the initial states are related by $R \times Id$, n is also in s, and so the insert(n) also returns the initial state s_1. Therefore the output states are related as required. If on the other hand, n is not already in the array in state s_2 (it is not in the set s in state s_1 either), the set of values held in the array in output state is going to be $\{s_2(a[i]) \mid 0 \leq i < s_2(m)\} \cup \{n\}$ and that is equal to $s_1(s) \cup \{n\}$, which is exactly the output set of insert(n) when run on state s_1.

For the remove() operation, if we have $s_1[R \times Id]s_2$, and run remove(n) operation on s_2, there are again two possible outcomes. One possibility is to get the same state with which we started – if n is not in the array, in which case because the initial states are related by $R \times Id$, it is neither in set s, and so the remove(n) also returns the initial state s_1. Therefore the output states are related as required. If on the other hand, n is in the array in state s_2 (it is also in the set s in s_1), the set of values held in the array in output state is going to be $\{s_2(a[i]) \mid 0 \leq i < s_2(m)\} \setminus \{n\}$ and that is equal to $s_1(s) \setminus \{n\}$, which is exactly the output set of remove(n) when run on state s_1.

For the has() operation, note that the state is unchanged after running the operation, in both the abstract and concrete implementations of the set, and so, since the initial states are related, the output states have to be related, too. □

We have proved that the concrete array representation of Set simulates the abstract specification, and therefore we can apply the simulation theorem. We illustrate the simulation theorem with the following program.

```
main()
  {int i = 0, sum = 0;
  insert(2);
  insert(5);
  insert(14);
  while i ≤ 10 do
      if has(i) then
          sum = sum + i;}
```

Suppose that both the set and the array are initially empty. Then if we denote by $x \mapsto a$ a function which maps variable x into value a, then we have $[s \mapsto \emptyset][R \times Id][a \mapsto []]$, where [] denotes an empty array. After the three insert() commands the abstract state is $s_1 = [i \mapsto 0, sum \mapsto 0, s \mapsto \{2,5,14\}]$, and the concrete state is $s_2 = [i \mapsto 0, sum \mapsto 0, a \mapsto [2,5,14]]$ and clearly $s_1[R \times Id]s_2$. In concrete case, the state obtained by running a **while** statement is $s_2[i \mapsto 10, sum \mapsto 7]$, and in the abstract case it is $s_1[i \mapsto 10, sum \mapsto 7]$, so clearly, these two states are related by $R \times Id$.

2.3 Failure of Lifting without Variable Separation

The "lifting theorem" is an essential result for data refinement and holds for a simple while-language, as we have shown. The main assumption for the Lifting theorem to work is that the module variables are not directly visible to the client program and that the module state can only be changed through the provided module operations. However, if we slightly change the setting, the Lifting theorem fails. Namely, if we let the client assign also to the module variables, then the Lifting theorem no longer holds.

Let the client language be slightly different from the one given in Section 2.2, in that now the client may also assign to the module variables.

$$c \ ::= \ var_c = e \mid var_m = e \mid f_i, \ i = 1, \ldots, n \mid c;c \mid \textbf{if } e \textbf{ then } c \textbf{ else } c \mid \textbf{while } e \textbf{ do } c$$

The semantics of the language remains the same.

In Section 2.2 we have shown that the array representation of the data type Set simulates the abstract specification. We now construct a simple client program which shows

Table 2.2: New client of the data type Set

```
void main(); {
    insert(2);
    insert(17);
    insert(21);
    a[0] = 42;
}
```

that the Lifting theorem does not hold in the new setting.

Assuming that the set is initially empty, the refinement relation $R \times Id$ is satisfied before the execution of the program. If we run the client program with the concrete implementation of the data type Set, then after $\mathsf{insert}(21)$, the array contains elements $2, 17, 21$. If we run the program with the abstract implementation after these three statements the set also contains values $2, 17, 21$. These two output states are clearly related by $R \times Id$. However, after the statement $\mathsf{a}[0] = 42$, the array contains elements $42, 17, 21$, while the set in the abstract implementation still contains values $2, 17, 21$. Evidently, the concrete and the abstract implementations no longer represent the same set, and hence the Lifting theorem fails for this client program. The manifest failure of the Lifting theorem is a consequence of the client's ability to access directly the internal representation of the module. Of course, it is easy to rule out this problem with static checking. The point of this section, though, was to illustrate the *technical* reliance of lifting on separation, in a simple way.

2.4 Unscoped Data

We have described so far the problems that arise with the lifting theorem when the client program can directly access the variables of a module. It may seem that keeping the variables of the client and the module separate and not allowing the client to access the variables of the module is a good enough solution. But what about the unscoped data? Even if we limit the programming language in a way that it disallows allocation, deallocation and address arithmetic, there are certain problems that are triggered by the mere presence of unscoped data. As we will see, scoping is *not* a solution to these problems.

We illustrate the problems caused by unscoped data with a simple example. We assume a very simple memory model, which besides a stack contains exactly one memory location l. The stack part consists of a client stack and a module stack as in Section 2.2. The heap part assigns a value to the only location. We denote the set of all states by $\Sigma = S \times H$, and individual elements of this set by (s,h).

$$S_c = \mathsf{Var}_c \to \mathsf{Val}, \qquad S_m = \mathsf{Var}_m \to \mathsf{Val} \qquad \mathsf{Val} = \ldots,-1,0,1,\ldots$$
$$H = l \to \mathsf{Val},$$

The programming language is slightly changed compared to the one in Section 2.2. It now allows direct reading from memory with a lookup command $x_c = *y_c$ and direct writing to a memory with an update command $*x_c = e$. The other commands are unchanged.

$$\begin{aligned} c \ ::= \ & x_c = e \mid *x_c = e \mid x_c = *y_c \mid f_i,\ i = 1,\ldots,n \mid c;c \\ & \mid \textbf{if } e \textbf{ then } c \textbf{ else } c \mid \textbf{while } e \textbf{ do } c \end{aligned}$$

The refinement relation is a relation between the abstract module state and the concrete module state, just like before. However, to extend the refinement relation to the client state, the extension needs to encompass both the heap and the stack.

$$\begin{aligned} R \otimes Id \ = \ & \{((s_0 \uplus s_1, l \mapsto -),(s_0 \uplus s_2, l \mapsto -)) \mid (s_1, l \mapsto -)[R](s_2, l \mapsto -) \vee \\ & (s_1[R]s_2 \ \wedge \ \exists\, v.\ l \mapsto v)\} \end{aligned}$$

The extension of a refinement relation with identity relation ensures that the client parts of the states are identical. Then, one possibility is that location l is in both concrete and abstract client states and in both states has the same value. The other possibility is that location l is in both concrete and abstract module states, and then the values location l has in these two states have to conform to the relation R.

Let us now consider a "one-place buffer". The abstract implementation has variable b for keeping the current value of the buffer, and variable $full$ that indicates whether the buffer is full or empty. It also has a variable *junk* which points to a location l in which some unimportant or junk data is kept. The concrete implementation has only

Table 2.3: One place buffer

Abstract buffer	**Concrete buffer**
int full, b, ∗prev;	**int** full, ∗b;
int get();	**int** get();
{**if** full **then**	{**if** full **then**
full = 0; **return** b;	full = 0; **return** ∗ b;
else abort.}	**else abort**.}
void put(**int** v);	**void** put(**int** v);
{**if** ¬full **then**	{**if** ¬full **then**
b = v; full = 1;	∗b = v; full = 1;
else abort.	**else abort**.
∗**int** bad();	∗**int** bad();
{**return** junk.}	{**return** b.}

two variables $full$ and b. Variable b points to a location which holds the value of the buffer. There are three operations:

- get() – for retrieving the value from the buffer,
- put() – for placing a value into the buffer and
- bad() which returns a pointer to the client.

The definitions are given in Table 2.3.

We define the refinement relation for these two implementations.

$$(s_1, l \mapsto v_1)[R](s_2, l \mapsto v_2) \iff s_1(full) = s_2(full) \wedge (s_1(full) = 1 \Rightarrow s_1(b) = v_2)$$

The refinement relation requires that the values of the abstract (variable b) and the concrete (location l) buffers are the same.

Now we prove that all the concrete operations simulate the corresponding abstract ones. Let $(s_1, l \mapsto v_1), (s_2, l \mapsto v_2)$ and $(s'_2, l \mapsto v'_2)$ be states such that $(s_1, l \mapsto v_1)[R \otimes$

$Id](s_2, l \mapsto v_2)$ and $(s_2, l \mapsto v_2)[op](s_2, l \mapsto v_2')$.

$$\begin{tikzcd}(s_1, l \mapsto v_1) \arrow[d, leftrightarrow, "R\otimes Id"'] & \\ (s_2, l \mapsto v_2) \arrow[r, "op"'] & (s_2, l \mapsto v_2')\end{tikzcd}$$

Let $op \equiv \mathsf{get}()$. If $s_1(full) = \mathsf{false}$ then also $s_2(full) = \mathsf{false}$ and both the abstract and the concrete operations abort. If, on the other hand, $s_1(full) = s_2(full) = \mathsf{true}$ then $s_1(b) = v_2$ and so they return the same value to the client and both set the variable $full$ to false, leaving all the other variables and locations unchanged. Therefore, the output states are related by $R \times Id$.

Let $op \equiv \mathsf{put}(\mathsf{v})$. If $s_1(full) = \mathsf{true}$ then also $s_2(full) = \mathsf{true}$ and so, both the abstract and the concrete operation abort. If, on the other hand, $s_1(full) = s_2(full) = 0$ then $s_2' = s_2[full \mapsto 1]$ and $l \mapsto v$. If we run the abstract program the output state is $s_1' = s_1[b \mapsto v, full \mapsto 1]$. Clearly, these two states are related by $R \times Id$ if s_1 and s_2 are (and they are).

Let $op \equiv \mathsf{bad}()$. Remark that the states are unchanged in both the abstract and concrete case, so they are related by $R \otimes Id$.

Now consider the following client program.

```
main(){
  int *x;
  put(3);
  x = bad();
  *x = 5}
```

This program, after putting the value 3 into the buffer, calls method bad() and places the pointer returned by it into the variable x. It then assigns value 5 to the contents of the pointer variable x. In the abstract case this will have no effect on the buffer while in the concrete case it affects the internal representation of the buffer, and this imbalance reflects on the (non) preservation of the refinement relation. After a call to bad() and assigning 5 to the returned pointer, the concrete variable b points to a location l which now holds value 5, while the abstract variable b still holds value 3. But the relation R requires that these values be the same whenever the variable $full$ has value true.

With this simple example we have given a clear picture of how unscoped data messes up abstraction and we have intentionally chosen an uncluttered setting in order to do that. One can immediately think of real-life examples, which would certainly be harder to present but still would exhibit the same nuisance. The first thing that comes to mind is the infrastructure code – operating systems, database servers, network servers; these unavoidably utilize unscoped data. Another point we would like to make here is that the example shows that difficulties arise with unscoped data even when there is no allocation, deallocation and address arithmetic. These further features indisputably make scoping harder, but they are not the primary problem.

3

Separation Contexts

The traditional method for proving data refinement fails when certain real-world requirements are imposed on the programming language. For instance, the mere introduction of pointers to the programming language wreaks havoc [40]. Things like allocation, deallocation and address arithmetic make the situation even worse. Cross-boundary pointers, pointers held by client and which point into the internal state of the module, cause the main difficulty. With malicious use they may disclose the internal representation of the module, and that is unacceptable.

Separation logic [72, 41] enables us to check code of a client for safety, even if there are cross-boundary pointers. It ensures that pointers are not dereferenced at the wrong time and without permission. We take the first step towards bringing the ideas from separation logic into the filed of data refinement, by defining a notion of a *separation context*. A separation context is a client program that does *not* interfere with the module's internal representation. The notion of a separation context is semantic and not logical, although later in the thesis we make a connection to the logic. We also present the idea of *ownership* – program state can always be divided into two parts such that the module owns one part and the client owns the other. The ownership here is dynamic and changes from one step of program execution to another.

3.1 An Inkling of Separation Contexts

To present the reader with the basic intuition behind separation contexts, we give a detailed discussion based on examples. We consider two different client programs which interact with a memory manager module through the module operations. There are two module operations $\mathsf{alloc}()$ – for allocating new memory, and $\mathsf{free}()$ – for disposing allocated memory. The memory manager module maintains a list of locations available for allocation, where the disposed locations are also returned. One possible implementation, which we will assume here, is the linked-list representation given in Table 1.6.

The program state may be viewed as consisting of two parts – the client part and the module part. The module part of the state contains a *free list*, and the client part encompasses the remaining part of the program state. One might think then that module operations change only the module part of the state and client operations change only the client part of the state. Neither of these two things is true in general. A module operation changes the whole program state, but in a way that will preserve the structure of the module's internal representation, in this case the free list, and that will satisfy the specification of the operation with respect to the client. For example, the operation $\mathsf{alloc}(x)$ takes a location from the free list and places it into variable x. This way both the module part and the client part of the state (the free list and the value of the variable x) are changed. In fact, client operations can also change both the module and the client part of the state. This is exactly the point where we want to focus our attention.

Consider the following client program which interacts with the memory manager module:

$$\mathsf{alloc(x)}; [\mathsf{x}] := 15; \mathsf{y} := [\mathsf{x}]; \mathsf{free(x)}; [\mathsf{x}] := 1.$$

This simple program allocates a new location and saves its address in variable x. After manipulating and deallocating it, command $[x] := 1$ dereferences the variable. This last statement will actually change the module part of the state, because after disposing the location pointed to by x, variable x still holds the address of the location, except that the location again becomes part of the free list, i.e. module part of the state, when it is disposed. The aspect of being in the "client" part or in the "module" part of the state, we call *ownership*, and dragging locations from one part to another *ownership transfer*.

Depending on the implementation of the memory manager module, dereferencing the location that is in the free list may lead to destruction of the internal structure of the free list and consequently crashing of the program.

Ownership transfer may be seen as shifting the boundary between the client and module parts of the state. This concept is of dynamic nature, rather than static. Clearly, the partition between the module and the client cannot be made in advance, before executing the program; it changes from state to state throughout the execution of the program. At any point of execution, the program state is being partitioned into two parts, and ownership is just membership in one or the other component of the separated states.

As we have seen in the above example, the given client program does not respect the separation between the client and the module; it illegally changes the module state, and potentially causes problems. Client operations, in a client program which respects the separation between the client and the module, should *only* access the client part of the state. We call such programs *separation contexts*. The above example is not a separation context. Note also that the client program can use the module operations in a wrong way.

$$\mathsf{alloc(x);[x] := 15;y := [x];free(x);free(x)}.$$

This client program disposes twice location held in variable x, and this will have a similar effect on the execution of the program as in the above example.

In contrast, the following code obeys separation: the client code reads and writes to its own part, and disposes a location which belongs to it.

$$\mathsf{alloc}(x);[x] := 15;y := [x];\mathsf{free}(x)$$

The last remaining question is: how do we split the state? To describe the internal representation of the module we use a special kind of relations, the so called *precise predicates* [58, 73]. They give us means to delineate the portion of the state that is owned by the module, and hence they induce the unique splitting between the client's and the module's parts of the state. As already mentioned, this splitting changes throughout the execution of the program, new locations are allocated and some are disposed. Using precise predicates, we can track all of these changes in each step of the execution.

3.2 Setting the Stage

This section provides the foundations on which our theory is based. We introduce the storage model and present some basic assertions of separation logic which will be used in the development of our theory, but mainly in our examples. We formalize the notion of a module using precise relations and *general local actions*, which also play an important role in defining the meaning of the client language.

3.2.1 The Storage Model

Our storage model extends the one used in the traditional setting, where the variables are explicitely divided into module variables and client variables. The state is enriched with the heap component, which represents the memory that can be directly accessed (pointers).

We consider a general storage model which we employ to conduct the results of our theory. Its instantiation, a concrete storage model, will be assumed in our examples, unless stated differently.

General storage model.

We define the storage model in an abstract way, which will allow various realizations of a program state. We assume that we are given two disjoint countably infinite sets of variables – client variables Var_c and module variables Var_m. Let S_c denote a set of all maps from client variables to values, and similary let S_m denote a set of all maps from module variables to values. Then the set of all stacks S is defined to be a set of all combinations of module and client stacks.

$$S_c = \mathsf{Var}_c \rightarrow \mathsf{Val} \qquad S_m = \mathsf{Var}_m \rightarrow \mathsf{Val}$$

$$S = \{s_c \uplus s_m \mid s_c \in S_c \ \wedge \ s_m \in S_m\}.$$

We denote the individual elements of set S by s.

Let H be a set of *heaps* which is a set equipped with a structure of a partial commutative monoid $(H, \cdot, e)$. In effect, our development is on the level of the abstract model theory of BI [57], rather than the single model used in separation logic [41, 72]. The unit e denotes the empty heap and for any heap h satisfies the laws of a neutral element with respect to the operation $\cdot$. We will usually refer to e as an *empty heap*. Partial operation $\cdot \subseteq H \times H \rightharpoonup H$ is associative and commutative.

1. $e \cdot h = h \cdot e = h$ (unit)

2. $h_1 \cdot (h_2 \cdot h_3) = (h_1 \cdot h_2) \cdot h_3$ (associativity)

3. $h_1 \cdot h_2 = h_2 \cdot h_1$ (commutativity)

Here, the equality means that both sides are either undefined, or they are both defined and equal.

We assume that the operation $\cdot$ is injective in the sense that for each heap h, partial function $h \cdot - : H \rightharpoonup H$ is injective. The injectiveness of the $\cdot$ is an important requirement and the consequences that it has for our theory will be explained later on.

The subheap order $\sqsubseteq$ is induced by $\cdot$ in the following way

$$h_1 \sqsubseteq h_2 \iff \exists h_3.\ h_1 \cdot h_3 \downarrow \ \wedge\ h_2 = h_1 \cdot h_3.$$

Here $\downarrow$ denotes the fact that the composition $h_1 \cdot h_3$ is defined. Two heaps h_1 and h_2 are called disjoint, denoted $h_1 \# h_2$, if $h_1 \cdot h_2$ is defined.

The RAM Model

The Random Access Memory model can be obtained by instantiating the general storage model in the following way. Let H be the set of finite partial functions from the set of addresses Ptr to the set of values Val. The set of addresses is a subset of the set of positive integers; this allows address arithmetic.

$$H = \mathsf{Ptr} \rightharpoonup_{\mathit{fin}} \mathsf{Val}, \text{ where } \mathsf{Ptr} \subseteq \{0,1,2,\ldots\} \quad \text{and} \quad \mathsf{Val} \subseteq \{\ldots,-1,0,1,\ldots\}.$$

We now define the operation $\cdot$, making sure that all the requirements of the general storage model are respected. We say that two heaps are disjoint $h \# h'$ if their domains are disjoint, i.e. $\mathsf{dom}(h) \cap \mathsf{dom}(h') = \emptyset$. The combination $h \cdot h'$ of two heaps is defined only when they have disjoint domains

$$h \cdot h'(a) = \begin{cases} h(a), & a \in \mathsf{dom}(h), \\ h'(a), & a \in \mathsf{dom}(h'). \\ \text{undefined}, & \text{otherwise} \end{cases}$$

When $h \# h'$ fails, we stipulate that $h \cdot h'$ is undefined.

It can be easily seen that the RAM model is a partial commutative monoid with the injective operation.

Note. For proving our formal results, we will assume that the underlying storage model is the general one. Then all the results also apply to the RAM model, and since it is much closer to intuition, for most of our examples we will assume it as the underlying storage model.

In general model (and hence also in RAM model), we denote by Σ the set of all *states*, i.e. a set of all pairs of stacks and heaps $\Sigma = S \times H$. Then, elements of set Σ are pairs (s,h) which consist of the stack and the heap component. If we want to be explicit about the module and client parts of the stack, we will use s_c or s_m for client and module part of the stack, instead of s.

Remark. There are other ways in which we could have set up our storage model. One possible way would be not to divide the client and module variables, but to define stack as partial map and extend operation $\cdot$ from heap to the whole state, treating the stack in a similar way as a heap. But this is unnecessary and introduces a problem known as "variables as resources", which was recently resolved in [62]. We chose our model because, apart from being simple enough, it is also a natural extension with heaps of the storage model used in traditional setting.

3.2.2 The assertion language of Separation logic

Separation logic [41, 72] is a model of Logic of Bunched Implications [56], designed with an assumption that RAM is the underlying storage model. It is an extension of Hoare logic [34], whose description can be found in any of the textbooks [70, 78, 75] and so we assume the reader is familiar with it. The usual assertion language of Hoare logic is extended with assertions that express properties about heaps.

$$\begin{array}{rll} P,Q,R ::= & B \mid E_1 \mapsto E_2 & \text{Atomic formulae} \\ & \mid \mathsf{false} \mid P \Rightarrow Q \mid \forall x.P & \text{Classical Logic} \\ & \mid \mathsf{emp} \mid P * Q \mid \mathsf{true} & \text{Spatial assertions.} \end{array}$$

Assuming

$$[\![E]\!]_s \in \mathsf{Int} \qquad [\![B]\!]_s \in \{\mathsf{true}, \mathsf{false}\},$$

where

$$[\![-]\!]_s : S \to \mathsf{Val}, \qquad \mathsf{Val} = \{\ldots, -1, 0, 1, \ldots\},$$

the semantics of the assertion language is given bellow.

$(s,h) \models B$	iff	$[\|B\|]_s = \mathsf{true}$
$(s,h) \models E \mapsto F$	iff	$[\|E\|]_s = \mathsf{dom}(h)$ and $h([\|E\|]_s) = [\|F\|]_s$
$(s,h) \models \mathsf{false}$		never
$(s,h) \models P \Rightarrow Q$	iff	if $(s,h) \models P$ then $(s,h) \models Q$
$(s,h) \models \forall x.P$	iff	$\forall v \in \mathsf{Int}.\ (s[x \mapsto v],h) \models P$
$(s,h) \models emp$	iff	$h = e$ is the empty heap
$(s,h) \models P * Q$	iff	$\exists h_0,h_1.\ h_0\#h_1,\ h = h_0 \cdot h_1,\ (s,h_0) \models P$ and $(s,h_1) \models Q$

Here, $s[x \mapsto v]$ denotes the same stack as s, except that it assigns value v to variable x. We will write similarly for any function $f[l \mapsto v]$ to denote the fact that it is the same as f, except that it assigns value v to l.

The interpretation of assertion $E \mapsto F$ asserts that the heap component of the state contains exactly one location with address E and the contents of that location is F. The assertion emp says that the heap component of the state is empty, while $P * Q$ asserts that the state can be split into two disjoint parts, such that in one part P holds and in the other Q holds. This assertion is called the *separating conjunction*. Other assertions have standard interpretation.

3.2.3 Precise relations

In developing our theory, we will make extensive use of a special kind of unary relations defined on states – *precise relations*. A precise relation unambiguously determines a portion of the state described by the relation – it gives a way to "pick out the relevant locations" [58, 73].

Definition 3 (Precise relation). *A relation $M \subseteq \Sigma$ is* precise *if for any state (s,h) there is at most one subheap $h_0 \sqsubseteq h$, such that $(s,h_0) \in M$.*

Predicates used to describe data structures are usually precise. Richard Bornat's approach to detecting the locations associated with a data structure by writing a formula or running a program which would pick out only those location, can be formalized by precise predicates. Bornat used this idea to provide spatial separation in traditional Hoare logic [16].

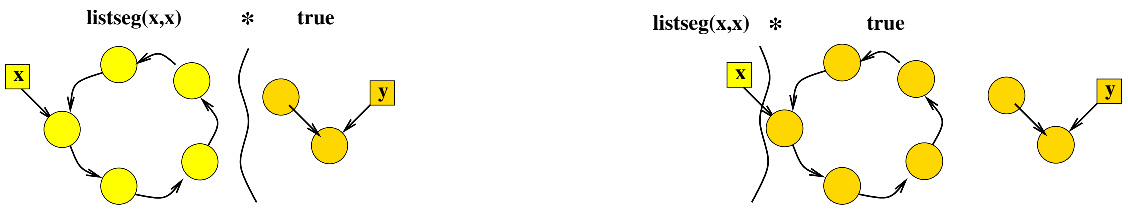


Figure 3.1: Circular list

We give several examples of precise predicates. Predicates emp, $E \mapsto E'$, $E \mapsto -$ are precise, $p * q$ is precise when both p and q are precise.

Consider the following predicate.

$$listseg\ (x, y) \overset{def}{\Longleftrightarrow} (x = y \ \wedge\ \mathsf{emp}) \ \vee\ (\exists z.\ x \mapsto z * listseg\ (z, y))$$

It is understood that this specifies the *least* predicate satisfying the recurrence. This is the only imprecise invariant that has been used in separation logic. This predicate allows existence of a cycle in a linked list segment from x to y. It is true in a heap which contains a non-empty circular list from x to x and nothing else, and it is also true of an empty heap, a proper subheap. However, even this predicate can be made precise, if it is restricted to forbid a cycle in the list segment [58].

$$lseg(x, y) \overset{def}{\Longleftrightarrow} (x = y \ \wedge\ \mathsf{emp}) \ \vee\ (x \neq y \ \wedge \exists z.\ x \mapsto z * lseg(z, y))$$

By adding the inequality $x \neq y$ in the second disjunct, the predicate $listseg(x, y)$ now requires that if x and y have the same value in the state, then the heap must be empty, and so the ambiguities introduced by allowing the cycles are prevented. Another commonly used imprecise predicate is true, and we usually use it to describe storage in which shape we are not particularly interested.

We define the *separating conjunction of unary relations* $M, M' \subseteq \Sigma$ by

$$M * M' = \{(s, h) \mid \exists h_0, h_1.\ h_0 \# h_1 \ \wedge\ h = h_0 \cdot h_1 \ \wedge\ (s, h_0) \in M \ \wedge\ (s, h_1) \in M'\}.$$

Taking into account that $\cdot$ is injective, if M is a precise relation, then it induces a unique splitting of the state.

Notation. For a precise relation $M \subseteq \Sigma$, we define h_M to be the unique subheap of a heap h, such that $h_M \in M$.

The uniqueness of the splitting induced by a precise relation is particularly important, since the nondeterminism of $*$ interacts badly with modularity, as already demonstrated in work with separation logic [58, 59]. For example, there are two possible splittings of the heap according to the predicate $listseg(x,x) * \mathsf{true}$. The splitting in the first picture in Figure 3.1 is possible because this splitting makes the second conjunct of the predicate $listseg(x,x)$ true. The second picture shows splitting which makes the first conjunct of the definition of the predicate $listseg(x,x)$ true.

However, this is not the case with the precise predicates. Given a precise predicate there is a unique splitting of the state.

3.3 The programming language

We introduce the programming language for the client programs and formally define the notion of a module. We also establish some basic facts about the denotations of the basic operations of the language and supply the semantics of the client programming language, which is, as the reader will see, parameterized by a given module.

3.3.1 The programming language syntax

Our model will use a simple language with two kinds of atomic operations: the client operations and the module operations.

The programming language is an extension of the simple while-language with a finite set of atomic client operations a_j $(j \in J)$ and a finite set of module operations f_i, $i \in I$. The syntax of the *user language* is given in Table 3.1.

We do not specify all the details of the language. For instance, we just say that expressions take values from the set Val, but we do not define what Val is in the abstract language. When the concrete implementation of the abstract language is decided, all the domains and the commands are appropriately instantiated. For example, we can instantiate the user language as in Table 3.2. The set Val is instantiated to be a subset of the integer values, and the atomic client operations can be set to be the standard commands for manipulating the state. Here, $x := E$ is standard assignment, command $[E_1] := E_2$ denotes a heap update, command $x := [E]$ denotes a heap lookup, and $\mathsf{alloc}()$ and $\mathsf{dispose}()$ are the operations for allocating and disposing memory, respectively. The

Table 3.1: The user language uninstantiated

$$
\begin{array}{rcll}
E & ::= & var \mid \ldots & [\![E]\!]_s \in \mathsf{Val}, var \in \mathsf{Var}_c, \\
B & ::= & \mathsf{false} \mid B \Rightarrow B \mid E = F \mid \ldots & [\![B]\!]_s \in \{\mathsf{true}, \mathsf{false}\} \\
c_{user} & ::= & \mathsf{a}_j,\ j \in J & \\
 & & \mid \mathsf{f}_i,\ i \in I & \\
 & & \mid c_{user}; c_{user} & \\
 & & \mid \mathbf{if}\ B\ \mathbf{then}\ c_{user}\ \mathbf{else}\ c_{user} & \\
 & & \mid \mathbf{while}\ B\ \mathbf{do}\ c_{users}, &
\end{array}
$$

$\mathsf{Var} = \{x, y, \ldots\}$, I, J – finite indexing sets.

module operations are instantiated when the module is defined. Note that this instantiation allows address arithmetic. In our examples we will use a concrete client language given in Table 3.2.

The language expressions do not access heap storage, i.e. they depend only upon the stack component of the state.

3.3.2 Local Actions

Before we introduce the semantics of our language, we present relations which have some special properties. In defining the semantics of the language we will require that the meaning of our commands satisfies these properties.

The commands of the programming language, apart from producing normal states in (s, h), are also able to produce a special state wrong. Intuitively, state wrong is produced when a command attempts to dereference a location that is not in the current state.

We now specify actions on states that access resources in a local way. A relation $r \subseteq \Sigma \times \Sigma \uplus \{\mathsf{wrong}\}$ is *local* [58] if it satisfies the following properties [81, 58]

- **Safety Monotonicity**: For all stacks s and heaps h and h_1 such that $h \# h_1$, if $\neg h[r]\mathsf{wrong}$, then $\neg h \cdot h_1[r]\mathsf{wrong}$.

- **Frame Property**: For all stacks s, s' and heaps h_0, h_1 and h' with $h_0 \# h_1$, if $\neg(s, h_0)[r]\mathsf{wrong}$ and $(s, h \cdot h_1)[r](s', h')$ then there is a heap $h'_0 \sqsubseteq h'$ such that $h'_0 \# h_1$, $h'_0 \cdot h_1 = h'$ and $(s, h_0)[r](s', h'_0)$.

Table 3.2: An instance of a user language

$$\begin{array}{rcl} E,F & ::= & \ldots \mid int \mid E+F \mid E\times F \mid E-F, \;\; int \in \mathsf{Int}, \\ B & ::= & \ldots \mid E \leq F \\ \mathsf{a} & ::= & x := E \\ & & \mid [E_1] := E_2 \\ & & \mid x := [E] \\ & & \mid \mathsf{alloc}() \\ & & \mid \mathsf{dispose}(x) \\ & & \mid f_i, \quad i \in I \end{array}$$

$\mathsf{Int} = \{\ldots -1, 0, 1, \ldots\}$, $\mathsf{Val} \subseteq \mathsf{Int}$

These two properties ensure that if heap h contains all the memory necessary for safe execution of the "command" r, then every computation from a bigger heap $h \cdot h_1$ is also safe. Moreover, such computation can be tracked from some computation from the smaller heap h. These properties ensure the soundness of the *Frame rule* from separation logic. All the commands that can be generated by the language given in Table 3.2 satisfy safety monotonicity and frame property. This is proved by O'Hearn and Yang in their paper on local reasoning [81].

In addition to safety monotonicity and frame property, we need our local actions to satisfy another property.

- **General Contents Independence**: For all stacks s, s' and heaps h_0, h_1, h'_0 if

$$h_1 \# h_0 \;\wedge\; \neg(s, h_0)[r]\mathsf{wrong} \;\wedge\; (s, h_0 \cdot h_1)[r](s', h'_0 \cdot h_1)$$

  then we have that for all h_2 such that $h_2 \sqsubseteq h_1$

$$(s, h_0 \cdot h_2)[r](s', h'_0 \cdot h_2)$$

The contents independence property expresses that if command r is safe if executed from heap h_0, i.e. it does not go wrong, the execution of r from a bigger heap $h_0 \cdot h_1$ does not look at the contents of heap h_1; it can only use the information that the heap memory in h_1 is allocated initially, and so heap h_1 can be replaced by any heap h_2 which contains at most all the heap memory of h_1. The contents independence, which is new, is necessary for the lifting theorem. 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. Note that $\mathsf{safe}(r, [])$ and $1 \notin \mathsf{dom}([])$. When we run "command" r in a state $[1 \mapsto 2]$ the output state is $[1 \mapsto 2]$. The contents independence property then insures that if we replace $[1 \mapsto 2]$ by another heap with the domain which is the subset of $\mathsf{dom}([1 \mapsto 2]) = \{1\}$, for instance $[1 \mapsto 3]$, the output state should be $[1 \mapsto 3]$, which is here not the case. The command r behaves differently depending on the contents of location 1.

Definition 4 (General Local Action)**.** *A* General local action*, in short* GLAct*, is an action that satisfies safety monotonicity, frame property and general contents independence.*

3.3.3 Modules

Here, we give a formal account of a module. Module comprises of a predicate that describes its resources and a collection of operations that preserve the predicate.

Definition 5 (Predicate preservation)**.** *The predicate $p \subseteq \Sigma$ is* preserved *by a command $r \subseteq \Sigma \times \Sigma$, if for all states (s,h) and (s',h') such that (s,h) satisfies predicate p, i.e. $(s,h) \in p$, and $\neg(s,h)[r]\mathsf{wrong}$ and $(s,h)[r](s',h')$ then $(s',h') \in p$.*

We denote by MOp the set of all module operations names. Let init and final be identifiers which are not in MOp.

Definition 6 (Module)**.** *A module is a pair (p, η), where $p \subseteq \Sigma$ is a precise predicate and $\eta : \mathsf{MOp} \cup \{\mathsf{init}, \mathsf{final}\} \longrightarrow \mathsf{GLAct}$, such that for all $f \in \mathsf{MOp}$*

1. $\forall (s,h), (s',h').\ ((\neg(s,h)[\eta(\mathsf{init})]\mathsf{wrong} \ \wedge \ (s,h)[\eta(\mathsf{init})](s',h')) \Rightarrow (s',h') \in p * \mathsf{true})$
2. $\forall (s,h), (s',h').\ ((\neg(s,h)[\eta(f)]\mathsf{wrong} \ \wedge \ (s,h) \in p * \mathsf{true} \ \wedge \ (s,h)[\eta(f)](s',h')) \Rightarrow (s',h') \in p * \mathsf{true})$.

Here, p denotes a predicate which describes a resource invariant of the module. Function η gives meaning to each module operation by assigning it a general local action, so it can be regarded as a module environment. In the two conditions of the definition, we allow the input and output states of operations to contain data other then the resource invariant of the module ($*\mathsf{true}$ part). The first condition then states that initialization operation builds an initial state of the module and the finalization operation gets rid of it. The second condition expresses that all operations of the module must preserve the module's resource invariant p.

The predicate p precisely describes the internal resources of the module; it mentions all the variables used by the module and outlines the shape of the data structure implemented by the module. Given a particular state, using precise relation p one can determine what portion of the state "belongs" to the module. This portion is unique, since only the module variables figure in p and the preciseness of the predicate ensures that there are no ambiguities as far as the heap is concerned. This means that in *every* step of computation, we can determine exactly which piece of the state is owned by the module; the remaining part of the state is owned by the user. Even if we had several modules interacting with the user program, the splitting of the state would still be unique, since each of the resource invariants of the modules is precise.

3.3.4 The semantics of the programming language

The denotation of both client and module commands will be given by binary relations $r : \Sigma \times \Sigma \uplus \{\mathsf{wrong}\}$.

For the semantics of all the operations in the client language as well as the modules operations we will consider only general local actions. In the uninitialized user programming language, we will assume that all the basic operations are general local actions. For the concrete commands we need to prove that. We give their semantics in Table 3.3.

Lemma 2. *All the operations of the concrete user programming language given in Table 3.2 are general local actions.*

Proof. All the commands satisfy safety monotonicity and frame property; the proof can be found in [81]. We only need to prove that all of them also satisfy general content independence.

Table 3.3: Semantics of the concrete basic operations

$(s,h)[x := E]v$	$\iff$	$\mathsf{v} = (\mathsf{s}',\mathsf{h}) \wedge \mathsf{s}' = \mathsf{s}[\mathsf{x} \mapsto [\![\mathsf{E}]\!]_\mathsf{s}]$
$(s,h)[[E_1] := E_2]v$	$\iff$	if $[\![\mathsf{E}_1]\!]_\mathsf{s} \notin \mathsf{dom}(\mathsf{h})$ then $\mathsf{v} = \mathsf{wrong}$ else $\mathsf{v} = (\mathsf{s},\mathsf{h}') \wedge$
		$h' = h[[\![E_1]\!]_s \mapsto [\![E_2]\!]_s]$
$(s,h)[x := [E]]v$	$\iff$	if$[\![\mathsf{E}]\!]_\mathsf{s} \notin \mathsf{dom}(\mathsf{h})$ then $\mathsf{v} = \mathsf{wrong}$ else
		$v = (s',h) \wedge s' = s[x \mapsto h([\![E]\!]_s)]$
$(s,h)[\mathsf{alloc}(x)]v$	$\iff$	$\exists n \notin \mathsf{dom}(h).\ h' = h \cdot n \mapsto - \wedge s' = s[x \mapsto n] \wedge v = (s',h')$
$(s,h)[\mathsf{dispose}(x)]v$	$\iff$	let $\mathsf{n} = \mathsf{s}(\mathsf{x})$ in if $\mathsf{n} \notin \mathsf{dom}(\mathsf{h})$ then $\mathsf{v} = \mathsf{wrong}$ else
		$v = (s,h') \wedge h = h' \cdot n \mapsto -$

Consider the update command $r \equiv [E_1] := E_2$, stacks s,s' and heaps h_0,h_0' and h_1, such that $\neg(s,h_0)[r]\mathsf{wrong}$ and $(s,h_0 \cdot h_1)[r](s',h_0' \cdot h_1)$. Then by the frame property and injectiveness of $\cdot$, it follows that $(s,h_0)[r](s,h_0')$. The definition of command r then implies that $h_0' = h_0[[\![E_1]\!] \mapsto [\![E_2]\!]]$. Let h_2 be any heap such that $h_2 \sqsubseteq h_1$. Then,

$$h_0' \cdot h_2 = h_0[[\![E_1]\!] \mapsto [\![E_2]\!]] \cdot h_2 = (h_0 \cdot h_2)[[\![E_1]\!] \mapsto [\![E_2]\!]]$$

and so, by the definition of command $[E_1] := E_2$, $(s,h_0 \cdot h_2)[[E_1] := E_2](s',h_0' \cdot h_2)$.

It can be proved in a similar fashion that all the other basic operations of the concrete language satisfy general content independence. □

Let (p,η) be a module. We give the operational semantics of a client programming language which interacts with the module (p,η). The $\leadsto_\eta \subseteq (c_{user} \times \Sigma) \times (\Sigma \uplus \{\mathsf{wrong}\})$ is the standard operational semantics and it is given in Table 3.4. When the client programming language is initialized, the meaning of the atomic client operation a_j is given using relation a_j. The meaning of the module operations f_i is provided by the module environment η. The operational semantics of all the compound statements is the standard big-step operational semantics.

The semantics $\leadsto_{(p,\eta)} \subseteq (c_{user} \times \Sigma) \times (\Sigma \uplus \{\mathsf{wrong},\mathsf{av}\})$ is the semantics parameterized by the module. Here, av denotes an *access violation*. A client program communicates with a module through the provided module operations and this is the only accepted manner in which they can exchange information. An access violation is encountered

Table 3.4: The language semantics

$$\frac{(s,h)[a_j](s',h')}{\mathsf{a}_j,(s,h) \rightsquigarrow_\alpha (s',h')} \qquad \frac{(s,h)[\eta(\mathsf{f}_i)](s',h')}{\mathsf{f}_i,(s,h) \rightsquigarrow_\alpha (s',h')} \qquad \frac{c_1,(s,h) \rightsquigarrow_\alpha (s',h') \quad c_2,(s',h') \rightsquigarrow_\alpha K}{c_1;c_2,(s,h) \rightsquigarrow_\alpha K}$$

$$\frac{(s,h)[a_j]\mathsf{wrong}}{\mathsf{a}_j,(s,h) \rightsquigarrow_\alpha \mathsf{wrong}} \qquad \frac{(s,h)[\eta(\mathsf{f}_i)]\mathsf{wrong}}{\mathsf{f}_i,(s,h) \rightsquigarrow_\alpha \mathsf{wrong}} \qquad \frac{[\![B]\!]_s = \mathsf{true} \quad c_1,(s,h) \rightsquigarrow_\alpha K}{\textbf{if } B \textbf{ then } c_1 \textbf{ else } c_2,(s,h) \rightsquigarrow_\alpha K}$$

$$\frac{c_1,(s,h) \rightsquigarrow_\alpha \mathsf{wrong}}{c_1;c_2,(s,h) \rightsquigarrow_\alpha \mathsf{wrong}} \qquad \frac{[\![B]\!]_s = \mathsf{false} \quad c_2,(s,h) \rightsquigarrow_\alpha K}{\textbf{if } B \textbf{ then } c_1 \textbf{ else } c_2,(s,h) \rightsquigarrow_\alpha K}$$

$$\frac{[\![B]\!]_s = \mathsf{true} \quad c;\textbf{while } B \textbf{ do } c,(s,h) \rightsquigarrow_\alpha K}{\textbf{while } B \textbf{ do } c,(s,h) \rightsquigarrow_\alpha K} \qquad \frac{[\![B]\!]_s = \mathsf{false}}{\textbf{while } B \textbf{ do } c,(s,h) \rightsquigarrow_\alpha (s,h)}$$

$K \in \Sigma \uplus \{\mathsf{wrong}\}, \qquad i \in I, j \in J, \qquad \mathsf{f}_i \in \mathsf{MOp} \qquad \alpha \in \{\eta,(p,\eta)\}$

when a client program improperly reads from or writes to a module part of the state. Semantics $\rightsquigarrow_{(p,\eta)}$ has all the rules given in Table 3.4 and one new rule

$$\frac{\neg(s,h)[a_j]\mathsf{wrong} \quad h = h_p \cdot h_u \quad (s,h_u)[a_j]\mathsf{wrong}}{\mathsf{a}_j,(s,h) \rightsquigarrow_{(p,\eta)} \mathsf{av}} .$$

There are several additional rules that emerge when we allow K in Table 3.4 to also take value av, that is, when $K \in \Sigma \uplus \{\mathsf{wrong}\} \uplus \{\mathsf{av}\}$.

The only difference between the standard semantics and the semantics parametrized by a module is that the standard one cannot detect that the client program has illegally accessed the internals of the module, while the other one can.

State $(s,h_p) \in p$ in the new rule of the parameterized semantics denotes the substate of (s,h) uniquely determined by predicate p and (s,h_u) denotes the rest of the state. The uniqueness of this splitting follows from the fact that p is precise and this enables us to determine which portion of the state belongs to whom. The stack part of the state is determined by s_m, that is the module part of the stack. We stress here the importance of the resource invariant of the module being precise. It enables us to correctly detect access violation. Suppose that the module resource invariant p is not necessarily precise. Then there would be ambiguities in splitting the state according to p, as shown in Section 3.2.3 and hence in determining which part of the state belongs to the module. That would unavoidably lead to many inconsistencies in meaning of the program. For instance, the semantics might report a faulty execution when there really isn't one, or conversely, the program might terminate properly, even if the execution is faulty.

Since we are dealing only with general local actions, we need to prove for the general language that all the commands that can be obtained in the general programming language are indeed general local actions.

Lemma 3. *All the commands that can be generated by the grammar of the general programming language are general local actions, provided that the atomic client and module operations are.*

Proof. The proof is conducted by induction on the structure of the command c.

If c is a client atomic operation or a module operation, then c is a general local action by the assumption of the lemma. Otherwise, let s, s' and h_0, h_0' and h_1 be such that $\neg(s,h_0)[c]\mathsf{wrong}$ and $(s,h_0\cdot h_1)[c](s',h_0'\cdot h_1)$.

For all other commands it is enough to prove that they preserve content independence, since its already known that they preserve frame property and safety monotonicity.

If c is a sequential composition of commands c_1 and c_2, then from $\neg(s,h_0)[c_1;c_2]$ wrong, we have that

$$\neg(s,h_0)[c_1]\mathsf{wrong}\ \wedge \tag{3.1}$$
$$(\forall s'',h_0''.\ (s,h_0)[c_1](s'',h_0'') \Rightarrow \neg(s'',h_0'')[c_2]\mathsf{wrong}). \tag{3.2}$$

From the definition of a sequential composition and the assumption $(s,h_0\cdot h_1)[c](s',h_0'\cdot h_1)$, we have that there exists a state (s'',h''), such that

$$(s,h_0\cdot h_1)[c_1](s'',h'')\ \wedge \tag{3.3}$$
$$(s'',h'')[c_2](s',h_0'\cdot h_1). \tag{3.4}$$

Using the fact that c_1 satisfies frame property and 3.1 and 3.3, we conclude that there exists a heap h_0'' such that

$$(s,h_0)[c_1](s'',h_0'')\ \wedge\ h''=h_0''\cdot h_1,\ \text{i.e.}(s,h_0\cdot h_1)[c_1](s'',h_0''\cdot h_1). \tag{3.5}$$

From 3.2 and 3.5 we get that $\neg(s'',h_0'')[c_2]\mathsf{wrong}$. So far we have:

$$\neg(s,h_0)[c_1]\mathsf{wrong}\ \wedge\ (s,h_0\cdot h_1)[c_1](s'',h_0''\cdot h_1) \tag{3.6}$$
$$\text{and}\ \neg(s'',h_0'')[c_2]\mathsf{wrong}\ \wedge\ (s'',h_0''\cdot h_1)[c_2](s',h_0'\cdot h_1). \tag{3.7}$$

We now exploit the fact that each of the commands c_1 and c_2 satisfy content independence. Let h_2 be any heap such that $h_2 \sqsubseteq h_1$. Then from 3.6 and 3.7, respectively, we have that

$$(s, h_0 \cdot h_2)[c_1](s'', h_0'' \cdot h_2) \quad \text{and} \quad (s'', h_0'' \cdot h_2)[c_2](s', h_0' \cdot h_2),$$

which by the definition of the sequential composition gives us

$$(s, h_0 \cdot h_2)[c_1; c_2](s', h_0' \cdot h_2).$$

Let c be **if** b **then** c_1 **else** c_2, and let s, s' and h_0, h_0' and h_1 be such that

$$\neg(s, h_0)[c]\mathsf{wrong} \qquad \text{and} \qquad (s, h_0 \cdot h_1)[c](s', h_0' \cdot h_1).$$

Suppose that $[\![b]\!]_s = \mathsf{true}$. Then, by the definition of the if-then-else statement,

$$\neg(s, h_0)[c_1]\mathsf{wrong} \qquad \text{and} \qquad (s, h_0 \cdot h_1)[c_1](s', h_0' \cdot h_1).$$

Similarly, when $[\![b]\!]_s = false$, we get that

$$\neg(s, h_0)[c_2]\mathsf{wrong} \qquad \text{and} \qquad (s, h_0 \cdot h_1)[c_2](s', h_0' \cdot h_1).$$

Let h_2 be an arbitrary state such that $h_2 \sqsubseteq h_1$. Then, since c_1 and c_2 both obey the content independence property, we have

$$[\![b]\!]_s = \mathsf{true} \implies (s, h_0 \cdot h_2)[c_1](s', h_0' \cdot h_2)$$
$$[\![b]\!]_s = \mathsf{false} \implies (s, h_0 \cdot h_2)[c_2](s', h_0' \cdot h_2)$$

Since $h_2 \sqsubseteq h_1$ was an arbitrary state, this is true for all such states, and hence, our command c satisfies the content independence property.

We leave out the while-case as it can be reasoned about in a similar fashion. □

We also owe an explanation for why we differentiate between the faulty states av and wrong. First reason is intuitive in nature. State wrong denotes a different kind of erroneous state in comparison to av. While using wrong we detect the dereferencing of unallocated memory, with av we detect illegal dereferencing of allocated memory. The other reason is more technical. Namely, if we had only one erroneous state, safety monotonicity property would fail for the most basic heap-manipulating operations. To see that, consider the following example. Suppose that the resource invariant of the module is given by a heap $[1 \mapsto 2 \cdot 2 \mapsto 3]$, and suppose that the current heap is $h = [1 \mapsto 2]$. Then the resource invariant is not satisfied in the current state, so the whole heap, h, must belong to the client and it is safe to perform a command $[1] := 42$, i.e. command $[1] := 42$ does not fault in state h. If we extend the current heap h with a disjoint heap $[2 \mapsto 3]$, the resource invariant becomes satisfied and command $[1] := 42$ now faults, which means that the heap-update command does not satisfy the safety monotonicity. Therefore, in accordance with the intuitive understanding, we keep av and wrong as two separate erroneous states.

3.4 Separation Contexts

An essential point in the parameterized semantics is the way in which module state is subtracted when client operations a_j are performed. As explained earlier, there are two kinds of faulty states: av and wrong. If a client program dereferences some location which is not allocated in the current state, then it will end up in a faulty state; this is reflected in the relation which provides the meaning of the client program. But even if the meaning of the client program does not relate the initial global (module + client) state to wrong, the faulty execution can still occur. In fact, the client may terminate normally when run in a global state, but go wrong when the module state is subtracted; in that case the client program also terminates in a faulty state, this time in av, because it attempted to access the module's state. A separation context is then a program (with a precondition) that does not lead to a faulty state.

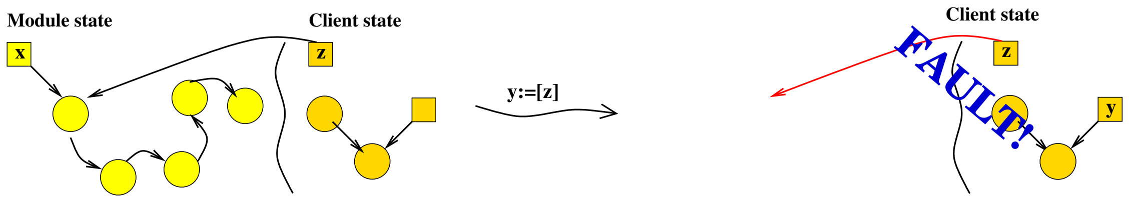


Figure 3.2: Client dereferencing module's state

For instance, Figure 3.2 gives a concrete picture of what happens if we run a command $y := [z]$ for heap lookup in an environment where a module, which maintains a list, is defined. Since variable z points inside the list, when we execute the command, the list-part of the state is subtracted, and in the user state variable z becomes a dangling pointer. When we try to read a dangling pointer, a fault is generated.

Note 1. *We denote by* $c[]$ *a program context. If* (p, η) *is a module, then* $c[(p, \eta)]$ *is a program obtained by filling the holes by the corresponding operations of the module, whose meaning is determined by function* η*. The meaning of program* $c[(p, \eta]$ *is then calculated in the semantics* $\rightsquigarrow_{(p,\eta)}$.

Definition 7. *Let* (p_0, η) *be a module and let* $p \subseteq \Sigma$ *be a unary predicate on states. A program* $c[(p_0, \eta)]$ *is a* separation context *for module* (p_0, η) *and precondition* p *if for all executions and all states* $(s,h) \in p * p_0$*,* $c,(s,h) \not\rightsquigarrow \mathsf{wrong}$ *and* $c,(s,h) \not\rightsquigarrow \mathsf{av}$.

A separation context is a client program with a precondition, obtained by filling the holes of the context with the corresponding operations of the module with which the program interacts, which never terminates in a faulty state. Precise unary relation p_0 describes the storage owned by the module – it is the resource invariant of the module. The semantics from which the meaning of the client program is obtained is parameterized by the module (p, η) and this enables us to observe the behavior of the client program with respect to the module. Another interesting aspect of separation contexts is that the behavior of the client program is observed only with respect to a certain, given precondition. We do not want to burden ourselves with an obligation to unnecessarily scrutinize all the behaviors of the client program from any possible state. We want to consider only those behaviors of a client program which are possible with respect to a given precondition, in the spirit of Hoare triples. A separation context is a client program which for a given set of states, i.e. a precondition, never accesses either the storage beyond the current state nor the storage of the module described by its resource invariant p. In particular, we want to explore the connection between the separation contexts and Hoare triples. We present the results regarding this matter in Chapter 5.

We now prove that given a module and a program, if a program is a separation context with respect to the module, then it *must* preserve the module's resource invariant.

Theorem 2. *Let* (p_0, η) *be a module, let* $p \subseteq \Sigma$ *be a unary predicate on states and let* c *be a separation context for* (p_0, η) *and* p*. Then for all such* p *and all states* (s,h) *and* (s',h')*, if* $(s,h) \in p * p_0$*, and* $c,(s,h) \rightsquigarrow (s',h')$*, then* $(s',h') \in (p_0 * \mathsf{true})$.

Proof. The proof is by induction on the structure of the command c.

Let p satisfy the conditions of the theorem, and let (s,h) satisfy $p * p_0$.

Let $c \equiv \mathsf{a}_j$. Suppose that $\mathsf{a}_j, (s,h) \rightsquigarrow (s',h')$, then we want to prove that $(s',h') \in p_0 * \mathsf{true}$. Relation p_0 is precise, which means that there is at most one $h_{p_0} \sqsubseteq h$ such that $(s,h_{p_0}) \in p_0$. Then there exists h_u such that $h = h_{p_0} \cdot h_u$ and $(s,h_u) \in p$. Since a_j is a separation context for (p_o, η) and p, then $\mathsf{a}_j, (s,h) \not\rightsquigarrow \mathsf{wrong}$ and $\mathsf{a}_j, (s,h) \not\rightsquigarrow \mathsf{av}$, which means $\neg(s,h_u)[a_j]\mathsf{wrong}$. By the assumption $\mathsf{a}_j, (s,h) \rightsquigarrow (s',h')$, i.e. $\mathsf{a}_j, (s, h_{p_0} \cdot h_u) \rightsquigarrow (s',h')$. Then, by the frame property which a_j obeys, there exists $h'_u \sqsubseteq h'$ and $h' = h_{p_0} \cdot h'_u$ and $(s,h_u)[a_j](s',h'_u)$. This means that $h_{p_0} \sqsubseteq h'$ and since $h_{p_0} \in p_0$, it follows that $h_{p_0} \cdot h'_u \in p_0 * \mathsf{true}$.

Consider the case when $c \equiv \eta(f)$, for some $f \in \mathsf{MOp}$. Then, as we assumed $(s,h) \in p_0 * p$, and by the definition of the module $\eta(f)$ preserves p_0, i.e. $(s',h') \in p_0 * \mathsf{true}$.

Suppose $c \equiv c_1;c_2$. From assumption that c is a separation context for (p_0, η) and p, we conclude that c_1 is also a separation context for (p_0, η) and p. Therefore, command c_1 satisfies the condition of the theorem and we can apply induction hypothesis to c_1. Let $sp(c,p)$ denote the strongest postcondition of a command c for the precondition p. If command c_1 produces a state (s',h') starting from (s,h), then $(s',h') \in sp(c_1, p_0 * p)$. From induction hypothesis, we know that (s',h') also satisfies $p_0 * \mathsf{true}$. This holds for all states $(s,h) \in p_0 * p$ and $(s',h') \in sp(c_1, p_0 * p)$, and so, it follows that $sp(c_1, p_0 * p)$ can be written as $p_0 * q$ for some predicate q. Command c_2 is a separation context for module (p_0, η) and precondition q. Suppose c_2 is not a separation context for (p_0, η) and $q = sp(c_1, p_0 * p)$. Then, c is not a separation context for (p_0, η) and p, and that contradicts assumption. We can apply the induction hypothesis again to c_2 and (p_0, η) and q, and conclude that the resulting state of c_2, which is also the resulting state of c, satisfies $p_0 * \mathsf{true}$.

Let $c \equiv$ **if** B **then** c_1 **else** c_2. This case follows directly from induction hypothesis; we just need to notice that, since c is a separation context for module (p_0, η) and precondition p, then c_1 must also be a separation context for (p_0, η) and $(p \wedge B = \mathsf{true})$. Dually, c_2 must be a separation context for (p_0, η) and $(p \wedge B = \mathsf{false})$.

For the **while** loop, we do an inner induction on the smallest derivation of **while** B **do** c', $(s,h) \rightsquigarrow (s',h')$. Suppose that for all derivations of length n if $(s,h) \in p_0 * \mathsf{true}$ and $c, (s,h) \rightsquigarrow$

(s',h') then $(s',h') \in p_0 * \mathsf{true}$. For $n = 0$, we have the inference rule

$$\frac{[\![B]\!]_s = \mathsf{false}}{\textbf{while } B \textbf{ do } c' \ (s,h) \rightsquigarrow (s,h)}$$

and as we know $(s,h) \in p_0 * p$, i.e. $(s,h) \in p_0 * \mathsf{true}$.

Now suppose that the length of the smallest derivation is $n+1$. Then the inference rule used is

$$\frac{[\![B]\!]_s = \mathsf{true} \quad c',(s,h) \rightsquigarrow (s'',h'') \quad \textbf{while } B \textbf{ do } c',(s'',h'') \rightsquigarrow (s',h')}{\textbf{while } B \textbf{ do } c',(s,h) \rightsquigarrow (s',h')}$$

By the outer induction hypothesis c' preserves the relation p_0. By the inner induction hypothesis $(s',h') \in p_0 * \mathsf{true}$, which completes the proof. □

A separation context with respect to the module (p_0,η) and the precondition p has a nice property that all its subcommands are separation contexts with respect to the corresponding preconditions. For example, if $c_1;c_2$ is a separation context for module (p_0,η) and precondition p, then so is c_1; subcommand c_2 is then a separation context for the same module and a precondition which is obtained as the strongest postcondition of c_1 for p. This observation was crucial in proving the compound statement cases of Theorem 2.

3.4.1 Separation Context Examples

We now revisit the ideas discussed in Section 3.1 in our more formal setting.

Greatest relation

In order to specify the operations of the memory manager module, we make use of the "greatest relation", for the specification $\{p\}\mathsf{f}\{q\}[X]$, which is the largest local relation satisfying a triple $\{p\} - \{q\}$ and changing only the variables in the set X. It is similar to the "generic commands" introduced by Schwarz [74] and the "specification statements" studied in the refinement literature [54], but adapted to work with locality conditions in [58].

More formally, for each specification $\{p\} - \{q\}[X]$, we define $\mathsf{great}(p,q,X)$

$$
\begin{aligned}
&(s,h)[\mathsf{great}(p,q,X)]\mathsf{wrong} \\
\overset{def}{\Longleftrightarrow}\ &(s,h) \notin p * \mathsf{true} \\
&(s,h)[\mathsf{great}(p,q,X)](s',h') \\
&\qquad (1)\ s(y) = s'(y) \text{ for all variables } y \notin X \text{ and} \\
\overset{def}{\Longleftrightarrow}\ &\qquad (2)\ \forall h_p, h_1.\ h_p \cdot h_1 = h\ \wedge\ (s,h_p) \in p \Longrightarrow \exists h'_q.\ h'_q \# h_1\ \wedge\ h'_q \cdot h_1 = h'\ \wedge \\
&\qquad\qquad (s',h'_q) \in q
\end{aligned}
$$

The first equivalence ensures that $\mathsf{great}(p,q,X)$ will run safely in a state (s,h) just when predicate p holds in some substate (s,h_p) of (s,h). The second equivalence defines how the greatest relation changes the state. The condition (1) makes sure that $\mathsf{great}(p,q,X)$ can modify only variables listed in X. Condition (2) defines $\mathsf{great}(p,q,X)$ to demonically dispose of the initial heap h_p which satisfies p and then, to angelically choose a heap h'_q from q and combine it with the remaining initial heap h_1 in order to get the final heap h'.

We define the predicate $\mathsf{list}(\alpha,\mathsf{ls})$. Let α be a sequence of integers. The predicate $\mathsf{list}(\alpha,x)$ is defined inductively on the sequence α by

$$
\mathsf{list}(\varepsilon,x) \overset{\text{def}}{=} x = nil \wedge \mathsf{emp}, \qquad \mathsf{list}(a\cdot\alpha,x) \overset{\text{def}}{=} x = a \wedge \exists y.\ x \mapsto y * \mathsf{list}(\alpha,y)
$$

where ε represents the empty sequence and $\cdot$ conses an element a onto the front of a sequence α. This predicate says that x points to a non-circular singly-linked list whose addresses are the the elements of the sequence α (this is called a "Bornat list" in [72]).

We use Bornat lists to define a memory manager module with resource invariant $\exists\alpha.\ \mathsf{list}(\alpha,\mathsf{ls})$ which denotes the free list and the operations $\mathsf{new}(x)$ and $\mathsf{dispose}(x)$. The operations $\mathsf{new}(x)$ and $\mathsf{dispose}(x)$ are defined as greatest relations satisfying the specifications given in Table 3.5. Operation $\mathsf{new}(x)$ takes the first element of the free list and assigns it to client variable x. If the free list is empty, the operation $\mathsf{cons}()$ is called which provides a fresh memory location. Operation $\mathsf{cons}()$ is the same as `malloc()`, except that it never fails, i.e. it always returns a location. Operation $\mathsf{dispose}(x)$ takes a location from the client variable x and returns it to the free list. The value of the client variable x stays unchanged, i.e. even though the location is now in the free list, variable x still points to it.

Table 3.5: Memory Manager Module

$$(\exists\alpha.\ \mathsf{list}(\alpha,\mathsf{ls}), \mathsf{new}(x), \mathsf{dispose}(x))$$

$$\begin{aligned}
\mathsf{new}_C(x): &\quad \{\mathsf{list}(a\cdot\alpha,\mathsf{ls})\} - \{\mathsf{list}(\alpha,\mathsf{ls}) * x \mapsto a\}[x,\mathsf{ls}] \\
&\quad \{\mathsf{list}(\varepsilon,\mathsf{ls})\} - \{\mathsf{list}(\varepsilon,\mathsf{ls}) * x \mapsto -\}[x,\mathsf{ls}] \\
\mathsf{dispose}_C(x): &\quad \{\mathsf{list}(\alpha,\mathsf{ls}) * x \mapsto a\} - \{\mathsf{list}(a\cdot\alpha,\mathsf{ls})\}[\mathsf{ls}]
\end{aligned}$$

```
$\mathsf{new}_C(x) \equiv$                      $\mathsf{dispose}_C(x) \equiv$
if $\mathsf{ls} \neq nil$ then                  if $\mathsf{ls} = nil$ then
    $x := \mathsf{ls}; \mathsf{ls} := \mathsf{ls.next}$          $\mathsf{ls} := x$
else                                            else
    $x := \mathsf{cons}()$                            $t := \mathsf{ls}; \mathsf{ls} := x; x. := t$
```

For future reference, we will call this the *concrete* interpretation of the memory manager module. With these definitions we can judge whether a program (together with a precondition) is a separation context.

Consider the following three programs

```
$\mathsf{program}_1$ :        $\mathsf{program}_2$ :        $\mathsf{program}_3$ :
  $\mathsf{new}(x)$;              $\mathsf{dispose}(x)$;          $[81] := 42$
  $[x] := 47$;                $[x] := 47$;
  $\mathsf{dispose}(x)$;
```

We indicate whether a program, together with a precondition, is a separation context in the following table.

Context	Separation context?
$\{\mathsf{emp}\}\ \mathsf{program}_1$	$\surd$
$\{x \mapsto -\}\ \mathsf{program}_2$	$\times$
$\{\mathsf{emp}\}\ \mathsf{program}_2$	$\times$
$\{81 \mapsto -\}\ \mathsf{program}_3$	$\surd$
$\{\mathsf{emp}\}\ \mathsf{program}_3$	$\times$

Most of the entries are easy to explain, and correspond to our informal discussion from earlier. The two separation contexts both access locations that are visible to them. For example, in the second-last entry the precondition $81 \mapsto -$ ensures that 81 is in the current client state. It cannot be in the free list, because of the use of $*$ to separate the module and client states. The last one, $[81] := 42$, is not a separation context because it either interferes with the free list or it dereferences a location beyond the current heap, but we do not know which of the two. It might (or might not) be the case that location 81 is in the free list, at any given point in time. We can easily construct an example state where 81 is indeed in the free list.

4

Forward Simulation and Data Refinement

Our main interest is proving data refinement between modules, i.e. showing that a certain implementation of some abstract data type is indeed its representation. We also need to assess modules' behavior in presence of the programs that are using their services, i.e. client programs. Once data refinement between two modules is proved, the behavior of the more concrete representation has to be at least as good as the behavior of the more abstract representation of the abstract data type. That allows client programs to use safely the concrete representation instead of the abstract one.

Forward simulation is one method for proving data refinement which we are going to study here. As we have argued and shown by examples so far, the traditional simulation method becomes unsound when pointers and manipulations with pointers are introduced into the programming language. We propose a new formal approach to the forward simulation method, which ensures its soundness even in presence of pointers.

4.1 An Informal Account of Simulation, Lifting and Data Refinement

Central aspects of data refinement are:

1. Small local relations connecting abstract and concrete modules representing the abstract data type.

2. Lifting small local relations from relating only modules to the whole programming language, i.e. the relations are extended to include a client program as well.

3. Soundness of the method.

The first of these three aspects of data refinement gives means for expressing the direct relationship between different representations of a certain abstract data type. Recall that the simulation method requires the abstract operation to be able to track the behavior of the concrete operation up to the refinement relation.

$$\begin{tikzcd}
(s_1,h_1) \arrow[r, dashed, "c_{\mathrm{abstract}}"] \arrow[d, leftrightarrow, "R{*}\mathsf{Id}"'] & \exists(s_1',h_1') \arrow[d, dashed, leftrightarrow, "R{*}\mathsf{Id}"] \\
(s_2,h_2) \arrow[r, "c_{\mathrm{concrete}}"'] & (s_2',h_2')
\end{tikzcd}$$

Usually this relationship can be established easily. The involved relation can always be made weak enough, so that the operations of the abstract and concrete modules preserve it. Even when the relation seems to be right and strong enough, it is not always clear that the lifting of the relation to the whole language can be performed. This presents the main difficulty with the simulation method. Having the relationship between different implementations of the abstract data type is useless without lifting. Lifting gives meaning to the simulation method; it expresses that the relationship between the pairs of the corresponding operations of the abstract and concrete modules, and hence the modules themselves, is maintained when they are actually being used in some computation.

In Chapter 2, we discussed the traditional simulation method and its shortcomings, how lifting fails when the pointers are introduced. Cross-boundary pointers cause the main difficulty because, when illegally dereferenced, they allow a client program to spot the difference between different implementations of the module and that breaks the abstraction.

In this chapter we give a new approach to the simulation method. The method is designed to take care only of the non-faulting executions of the program, where among the faulting executions is also illegal dereferencing of the cross-boundary pointers from the client to the module. This also gives rise to the correctness of the lifting theorem. Here, we give an informal statement of the lifting theorem.

Theorem 3 (Simulation theorem)**.** *If concrete module simulates the abstract one, then the simulation lifts to all separation contexts.*

The crucial assumption here is that the client program is a separation context. Recall that a separation context is a client program which neither dereferences the internals of the module, nor it dereferences memory that is not guaranteed to exist by the precondition. The lifting theorem is correct only for separation contexts. It fails for arbitrary contexts. To state more formally the theorem, we first need to define the simulation method, and introduce some additional notation and terminology.

We define a concrete module to be a data refinement of an abstract module, if none of the non-faulting executions of all programs can distinguish between them. The lifting theorem is also a first step in proving the soundness of the simulation method.

> **Theorem 4** (Soundness). *If a module forward simulates another module, it data refines it.*

In this chapter, we build our theory of data refinement. We prove the lifting theorem and the soundness of the forward simulation method.

4.2 Data refinement

So far, we have only intuitively described data refinement. We say that one module data refines another if in all computations the abstract module can be replaced by the concrete one. Here, we give a formal account of data refinement.

To define data refinement, we need to be able to compare behavior of the programs when they execute on the same state. To be able to do that, we wrap around each program the initialization and finalization operations of the module with which it interacts, to make it a *complete* program. We use the initialization operation of the module to build, starting from the initial client state, a state which also encompasses a module state. Similarly, we use the finalization operation to get rid of the module's sub-state and to enable the computation of a program to finish in a client state.

Definition 8 (Complete command). *For a given command c and module (p, η), a complete command $c_c\langle(p,\eta)\rangle$ is $\mathsf{init}_{(p,\eta)}; c; \mathsf{final}_{(p,\eta)}$.*

To conclude that one module data refines another, we need to check that for all complete commands, its execution in the concrete module environment can be mimicked by its execution in the abstract module environment. In other words, everything the concrete program can do, the abstract program can do as well, as long as it does not fault.

Definition 9 (Data refinement). *A module (q,μ)* data-refines *another module (p,η) iff for all complete commands c and all states (s,h), if $c\langle(p,\eta)\rangle$ does not generate an error from (s,h) (i.e., $c,(s,h) \not\rightarrow_{(p,\eta)} \mathsf{av} \wedge c,(s,h) \not\rightarrow_{(p,\eta)} \mathsf{wrong}$), then*

$$\big(c,(s,h) \not\rightarrow_{(q,\mu)} \mathsf{av} \ \wedge\ c,(s,h) \not\rightarrow_{(q,\varepsilon)} \mathsf{wrong}\big) \wedge$$
$$\big(\forall (s',h').\ c,(s,h) \rightsquigarrow_{(q,\varepsilon)} (s',h') \ \Rightarrow\ c,(s,h) \rightsquigarrow_{(p,\eta)} (s',h')\big).$$

4.3 Binary relations

The simulation method requires use of binary relations. In order for the simulation method to be sound, we need to narrow down the set of the binary relations to be used in the method.

We introduce a special kind of binary relations – *coupling* relations, which connect the internals of two different modules.

Definition 10. *A* coupling *relation $R \subseteq \Sigma \times \Sigma$ between modules (p,η) and (q,ε) is a binary relation such that*

$$(s_1,h_1)[R](s_2,h_2) \Longrightarrow (s_1,h_1) \in p \ \wedge\ (s_2,h_2) \in q.$$

We can lift the notion of preciseness from unary to binary relations.

Definition 11. *For a binary relation $R \subseteq \Sigma \times \Sigma$, we say that it is* precise, *if each of its two projections is a precise unary relation.*

Note that the coupling relation is always precise since both its projections, the resource invariants of the modules in question, are precise by the definition of a module.

We illustrate coupling relations with an example. Suppose we have two different implementations of a memory manager module. In the first implementation we assume that f is a set variable, which keeps track of all owned locations. In the second implementation, we let this information be kept in a list. We use the list predicate $\mathsf{list}(\alpha,\mathsf{ls})$, defined in Section 3.2.3. Now, a precise binary relation

$$R = \left\{ ((s,h),(s',h')) \;\middle|\; \begin{array}{c} ((s,h) \models \forall_* p \in f.\ \mathit{allocated}(p) \wedge ((s',h') \models \mathsf{list}(\alpha,\mathsf{ls})) \wedge \\ \mathit{set}(\alpha) = s(f) \end{array} \right\},$$

where $set(\alpha)$ is defined as the set of pointers in the sequence α, is a coupling relation between these two implementations. Here, $\forall_* p \in f.\ allocated(p)$ denotes a predicate $allocated(p_1) * \ldots * allocated(p_n)$, where $f = \{p_1, \ldots, p_n\}$, and $allocated(p)$ asserts that p is allocated in the current heap. Relation R relates pairs of states, such that the first state in the pair can be described as a set of different pointers, and the other state is determined by the list of exactly the pointers that appear in the mentioned set.

Coupling relation expresses a relationship between the "abstract" and the "concrete" module without regard for any external users of the module. As well as examining the internal behavior of the modules and preserving the relationship between the modules themselves, it is also important to observe their external behavior. Our refinement theory assumes an environment in which both module and client program figure, and hence, we also need to give means by which the relation between the client and the module in the process of refinement is expressed. The separating conjunction of binary relations enables us to move from reasoning about internals of the module to reasoning which also includes the externals of the module, whether it be a client program or even other modules participating in a computation.

For two binary relations $R, S \subseteq \Sigma \times \Sigma$ on states, we define their *separating conjunction* [67] as

$$R * S = \left\{ ((s_1, h_1), (s_2, h_2)) \;\middle|\; \begin{array}{c} \exists h_1', h_1'', h_2', h_2''.\ h_1 = h_1' \cdot h_1'' \wedge h_2 = h_2' \cdot h_2'' \wedge \\ (s_1, h_1')[R](s_2, h_2') \wedge (s_1, h_1'')[S](s_2, h_2'') \end{array} \right\}$$

The separating conjunction of binary relations empowers us to *expand* a coupling relation between two modules. For instance, for two modules (p, η) and (q, ε), we expand relation R between them to a relation $R * \mathsf{Id}$, where $\mathsf{Id} \subseteq \Sigma \times \Sigma$ denotes the identity relation on states. The R part of the relation $R * \mathsf{Id}$, which we will extensively use, imposes on modules' internals to be related by R. On the other hand, the Id part of it requires that the client program that is using the modules is the same. This means that the state of the client program does not change when we replace one module by another. We also use widely relation Δ_p , where $p \in \Sigma$ is a unary predicate on states. Relation Δ_p denotes the identity relation restricted to states satisfying predicate p, $\Delta_p = \mathsf{Id} \cap p \times p$.

4.4 Forward simulation and Simulation theorem

To prove that a concrete program simulates the abstract one, forward simulation works by ensuring that what ever step of computation the concrete program makes starting from some state, then starting from a related state the abstract program can perform the same step (up to the simulation relation).

Notation. We use the notation $c\langle\alpha\rangle$ to specify that the execution of program c is evaluated in semantics $\rightsquigarrow_\alpha$. For instance, $c\langle(p,\eta)\rangle$ denotes a program c which is evaluated in a semantics determined by the module (p,η). We use $\rightsquigarrow$ to specify that the computation is evaluated in standard semantics.

Definition 12 (Forward simulation)**.** *Operation b simulates another operation a, denoted by $b\langle\beta\rangle[\mathsf{fsim}(R_0,R_1)]a\langle\alpha\rangle$ iff for all states (s_1,h_1) and (s_2,h_2),*

1. *if $((s_1,h_1)[R_0](s_2,h_2)\ \wedge\ (b,(s_2,h_2)\rightsquigarrow_\beta \mathsf{wrong}\ \vee\ b,(s_2,h_2)\rightsquigarrow_\beta \mathsf{av}))$, then $a,(s_1,h_1)\rightsquigarrow_\alpha \mathsf{wrong}\ \vee\ a,(s_1,h_1)\rightsquigarrow_\alpha \mathsf{av}$, and*

2. *if $((s_1,h_1)[R_0](s_2,h_2)\wedge a,(s_1,h_1)\not\rightsquigarrow_\alpha \mathsf{wrong}\wedge a,(s_1,h_1)\not\rightsquigarrow_\alpha \mathsf{av}\wedge b,(s_2,h_2)\rightsquigarrow_\beta (s_2',h_2'))$ then there exists (s_1',h_1') such that*

$$a,(s_1,h_1)\rightsquigarrow_\alpha (s_1',h_1')\wedge(s_1',h_1')[R_1](s_2',h_2'),$$

where $\rightsquigarrow_\alpha$ and $\rightsquigarrow_\beta$ determine either standard or parameterized semantics and can be the same.

The following diagrams can help when thinking of the two conditions of the definition 12.

$$\begin{tikzcd}
(s_1,h_1) \arrow[r, dashed, "a", "\alpha"'] \arrow[d, leftrightarrow, "R_0"'] & \mathsf{wrong}\ (\mathsf{av}) \arrow[d, dashed, leftrightarrow, "R_1"] \\
(s_2,h_2) \arrow[r, "b"', "\beta"] & \mathsf{wrong}\ (\mathsf{av})
\end{tikzcd}
\qquad
\begin{tikzcd}
(s_1,h_1) \arrow[r, dashed, "a", "\alpha"'] \arrow[d, leftrightarrow, "R_0"'] & (s_1',h_1') \arrow[d, dashed, leftrightarrow, "R_1"] \\
(s_2,h_2) \arrow[r, "b"', "\beta"] & (s_2',h_2')
\end{tikzcd}$$

If $R_0=R_1=R$, we write $\mathsf{fsim}(R)$.

Remark. In our method we specify the semantics on each level of simulation. This is due to the fact that we usually deal with two different semantics on two different levels of abstraction. Actually, the semantics is the same, it is only parameterized in two different ways. The semantics can either be standard or it can be parameterized by a module. If we want to consider simulation between two different modules (p,η) and (q,μ), then we need to allow for a program to be executed in semantics $\rightsquigarrow_{(p,\eta)}$ on one level (say abstract) and in semantics $\rightsquigarrow_{(q,\mu)}$ on the other (say concrete).

Now that simulation between two operations is defined, we can state the conditions under which one module is considered to be simulated by another.

Definition 13. *Module (q,μ) simulates module (p,η) if and only if there exists a relation $R \subseteq p \times q$, such that for all $f \in \mathsf{MOp} \cup \{\mathsf{init}\} \cup \{\mathsf{final}\}$, $f\langle(q,\mu)\rangle[\mathsf{fsim}(R * \mathsf{Id})]f\langle(p,\eta)\rangle$.*

Note that to have a concrete module simulate the abstract one, they need to have the same set of operations and in each pair of corresponding operations, the concrete operation has to simulate the abstract one. Important here is also that, apart from preserving the module simulation relation, the module operations need to preserve the identity relation on the external part of the state – the client state.

One of the causes of unsoundness of the traditional forward simulation method is that for some client operations we have the situation where an operation fails to simulate itself. Namely, if we run the client operation in two different module environments, even when the modules simulate one another, the client operation does not preserve the refinement relation. This is especially true for allocation. Refinement relation preservation by all the client operations is crucial for lifting. To prevent this complication we need to impose certain restrictions on the simulation relation. In order for forward simulation to be sound with respect to data refinement, we have to make sure that for the related pairs of states, the abstract state is "smaller" then the concrete one. To see why a bigger abstract state in a pair of related states might cause problems, consider the following example.

Let simulation relation R contain a pair $([2 \mapsto 3], [])$. Here, for simplicity, we assume that the states consist only of heaps. The abstract state contains exactly one location with address 2 and contents 3, and the concrete state is empty. Suppose now that we have two states related by $R * \mathsf{Id}$, $h_1 = [1 \mapsto 2, 2 \mapsto 3]$ and $h_2 = [1 \mapsto 2]$, and we run the alloc

operation on the concrete state.

$$\begin{tikzcd}
h_1 \arrow[r, dashed, "\mathsf{alloc}"] \arrow[d, leftrightarrow, "R*\mathsf{Id}"'] & ??? \arrow[d, dashed, leftrightarrow, "R*\mathsf{Id}"] \\
h_2 \arrow[r, "\mathsf{alloc}"'] & h_2 \cdot [2 \mapsto 3]
\end{tikzcd}$$

The operation alloc executed on state h_2 might return location 2, as it is available in the state h_2. However, there is no such location that can be returned by alloc when executed in state h_1 and relation $R * \mathsf{Id}$ be reestablished at the same time. This example justifies our need to restrict the set of simulation relations to only *growing* relations.

Now we define more formally what it means for a relation to be growing.

Definition 14. *A relation $R \subseteq \Sigma \times \Sigma$ is growing if and only if for all states (s,h) and (s',h')*

$$(s,h)[R](s',h') \Longrightarrow (\forall h''.\ h''\#h' \Rightarrow h''\#h).$$

Intuitively, R relates the internal states of two modules, and the above condition means that one module uses more memory than the other. For example, in RAM model, the condition for growing relations boils down to

$$(s,h)[R](s',h') \Longrightarrow \mathsf{dom}(h) \subseteq \mathsf{dom}(h').$$

Note that when the identity relation Id is composed by separating conjunction with a growing relation R, the resulting relation $R * \mathsf{Id}$ is also growing.

Having put the restrictions on the simulation relation, every client operation now simulates itself.

Lemma 4. *Let R be a coupling, growing relation between the modules (p,η) and (q,μ). Then, for all client operations a, we have that*

$$a\langle(q,\mu)\rangle[\mathsf{fsim}(R * \mathsf{Id})]a\langle(p,\eta)\rangle.$$

Proof. Let (s_1,h_1) and (s_2,h_2) be two arbitrary states such that $(s_1,h_1)[R * \mathsf{Id}](s_2,h_2)$. States (s_1,h_1) and (s_2,h_2) can be written as $(s_1,h_1) = (s_c \uplus s_{1m}, h_c \cdot h_p)$ [1] and $(s_2,h_2) = (s_c \uplus s_{2m}, h_q \cdot$

[1]Recall that the stacks consist of two parts: the client part s_c and the module part s_m

$h_c)$, for some heaps h_p, h_q, h_c such that $(s_{1m}, h_p)[R](s_{2m}, h_q)$. We prove that the two conditions from the definition of simulation are satisfied.

To prove the first condition, suppose that $a, (s_2, h_2) \rightsquigarrow_{(q,\mu)} \mathsf{wrong}$ or $a, (s_2, h_2) \rightsquigarrow_{(q,\mu)} \mathsf{av}$. We show that then $a, (s_c, h_c) \rightsquigarrow_{(q,\mu)} \mathsf{wrong}$, which will give us the desired result by the following reasoning. By the rules of semantics $\rightsquigarrow_{(q,\mu)}$, the derivation $a, (s_c, h_c) \rightsquigarrow_{(q,\mu)} \mathsf{wrong}$ is possible only when $(s_c, h_c)[a]\mathsf{wrong}$. But then, rules of semantics $\rightsquigarrow_{(p,\eta)}$ also yield $a, (s_c, h_c) \rightsquigarrow_{(p,\eta)} \mathsf{wrong}$, and hence

$$a, (s_1, h_1) \rightsquigarrow_{(p,\eta)} \mathsf{wrong} \qquad \text{or} \qquad a, (s_1, h_1) \rightsquigarrow_{(p,\eta)} \mathsf{av},$$

and that is exactly what we want to show. To get there, note that derivation $a, (s_2, h_2) \rightsquigarrow_{(q,\mu)} \mathsf{wrong}$ is possible only when $(s_2, h_2)[a]\mathsf{wrong}$, and then by the safety monotonicity for a, $(s_2, h_c)[a]\mathsf{wrong}$, and since the client operations are accessing only client variables, $(s_c, h_c)[a]\mathsf{wrong}$. If $a, (s_2, h_2) \rightsquigarrow_{(q,\mu)} \mathsf{av}$, we have that $(s_c, h_c)[a]\mathsf{wrong}$ immediately.

For the second condition, suppose that

$$a, (s_1, h_1) \not\rightsquigarrow_{(p,\eta)} \mathsf{wrong} \wedge a, (s_1, h_1) \not\rightsquigarrow_{(p,\eta)} \mathsf{av} \wedge a, (s_2, h_2) \rightsquigarrow_{(q,\mu)} (s_2', h_2').$$

Then, by what we have just shown,

$$a, (s_2, h_2) \not\rightsquigarrow_{(q,\mu)} \mathsf{wrong} \wedge a, (s_2, h_2) \not\rightsquigarrow_{(q,\mu)} \mathsf{av},$$

i.e.

$$a, (s_c, h_c) \not\rightsquigarrow_{(q,\mu)} \mathsf{wrong}.$$

We can apply the frame property for a, and so there exists a heap h_c' such that $h_2' = h_c' \cdot h_q$ and $a, (s_c, h_c) \rightsquigarrow_{(q,\mu)} (s_c', h_c')$, where s_c' is the client part of the stack s_2'.

Let $(s_1', h_1') = (s_c' \uplus s_{1m}, h_c' \cdot h_p)$. We claim this is the state we are looking for in order for the second condition of simulation to be fulfilled. Firstly, state (s_1', h_1') is defined. R is growing, so for all heaps h, if $h \# h_q$ then $h \# h_p$ and since we already know that $h_c' \# h_q$, it follows that $h_c' \cdot h_p$ is defined. Secondly, states (s_1', h_1') and (s_2', h_2') are related by $R * \mathsf{Id}$. Finally, we need to show that $a, (s_1, h_1) \rightsquigarrow_{(p,\eta)} (s_1', h_1')$. The following things hold, by what we have proved so far.

1. $(s_c, h_c)[a](s_c', h_c')$

2. $\neg(s_c, h_c)[a]\mathsf{wrong}$

3. $(s_2, h_c \cdot h_q)[a](s_2', h_c' \cdot h_q)$

4. $\forall h.\ h\#h_q \Rightarrow h\#h_p$

By the general contents independence property of a, from 2, 3 and 4 and by the fact that client operations may mention only client variables, it follows that $(s_c, h_c \cdot h_p)[a](s_c', h_c' \cdot h_p)$, and hence $a, (s_1, h_1) \rightsquigarrow_{(p,\eta)} (s_1', h_1')$. □

Lemma 4 gives a basis for the simulation (or lifting) theorem. The following lemma ensures that the sequential composition of commands also respects forward simulation.

Lemma 5. *For all relations R_0, R_1 and R_2 and operations a, a', b and b', if $a'\langle\beta\rangle[\mathsf{fsim}(R_0, R_1)]a\langle\alpha\rangle$ and $b'\langle\beta\rangle[\mathsf{fsim}(R_1, R_2)]b\langle\alpha\rangle$, then*

$$(a'; b')\langle\beta\rangle[\mathsf{fsim}(R_1, R_2)](a; b)\langle\alpha\rangle.$$

Proof. Let a, a', b and b' be such that

$$a)\quad a'\langle\beta\rangle[\mathsf{fsim}(R_0, R_1)]a\langle\alpha\rangle \qquad \text{and} \qquad b)\quad b'\langle\beta\rangle[\mathsf{fsim}(R_1, R_2)]b\langle\alpha\rangle. \tag{4.1}$$

Let (s_1, h_1) and (s_2, h_2) be R_0 related states, such that

$$a'; b', (s_2, h_2) \rightsquigarrow_\beta \mathsf{av} \vee a'; b', (s_2, h_2) \rightsquigarrow_\beta \mathsf{wrong}. \tag{4.2}$$

We prove that then $a;b$ also produces a faulty state. From 4.2, by the definition of sequential composition we have that either $a', (s_2, h_2) \rightsquigarrow_\beta \mathsf{av} \vee a', (s_2, h_2) \rightsquigarrow_\beta \mathsf{wrong}$, or there exists a state (s_2', h_2') such that $a', (s_2, h_2) \rightsquigarrow_\beta (s_2', h_2') \wedge (b', (s_2', h_2') \rightsquigarrow_\beta \mathsf{av} \vee b', (s_2', h_2') \rightsquigarrow_\beta \mathsf{wrong})$. Since 4.1a) holds, we have that then either

$$a, (s_1, h_1) \rightsquigarrow_\alpha \mathsf{av} \ \vee a, (s_1, h_1) \rightsquigarrow_\alpha \mathsf{wrong} \tag{4.3}$$

in which case the first condition of the simulation is proved, or again by the definition of sequential composition, there exists a state (s_1', h_1') such that

$$a, (s_1, h_1) \rightsquigarrow_\alpha (s_1', h_1') \wedge (s_1', h_1')[R_1](s_2', h_2') \wedge (b', (s_2', h_2') \rightsquigarrow_\beta \mathsf{av} \vee b', (s_2', h_2') \rightsquigarrow_\beta \mathsf{wrong}).$$

Then, because 4.1b) holds, we have that

$$a, (s_1, h_1) \rightsquigarrow_\alpha (s_1', h_1') \wedge (b, (s_1', h_1') \rightsquigarrow_\alpha \mathsf{av} \vee b, (s_1', h_1') \rightsquigarrow_\alpha \mathsf{wrong}). \tag{4.4}$$

From 4.3 and 4.4, we conclude

$$a;b,(s_1,h_1) \rightsquigarrow_\alpha \mathsf{av} \ \vee \ a;b,(s_1,h_1) \rightsquigarrow_\alpha \mathsf{wrong}$$

which we wanted to prove.

Now suppose that $a;b,(s_1,h_1) \not\rightsquigarrow_\alpha \mathsf{av} \ \wedge \ a;b,(s_1,h_1) \not\rightsquigarrow_\alpha \mathsf{wrong}$. Then $a';b'$ does not produce a faulty state from (s_2,h_2) either by what we have just shown. Let (s_2'',h_2'') be a state such that $a';b',(s_2,h_2) \rightsquigarrow_\beta (s_2'',h_2'')$. By the definition of sequential composition, this means that there exists a state (s_2',h_2') such that $a',(s_2,h_2) \rightsquigarrow_\beta (s_2',h_2')$ and $b',(s_2',h_2') \rightsquigarrow_\beta (s_2'',h_2'')$. From 4.1a) it follows that there exists a state (s_1',h_1') such that

$$a,(s_1,h_1) \rightsquigarrow_\alpha (s_1',h_1') \ \wedge \ (s_1',h_1')[R_1](s_2',h_2') \ \wedge \ b',(s_2',h_2') \rightsquigarrow_\beta (s_2'',h_2'').$$

Using the assumption in 4.1b) this can be further transformed into

$$a,(s_1,h_1) \rightsquigarrow_\alpha (s_1',h_1') \ \wedge \ \wedge \, b,(s_1',h_1') \rightsquigarrow_\alpha (s_1'',h_1'') \ \wedge \ (s_1'',h_1'')[R_2](s_2'',h_2'')$$

i.e.

$$a;b,(s_1,h_1) \rightsquigarrow_\alpha (s_1'',h_1'') \ \wedge \ (s_1'',h_1'')[R_2](s_2'',h_2'').$$

which completes the proof of the second condition. □

Given abstract data type and its concrete representation, having a simulation between them when they are regarded as isolated objects is practically useless. Simulation theorem provides conditions under which this simulation can be extended to the whole language. This is important, as it gives a connection between the different representations of the same data type in broader contexts. When concrete representation simulates the abstract data type, the simulation theorem ensures that every computation which uses the concrete representation simulates the corresponding computation which instead uses the abstract data type, respecting the simulation relation $R * \mathsf{Id}$.

Theorem 5 (Simulation theorem)**.** *Let module (q,μ) simulate module (p,η) by a growing relation R. Then for all commands c and all predicates p_0, if c is a separation context for precondition p_0 and module (p,η), then we have*

$$c\langle(p,\eta)\rangle[\mathsf{fsim}(R * \Delta_{p_0}, R * \mathsf{Id})]c\langle(q,\mu)\rangle.$$

Proof. We prove the theorem by the induction on the structure of command c. When c is a module operation, the theorem follows from the assumption that (q,μ) simulates (p,η). When c is an atomic client operation, the theorem holds because of Lemma 4. When c is a sequential composition, the induction step goes through because of lemma 5.

Let $c \equiv \textbf{if } B \textbf{ then } c_1 \textbf{ else } c_2$. We only need to prove that the second condition of the forward simulation holds, because our program is a separation context for the abstract module and a precondition p_0. But we first need to prove that it is a separation context for the concrete module and the same precondition, too. Let states (s_1,h_1) and (s_2,h_2) be such that $(s_1,h_1)[R*\Delta_{p_0}](s_2,h_2)$ and suppose program c is not a separation context for the concrete module, i.e.

$c,(s_2,h_2) \leadsto_{(q,\mu)} \mathsf{wrong} \ \vee\ c,(s_2,h_2) \leadsto_{(q,\mu)} \mathsf{av}$

$\implies$ ($\because$ definition of the if-then-else statement)

$([\![B]\!]_{s_2} = \mathsf{true} \wedge (c_1,(s_2,h_2) \leadsto_{(q,\mu)} \mathsf{wrong} \vee c,(s_2,h_2) \leadsto_{(q,\mu)} \mathsf{av})) \vee$

$([\![B]\!]_{s_2} = \mathsf{false} \wedge (c_2,(s_2,h_2) \leadsto_{(q,\mu)} \mathsf{wrong} \vee c_2,(s_2,h_2) \leadsto_{(q,\mu)} \mathsf{av}))$

$\implies$ ($\because$ induction hypothesis and $(s_1,h_1)[R*\mathsf{Id}](s_2,h_2)$ and laws of classical logic)

$([\![B]\!]_{s_1} = \mathsf{true} \wedge (c_1,(s_1,h_1) \leadsto_{(p,\eta)} \mathsf{wrong} \vee c,(s_1,h_1) \leadsto_{(p,\eta)} \mathsf{av})) \vee$

$([\![B]\!]_{s_1} = \mathsf{false} \wedge (c_2,(s_1,h_1) \leadsto_{(p,\eta)} \mathsf{wrong} \vee c_2,(s_1,h_1) \leadsto_{(p,\eta)} \mathsf{av}))$

$\implies$ ($\because$ definition of the if-then-else statement)

$c,(s_1,h_1) \leadsto_{(p,\eta)} \mathsf{wrong} \vee c,(s_1,h_1) \leadsto_{(p,\eta)} \mathsf{av}.$

This contradicts the assumption that c is a separation context for p_0 and (p,η).

Now we prove the second condition of the forward simulation. Let states (s_1,h_1), (s_2,h_2) and (s_2',h_2') be such that $(s_1,h_1)[R*\Delta_{p_0}](s_2,h_2)$ and

$c,(s_1,h_1) \not\leadsto_{(p,\eta)} \mathsf{wrong} \wedge c,(s_1,h_1) \not\leadsto_{(p,\eta)} \mathsf{av} \wedge c,(s_2,h_2) \leadsto_{(q,\mu)} (s_2',h_2')$

$\implies$ ($\because$ definition of if-then-else statement)

$([\![B]\!]_{s_2} = \mathsf{true} \wedge c_1,(s_2,h_2) \leadsto_{(q,\mu)} (s_2',h_2')) \vee ([\![B]\!]_{s_2} = \mathsf{false} \wedge c_2,(s_2,h_2) \leadsto_{(q,\mu)} (s_2',h_2'))$

$\implies$ ($\because$ induction hypothesis and $(s_1,h_1)[R*\mathsf{Id}](s_2,h_2)$)

$(\exists(s_1',h_1').\ [\![B]\!]_{s_1} = \mathsf{true} \wedge c_1,(s_1,h_1) \leadsto_{(p,\eta)} (s_1',h_1') \wedge (s_1',h_1')[R*\mathsf{Id}](s_2',h_2')) \vee$

$(\exists(s_1',h_1').\ [\![B]\!]_{s_1} = \mathsf{false} \wedge c_2,(s_1,h_1) \leadsto_{(p,\eta)} (s_1',h_1') \wedge (s_1',h_1')[R*\mathsf{Id}](s_2',h_2'))$

$\iff$ ($\because$ rules of classical logic)

$\exists(s_1',h_1').\ (([\![B]\!]_{s_1} = \mathsf{true} \wedge c_1,(s_1,h_1) \rightsquigarrow_{(p,\eta)} (s_1',h_1') \wedge (s_1',h_1')[R*\mathsf{Id}](s_2',h_2')) \vee$

$([\![B]\!]_{s_1} = \mathsf{false} \wedge c_2,(s_1,h_1) \rightsquigarrow_{(p,\eta)} (s_1',h_1') \wedge (s_1',h_1')[R*\mathsf{Id}](s_2',h_2')))$

$\Longrightarrow$ ($\because$ definition of if-then-else statement)

$c,(s_1,h_1) \rightsquigarrow_{(p,\eta)} (s_1',h_1') \wedge (s_1',h_1')[R*\mathsf{Id}](s_2',h_2')$.

Let $c \equiv$ **while** B **do** c_1. In this case we do an inner induction on the length of the derivation for the **while**-loop. Suppose the theorem is true for all the derivations of length n.

For the derivations of length 0 we have that $[\![B]\!]_{s_2} = \mathsf{false}$, and since $(s_1,h_1)[R*\mathsf{Id}](s_2,h_2)$, then also $[\![B]\!]_{s_1} = \mathsf{false}$. In both semantics $\rightsquigarrow_{(p,\eta)}$ and $\rightsquigarrow_{(q,\mu)}$ the initial states are unchanged by the execution, and so the theorem holds in this case.

Let the length of the derivation be $n+1$. To obtain this derivation, we must use the rule

$$\frac{[\![B]\!]_s = \mathsf{true}\quad c_1;\textbf{while } B \textbf{ do } c_1,(s_2,h_2) \rightsquigarrow_{(q,\mu)} K}{\textbf{while } B \textbf{ do } c_1,(s_2,h_2) \rightsquigarrow_{(q,\mu)} K}$$

where K is wrong, av or a state (s_2',h_2'). We apply the similar reasoning as in case of sequential composition to c_1;**while** B **do** c_1. We apply the outer induction hypothesis to c_1 and the inner induction hypothesis to **while** B **do** c_1 since this is now a derivation of length n. $\square$

The reader will have noticed that our forward simulation method is tailored to distinguish between well-behaved programs and faulty ones. This is also reflected in the Simulation theorem. The theorem guarantees that having a concrete module simulate the abstract one, then the client program which is executed together with the concrete module will simulate the same client program when executed with abstract module, *only* when the mentioned client program is a separation context. In other words, the Simulation theorem does not say anything about the programs that are not separation contexts, and fails without this restriction.

One consequence of the Simulation theorem is that it is sufficient to prove that a client program is a separation context with respect to the abstract data type and a precondition once and for all; from that fact, it will follow that it is also a separation context for all legitimate implementations of the abstract data type.

Corollary 1. *Let module (q,μ) simulate module (p,η) by a growing relation R and let c be a client program. For all preconditions $p_0 \subseteq \Sigma$, if $c\langle(p,eta)\rangle$ is a separation context for (p,η) and p_0, then $c\langle(q,\mu)\rangle$ is a separation context for (q,μ) and p_0.*

This result, even though it comes out as a corollary is very useful for practical purposes. It relieves us of a burden of checking whether a client program is a separation context for each possible representation of the abstract data type. We get this for free by just proving once that it is a separation context for the most abstract representation.

4.5 Soundness of the forward simulation method

The traditional forward simulation method is unsound in presence of the low level pointer operations, as we have discussed and explained on several occasions. However, the modified forward simulation method, which considers only non-faulting executions and uses only growing relations is a sound method for proving data refinement. We prove the soundness of the forward simulation method with respect to the defined notion of data refinement.

We first prove a lemma which we use later in the proof of the soundness of the forward simulation method. The lemma gives an initial link between data refinement and forward simulation requiring that, in order to prove that a concrete module data refines the abstract one, it is necessary and sufficient to have a simulation between the two modules with respect to the identity relation.

Lemma 6. *A module (q,ε) data-refines another module (p,η) iff for all complete commands c, we have that $c\langle(q,\varepsilon)\rangle[\mathsf{fsim}(\mathsf{Id},\mathsf{Id})]c\langle(p,\eta)\rangle$.*

Proof. We prove this lemma by unrolling $\mathsf{fsim}(\mathsf{Id},\mathsf{Id})$. By the definition of fsim, we have that for all states (s,h)

1. if $(c,(s,h) \rightsquigarrow_{(q,\mu)} \mathsf{wrong} \vee c,(s,h) \rightsquigarrow_{(q,\mu)} \mathsf{av})$, then $c,(s,h) \rightsquigarrow_{(p,\eta)} \mathsf{wrong} \vee c,(s,h) \rightsquigarrow_{(p,\eta)} \mathsf{av}$, and

2. if $(c,(s,h) \not\rightsquigarrow_{(p,\eta)} \mathsf{wrong} \wedge c,(s,h) \not\rightsquigarrow_{(p,\eta)} \mathsf{av} \wedge c,(s,h) \rightsquigarrow_{(q,\mu)} (s'',h''))$ then there exists (s',h') such that

$$c,(s,h) \rightsquigarrow_{(p,\eta)} (s',h') \wedge (s',h')[\mathsf{Id}](s'',h''),$$

i.e., for all states (s,h), if $c,(s,h) \not\rightsquigarrow_{(p,\eta)} \mathsf{av} \wedge c,(s,h) \not\rightsquigarrow_{(p,\eta)} \mathsf{wrong}$, then

$$\big(c,(s,h) \not\rightsquigarrow_{(q,\mu)} \mathsf{av} \ \wedge\ c,(s,h) \not\rightsquigarrow_{(q,\varepsilon)} \mathsf{wrong}\big) \ \wedge$$
$$\forall (s',h').\ c,(s,h) \rightsquigarrow_{(q,\varepsilon)} (s',h') \ \Rightarrow\ c,(s,h) \rightsquigarrow_{(p,\eta)} (s',h')$$

and this is exactly the definition of data refinement.

□

Finally, we state and prove soundness of the forward simulation method with respect to data refinement.

Theorem 6 (Soundness)**.** *If a module (q,ε) forward-simulates another module (p,η) by a growing simulation relation $R \subseteq p \times q$, then (q,ε) data-refines (p,η).*

Proof. Suppose that a module (q,ε) simulates another module (p,η) by a growing relation $R \subseteq p \times q$. We will show that for all complete commands c, $c\langle(q,\varepsilon)\rangle$ $[\mathsf{fsim}(\mathsf{Id},\mathsf{Id})]$ $c\langle(p,\eta)\rangle$, and then by Lemma 22, module (q,ε) data-refines (p,η). Let c be an arbitrary (not complete) program. Then by the Simulation theorem,

$$c\langle(q,\varepsilon)\rangle[\mathsf{fsim}(R * \mathsf{Id})]c\langle(p,\eta)\rangle.$$

Also, since (q,ε) simulates (p,η) by R, we have

$$\mathsf{init}_{(q,\mu)}[\mathsf{fsim}(\mathsf{Id}, R * \mathsf{Id})]\mathsf{init}_{(p,\eta)}, \text{ and}$$
$$\mathsf{final}_{(q,\mu)}[\mathsf{fsim}(R * \mathsf{Id}, \mathsf{Id})]\mathsf{final}_{(p,\eta)}.$$

Then, by Lemma 5

$$\mathsf{init}_{(q,\mu)}; c\langle(q,\mu)\rangle; \mathsf{final}_{(q,\mu)}[\mathsf{fsim}(\mathsf{Id},\mathsf{Id})]\mathsf{init}_{(p,\eta)}; c\langle(p,\eta)\rangle; \mathsf{final}_{(p,\eta)}.$$

The definition of a complete program then yields

$$c_c\langle(q,\varepsilon)\rangle[\mathsf{fsim}(\mathsf{Id},\mathsf{Id})]c_c\langle(p,\eta)\rangle.$$

□

4.6 Examples

To illustrate the forward simulation method, we thoroughly examine the relationship between different implementations of a memory manager data type, which has operations for allocating fresh memory and disposing used memory.

In this example, we assume that the underlying storage model is similar to the RAM model, and it differs from it in that variables can hold both values and sets of values, i.e. $s : \mathsf{Var} \longrightarrow \mathsf{Val} \cup \mathcal{P}(\mathsf{Val})$.

For the *abstract* version of the memory manager we will consider the"magical malloc module". It is magical in that the module does not own any locations at all, producing them as if out of thin air. (In implementation terms, the thin air is like a call to a system routine such as sbrk.) Therefore, the resource invariant of the module, p in our formal setup, is the predicate emp, and so we denote the abstract module by (emp, η), where η provides the meaning of the operations. Module environment η maps the operations $\mathsf{new}_A(x)$ and $\mathsf{dispose}_A(x)$ to the greatest relations satisfying the following specifications.

$$\mathsf{new}_A(x) : \{\mathsf{emp}\} - \{x \mapsto -\}[x], \quad \mathsf{dispose}_A(x) : \{x \mapsto -\} - \{\mathsf{emp}\}[\,].$$

This is the meaning of allocation and disposal that is usually presumed in separation logic.

On the intermediate level, the intention is to keep locations owned by the module in a set. The resource invariant of the module is then the value of the set variable, which we will denote by f. The operations of the intermediate interpretation are mapped by the environment μ to the greatest relations satisfying the following specifications.

$$\begin{aligned}\mathsf{new}_I(x) : \quad & \{f = Y \neq \emptyset\} - \{(f = Y \setminus \{x\} * x \mapsto -)\}[x, f]\\ & \{f = \emptyset\} - \{f = \emptyset * x \mapsto -\}[x]\\ \mathsf{dispose}_I(x) : & \{f = Y * x \mapsto -\} - \{f = Y \cup \{x\}\}[f]\end{aligned}$$

Set Y is used to keep track of the initial contents of f. Note that it is not altered because it is not in the modifies set, a set of actual locations owned by the module. We intend that $\mathsf{new}_I(x)$ is the greatest relation satisfying both stated specifications. We assume that if the set of owned locations becomes empty, we call a "system routine" (like sbrk) to get a new location.

Now we prove the simulation between these two implementations. First, we define the simulation relation

$$R_{AI} = \{((s_1,h_1),(s_2,h_2)) \mid (s_1,h_1) \in \mathsf{emp} \wedge s_2(f) = \mathsf{dom}(h_2))\}.$$

Finally, we can state a lemma, which gives a relationship between the abstract and the concrete implementations.

Lemma 7. *The module (f,μ) simulates module (emp,η) with respect to the relation R_{AI}.*

Proof. To prove that one module simulates another, we need to prove for each pair of operations that the concrete operation simulates the corresponding abstract one. Consider states (s_1,h_1) and (s_2,h_2) such that $(s_1,h_1)[R_{AI} * \mathsf{Id}](s_2,h_2)$.

We first argue that the intermediate alloc simulates the abstract one. Note that $\mathsf{alloc}(x)$ does not produce a faulty state either in the $\rightsquigarrow_{\mathsf{emp},\eta}$ or in $\rightsquigarrow_{f,\mu}$ semantics (the module invariants hold in both states), and so we only need to prove that the second condition of the forward simulation is satisfied. To do that, suppose that (s_2',h_2') is such that $\mathsf{alloc}(x),(s_2,h_2) \rightsquigarrow_{(f,\mu)} (s_2',h_2')$. Then $s_2' = s_2[x \mapsto l]$, where l is the newly allocated location. If the value of f in state (s_2,h_2) is the empty set, then location l in state (s_2',h_2') is disjoint from heap h_2, i.e. $h_2' = h_2 \cdot [l \mapsto -]$. This location is not in heap h_1 either (since $\mathsf{dom}(h_1) \in \mathsf{emp}$), and so it can be returned by $\mathsf{alloc}(x)$ in $\rightsquigarrow_{(\mathsf{emp},\eta)}$ semantics. The output state on the abstract level then is $(s_1[x \mapsto l], h_1 \cdot [l \mapsto -])$.The output states are related by $R * \mathsf{Id}$: the module parts are unchanged and the client parts are changed in the same way, so they remain identical. Similarly, if the value of f is a non-empty set in (s_2,h_2), then an element from that set is returned by alloc on the concrete level. Since the abstract module state is empty and the abstract and concrete client states are the same, it must be that the location returned by the concrete alloc is available in the abstract state. Then, the abstract alloc can return the same location as the concrete one in a way that the output states remain related by $R * \mathsf{Id}$.

Now, we prove that the intermediate $\mathsf{dispose}$ simulates the abstract one. Let $\mathsf{dispose}(x)$, $(s_2,h_2) \rightsquigarrow_{(f,\mu)}$ wrong. This can happen only when x holds a dangling pointer, and as x is a client variable and the concrete and abstract client states are identical, it must be that $\mathsf{dispose}(x),(s_1,h_1) \rightsquigarrow_{(\mathsf{emp},\eta)}$ wrong. Now, suppose that $\mathsf{dispose}$ does not output any faulty states and let (s_2',h_2') be such that $\mathsf{dispose}(x),(s_2,h_2) \rightsquigarrow_{(f,\mu)} (s_2',h_2')$. Then, (s_2',h_2') is such that x has value l both in s_2 and s_2' and $h_2 = h_2' \cdot [l \mapsto -]$. By the specification of

$\eta(\mathsf{dispose}(x))$, state (s'_1, h'_2) such that $h_1 = h'_1 \cdot [l \mapsto -]$ is the output state of $\mathsf{dispose}(x)$ in $\rightsquigarrow_{(p,\eta)}$ semantics, and it is easy to see that $(s'_1, h'_1)[R * \mathsf{Id}](s'_2, h'_2)$. □

Now, we consider even more concrete implementation of the memory manager module. It is the concrete implementation defined in the Section 3.4.1 in Table 3.5, which we will denote by (p, ν), where $p = \exists\alpha.\ \mathsf{list}(\alpha, \mathsf{ls})$. Function ν maps the operations of the module to the greatest relations satisfying the specifications given in Table 3.5. We examine the relationship between the intermediate and the concrete implementations because it involves a more subtle simulation relation. The simulation relation is given by

$$R_{IC} = \{((s_1,h_1),(s_2,h_2)) \mid s_1(f) = \mathsf{dom}(h_1)\ \wedge\ (s_2,h_2) \in \mathsf{list}(\alpha,\mathsf{ls})\ \wedge\ s_1(f) = set(\alpha)\}.$$

Here, $set(\alpha)$ denotes a set which consists of the individual elements of the sequence α.

Lemma 8. *The module* $(\mathsf{list}(\alpha,\mathsf{ls}), \nu)$ *simulates module* (f, μ) *with respect to the relation* R_{IC}.

Proof. Again, we prove for each pair of operations, that the concrete operation simulates the corresponding abstract one. Let states (s_1, h_1) and (s_2, h_2) be such that $(s_1, h_1)[R_{IC} * \mathsf{Id}](s_2, h_2)$.

We first consider the operation alloc. Neither concrete nor abstract interpretation of $\mathsf{alloc}(x)$ output a faulty state when executed in (s_2, h_2) and (s_1, h_1), respectively, because in each of the states the corresponding module invariant holds and the operations do not require any additional resources to run safely. Let (s'_2, h'_2) be such that $\mathsf{alloc}(x), (s_2, h_2) \rightsquigarrow_{(p,\nu)} (s'_2, h'_2)$ and let $x \mapsto l$ in state (s'_2, h'_2), where l is either in the free list (i.e. in the sequence α) in state (s_2, h_2) or it is not in the current state at all. If l is in the free list, then $s'_2 = s_2[x \mapsto l]$ and $h'_2 = h_2[\mathsf{ls} \mapsto list\ \alpha']$, where $\alpha = l \cdot \alpha'$ ($\mathsf{ls} \mapsto list\ \alpha$ denotes the fact that the free list contains locations given by sequence α). Since $(s_1, h_1)[R_{IC} * \mathsf{Id}](s_2, h_2)$, location l is also in the free set in state (s_1, h_1), and is available for allocation. Let (s'_1, h'_1) be a state such that $s'_1 = s_1[x \mapsto l]$ and $(h'_1 = h_1[Y \mapsto Y \setminus \{l\}]$. Then, it can be easily seen that $\mathsf{alloc}(x), (s_1, h_1) \rightsquigarrow_{(f,\mu)} (s'_1, h'_1)$ and $(s'_1, h'_1)[R_{IC} * \mathsf{Id}](s'_2, h'_2)$. If, on the other hand, l is not in state (s_2, h_2), then $h'_2 = h_2 \cdot l \mapsto -$. Because (s_1, h_1) is related to (s_2, h_2) by $R * \mathsf{Id}$, location l is not in the state (s_1, h_1) either, and so $h'_1 = h_1 \cdot l \mapsto -$. Then again (s'_1, h'_1) is the output state of $\mathsf{alloc}(x)$, according to μ, and it is also related to (s'_2, h'_2) by $R_{IC} * \mathsf{Id}$.

Finally, we consider the operation $\mathsf{dispose}()$. If variable x points to a location which is not in the client part of the state (s_2, h_2) then it is not in the client part of the state (s_1, h_1) either, and so the execution of $\mathsf{dispose}(x)$ in both states (s_1, h_1) and (s_2, h_2) (in corresponding semantics) leads to a faulty state. Suppose now that $\mathsf{dispose}(x)$ does not produce a faulty state and state (s'_2, h'_2) is such that $\mathsf{dispose}(x), (s_2, h_2) \rightsquigarrow_{(p,v)} (s'_2, h'_2)$. Let client variable x point to a location l (i.e. $x \mapsto l$) and let free list hold sequence α in state (s_2, h_2). Then $s'_2 = s_2$ and $h'_2 = h_2[\mathsf{ls} \mapsto \mathit{list}\ \alpha']$, where $\alpha' = l \cdot \alpha$. Then state $(s_1, h_1[Y \mapsto Y \cup \{l\}]$ is the output state of $\mathsf{dispose}(x)$ according to μ, and it is such that $(s'_1, h'_1)[R_{IC} * \mathsf{Id}](s'_2, h'_2)$. $\square$

We have examined the relationship between three different implementations of the memory manager module. We have chosen the memory manager because, apart from being a canonical example, it is often a stumbling point for the theories that deal with abstraction. Firstly, it deals with pointers and secondly, it illustrates the concept of ownership transfer between the module and the client program. The latter usually presents problems which cannot be easily solved by imposing certain language mechanisms, such as types, and if they can be solved, the solutions tend to be complex and unnatural.

The relationship between the three implementations of the memory manager module shows a hierarchy and so we have named them accordingly. Our results, the soundness and simulation theorems, ensure that this hierarchy is preserved in terms of data refinement and that each more abstract implementation of the memory manager can be replaced by any more concrete implementation in all well-behaving computations.

4.7 Discussion

Some of the work in chapters 3 and 4 is joint work with Noah Torp-Smith and Peter O'Hearn and was presented at the workshop SPACE [51] and at the conference FSTTCS [52]. However, material presented in the thesis somewhat improves on that work.

In the paper [52], client primitive operations are required to be deterministic, which is forced by setting them to be functions. One effect of this is that, when frame conditions are imposed, the client operations are unable to do any allocation. Allocation could be viewed only as a module operation. Technically, the determinism restriction was necessary for simple simulation theorem to be true. Our operations were assumed to be local actions, that is, they had to satisfy well known locality properties: safety monotonicity and frame property. It turned out that, even though these two properties are fundamental in work with separation logic, they are not sufficient, at least not in the area of application of the ideas from separation logic to data refinement.

In the thesis, we introduce a new property, called content independence, which says that if a command has a safe, non-faulting execution starting from some state, then it does not care about the contents of any extra memory. The only thing that matters is that the command "knows" of this extra part, but not its contents. All the standard operations including allocation and deallocation satisfy this property. Now, allocation and deallocation are the part of the language and thanks to content independence the Simulation theorem holds for a language that contains these two client operations. Of course, one could implement his own memory manager as a module.

Another interesting point is that while in work presented in [52] the client program is limited to the usage of only one module, content independence also opens up a way to handling multiple modules. We do not give a solution to that in the thesis, but leave it for future work, as there are other subtleties.

While in our previous work we only considered lifting of simulation, i.e. Simulation theorem, in the thesis we also prove soundness results for the forward simulation method with respect to data refinement.

5

Proving Separation Contexts

So far, we have introduced separation contexts, a class of client programs which, interacting with a certain module, access the internals of the module only through the provided module operations and do not interfere with the module otherwise. Separation contexts are an important concept when considering data abstraction in presence of pointers, in a modular way. The Simulation theorem ensures that a relationship between different implementations of an abstract data type can be lifted to separation contexts. If we do not put any restrictions on the client programs, that is if we allow non-separation contexts, the Simulation theorem fails. This fact stresses the importance of separation contexts.

But, separation contexts were introduced semantically, and no method was given to check or prove them. In this chapter we show that it is enough to prove that a program satisfies a certain specification in separation logic to make sure that we are dealing with a separation context. Moreover, now that there exist tools based on separation logic, such as Smallfoot [13, 14, 15], this implies that some amount of automatic checking of separation contexts can be done. We illustrate this latter point by example.

5.1 More about Separation Logic

In Chapter 3 we have introduced the assertion language of separation logic. This chapter studies the relationship between separation contexts and logic, that is, how can one prove that a certain program is a separation context in logic. We will illustrate this point with several examples, which require acquaintance with some axioms and rules of separation logic [72].

We first introduce the *tight interpretation* [60] of triples.

Definition 15 (Tight interpretation)**.** *For predicates p and q and a command c, we say that a specification $\{p\}c\{q\}$ holds if and only if for all states (s,h) which satisfy p,*

1. $\neg(s,h)[c]\mathsf{wrong}$*, and*
2. *if $(s,h)[c](s',h')$, then state (s',h') satisfies q.*

The tight interpretation of triples ensures that the command does not touch any storage that is not described by the precondition. The command faults as soon as it dereferences memory beyond what is guaranteed to exist in the current state by the precondition. However, if the command touches only what is provided by the precondition and it produces an output state, then the output state needs to satisfy the postcondition. Tight interpretation of triples is assumed in all work on Separation logic and plays an important role in the proof of the soundness of the frame rule, to be introduced bellow.

We now introduce the so called "small axioms" [60].

$$\{E \mapsto -\}[E] := F\{E \mapsto F\}$$
$$\{x \mapsto -\}\mathsf{dispose}(x)\{\mathsf{emp}\}$$
$$\{\mathsf{emp}\}x := \mathsf{alloc}()\{x \mapsto -\}$$
$$\{x = n\}x := E\{x = E[n/x]\}$$
$$\{E \mapsto n \ \wedge \ x = m\}x := [E]\{x = n \ \wedge \ E[m/x] \mapsto n\}$$

Small axioms are used to reason locally. Namely, each of these axioms specifies the corresponding command in terms of storage necessary for it to run faultlessly. For instance, the first axiom says that in order to assign to a memory location, the location has to be allocated, i.e., it exists in the current heap. The second axiom says that after disposing the only location in the heap there is no starage left. After allocation of one location, the location becomes available in the current heap. In order to read the value saved on a certain memory location, the location has to exist in the current heap.

Small axioms give us the basis for reasoning about programs. *Frame rule* [60] then gives means to moving from local to global reasoning.

Frame rule

$$\frac{\{P\}C\{Q\}}{\{P*R\}C\{Q*R\}}\ \mathrm{Modifies}(C)\cap\mathrm{Free}(R)=\{\}$$

Here, the idea is that a precondition P also specifies the storage sufficient for the safe execution of a program C, which if terminates ends up in a state satisfying Q. Then if we run program C in a state that has additional memory, beyond that described by P, then that extra memory will remain unchanged, under the condition that the variables changed by the program C are not free in predicate describing that extra memory. Detailed proof of the soundness of the frame rule is given in Yang's thesis [80]. The proof is based on the locality properties: safety monotonicity and frame property introduced in Section 3.3.4 and the tight interpretation of triples.

5.2 Connection Between Separation Logic and Separation Contexts

The main result of this chapter is a theorem which gives conditions under which separation contexts can be proved in logic. We first need to prove few smaller results, which are then used in the proof of the main theorem.

Let p be a precise predicate and let r be a general local action such that

1. r never generates av with respect to p, i.e. for all states (s,h), such that $h=h_0\cdot h_p$, where $(s,h_p)\in p$
$$(s,h_0)[r]\mathsf{wrong}\Longrightarrow(s,h)[r]\mathsf{wrong}$$
2. r preserves p: for all (s,h) and (s',h')
$$(s,h)\in p*\mathsf{true}\ \wedge\ \neg(s,h)[r]\mathsf{wrong}\ \wedge\ (s,h)[r](s',h')\Longrightarrow(s',h')\in p*\mathsf{true}.$$

Here, we denote by (p,η) a module with a resource invariant p and operations defined by η. Operation r is any action satisfying the above conditions. It may be defined by η, as the above conditions coincide with the definition of a module. In that case, there is some r such that $\eta(\mathsf{r})=r$.

Definition 16 (Hiding)**.** *The hiding of r by p, denoted* $\mathsf{hide}(r,p)$*, is an action defined as follows:*

$$\begin{aligned}
(s,h)[\mathsf{hide}(r,p)]\mathsf{wrong} &\iff \exists h_p.\ (s,h_p)\in p \wedge h_p\#h \wedge (s,h\cdot h_p)[r]\mathsf{wrong}\\
(s,h)[\mathsf{hide}(r,p)](s',h') &\iff \exists h_p,h'_p.\ (s,h_p),(s',h'_p)\in p \wedge h_p\#h \wedge h'_p\#h' \wedge\\
&\qquad (s,h_p\cdot h)[r](s',h'_p\cdot h').
\end{aligned}$$

The definition specifies an action $\mathsf{hide}(r,p)$ which "hides" the p-satisfying piece of the input and output states, where p is a precise predicate. Action $\mathsf{hide}(r,p)$ is defined just when original action r requires that parts of its input and output states satisfy predicate p. Also, $\mathsf{hide}(r,p)$ goes wrong on a state, whenever original action r generates wrong starting from the same state extended by some disjoint state that satisfies p.

In the proofs of the lemmas that follow and the main theorem, we use the results from the previous chapter – the simulation method and the lifting theorem. These require that all actions that are involved need to be local actions, i.e. actions which satisfy safety monotonicity, frame property and general content independence. The following lemma makes sure that this is the case for action $\mathsf{hide}(r,p)$.

Lemma 9. *Action* $\mathsf{hide}(r,p)$ *is a general local action.*

Proof. We need to prove that $\mathsf{hide}(r,p)$ satisfies all three properties of the general local actions: safety monotonicity, frame property and general content independence.

For safety monotonicity, consider s and heaps h_0 and h_1 such that $h_0\#h_1$ and $(s,h_0\cdot h_1)[\mathsf{hide}(r,p)]\mathsf{wrong}$. Then, by the definition of $\mathsf{hide}(r,p)$ there is a heap h_p satisfying the following conditions:

$$(h_0\cdot h_1)\#h_p \wedge (s,h_0\cdot h_1\cdot h_p)[r]\mathsf{wrong}.$$

Since r satisfies safety monotonicity, the second conjunct implies that $(s,h_0\cdot h_p)[r]\mathsf{wrong}$. This faulting computation of r from $(s,h_0\cdot h_p)$ proves the required $(s,h_0)[\mathsf{hide}(r,p)]\mathsf{wrong}$.

For the frame property, suppose that s,s' and h_0,h_1,h' are such that

$$(s,h_0\cdot h_1)[\mathsf{hide}(r,p)](s',h') \wedge \neg(s,h_0)[\mathsf{hide}(r,p)]\mathsf{wrong}.$$

If we unroll the definition of $\mathsf{hide}(r,p)$, from the first condition we get heaps h_p,h'_p that satisfy predicate p such that

$$h_p\#(h_0\cdot h_1) \wedge h'_p\#h' \wedge (s,h_p\cdot h_0\cdot h_1)[r](s',h'_p\cdot h'),$$

and from the second condition we have that

$$\neg(s, h_0 \cdot h_p)[r]\mathsf{wrong}.$$

We now apply the frame property for r, and obtain a heap m' such that

$$m' \# h_1 \ \wedge\ h' \cdot h'_p = m' \cdot h_1 \ \wedge\ (s, h_0 \cdot h_p)[r](s', m').$$

We now show that the heap m' can be split into h'_p and some heap h'_0. Note that since h'_p satisfies predicate p, this splitting strategy of m' gives $(s, h_0)[\mathsf{hide}(r, p)](s', h'_0)$ and $h'_0 \cdot h_1 = h'$, as required. The main strategy for obtaining the desired splitting of m' is to use the assumptions about r and p: r preserves p, r never generates av, and p is precise. Since state $(s, h_0 \cdot h_p)$ satisfies $p * \mathsf{true}$ and from this state r generates neither av nor wrong, the final state (s', m') of execution $(s, h_0 \cdot h_p)[r](s', m')$ also satisfies $p * \mathsf{true}$. Thus, there is a splitting $m'_p \cdot m'_0$ of m' such that $(s', m'_p) \in p$. Since $m' \cdot h_1 = h' \cdot h'_p$, we have

$$m'_p \cdot m'_0 \cdot h_1 = h' \cdot h'_p.$$

Predicate p is precise, so p-satisfying heaps m'_p and h'_p must be the same. This shows that $m'_p \cdot m'_0$ is the required splitting of m'.

To prove that general content independence holds for $\mathsf{hide}(r, p)$, assume that s, s' and h, h', h_0 are such that

$$h \# h_0 \ \wedge\ h' \# h_0 \ \wedge\ \neg(s, h)[\mathsf{hide}(r, p)]\mathsf{wrong} \ \wedge\ (s, h \cdot h_0)[\mathsf{hide}(r, p)](s', h' \cdot h_0).$$

Then, by the definition of $\mathsf{hide}(r, p)$, there exist heaps h_p, h'_p that satisfy p such that

$$(h \cdot h_0) \# h_p \ \wedge\ (h' \cdot h_0) \# h'_p \ \wedge\ (s, h \cdot h_0 \cdot h_p)[r](s', h' \cdot h_0 \cdot h'_p) \ \wedge\ \neg(s, h \cdot h_p)[r]\mathsf{wrong}.$$

For all h_1 such that for all states m, if $m \# h_0$ then $m \# h_1$, by the general content independence for r, we have that $(s, h \cdot h_1 \cdot h_p)[r](s', h' \cdot h_1 \cdot h'_p)$, which is equivalent to $(s, h \cdot h_1)[\mathsf{hide}(r, p)](s', h' \cdot h_1)$ by the definition of $\mathsf{hide}(r, p)$. Thus, the general content independence property also holds for $\mathsf{hide}(r, p)$. $\square$

Action $\mathsf{hide}(r, p)$ is defined in terms of action r. The connection between the two actions is also reflected in the relationship between the specifications of the two actions, i.e. the Hoare-triples which they satisfy. In accordance with its definition, the specification of $\mathsf{hide}(r, p)$ hides the p-part in the pre and post conditions which appear in the specification of r.

Lemma 10. *For all predicates q, q' if $\{q * p\} r \{q' * p\}$, then $\{q\}\mathsf{hide}(r, p)\{q'\}$.*

Proof. Let (s, h) be a state satisfying the precondition q. First, we show that $\mathsf{hide}(r, p)$ does not fault from (s, h). To obtain a contradiction, suppose $(s, h)[\mathsf{hide}(r, p)]\mathsf{wrong}$. Then, there exists a heap h_p satisfying p such that $h_p \# h$ and $(s, h \cdot h_p)[r]\mathsf{wrong}$. But $h \cdot h_p$ satisfies $q * p$, so $(s, h \cdot h_p)[r]\mathsf{wrong}$ contradicts the assumption that $\{q * p\} r \{q' * p\}$ holds. Next, we prove that all the outputs of $\mathsf{hide}(r, p)$ from (s, h) satisfy q'. Let (s', h') be an arbitrary state such that $(s, h)[\mathsf{hide}(r, p)](s', h')$. Then, by the definition of $\mathsf{hide}(r, p)$, there exist h_p, h'_p satisfying p such that

$$h \# h_p \;\wedge\; h' \# h'_p \;\wedge\; (s, h \cdot h_p)[r](s', h' \cdot h'_p).$$

Since $(s, h \cdot h_p) \in p * q$ and the triple $\{q * p\} r \{q' * p\}$ holds, state $(s', h' \cdot h'_p)$ has to be in $q' * p$. This implies that (s', h') must satisfy q', because h'_p is the unique subheap of $h' \cdot h'_p$ that satisfies p. □

In Chapter 3 we have introduced the semantics $\rightsquigarrow_{(p,\eta)}$ which is parameterized by a module (p, η). It differs from the standard semantics $\rightsquigarrow_\eta$ only in that the parameterized one can detect an access violation – an attempt of the client program to dereference storage owned by the module. The resources of the module are described by a precise predicate and that makes it possible to determine which part of the state belongs to the client and which to the module. The standard semantics has no knowledge of this partitioning of the state and module's resources, and hence does not produce an error when something that belongs to the module is read from or written to. However, if we consider a module with no resources, i.e. a module with resources which can be described by a precise predicate emp, then a semantics parameterized by such module is identical to the standard semantics. The following lemma states that more formally.

Lemma 11. *For all η, $\rightsquigarrow_\eta$ is the same as $\rightsquigarrow_{(\mathsf{emp},\eta)}$.*

Proof. We prove the lemma by induction on the structure of c. When c is a module operation, since in both semantics the execution of c is determined by its definition given by η, it is easy to see that in this case $\rightsquigarrow_\eta$ and $\rightsquigarrow_{(\mathsf{emp},\eta)}$ are the same. Similarly, when c is a basic command a, the cases

1. $a, (s, h) \rightsquigarrow_\eta (s', h') \iff a, (s, h) \rightsquigarrow_{(\mathsf{emp},\eta)} (s', h')$, and

2. $a,(s,h) \rightsquigarrow_\eta \mathsf{wrong} \iff a,(s,h) \rightsquigarrow_{(\mathsf{emp},\eta)} \mathsf{wrong}$

are straightforward. The only case that differs these two semantics is when a command produces an av. Suppose now that (s,h) is such state that $a,(s,h) \rightsquigarrow_{(\mathsf{emp},\eta)} \mathsf{av}$ (note that there is no rule in $\rightsquigarrow_\eta$ for this). Then, by the corresponding rule, we have that

$$h = h_{\mathsf{emp}} \cdot h_U \ \wedge \ \neg(s,h)[a]\mathsf{wrong} \ \wedge \ (s,h_U)[a]\mathsf{wrong}$$

which is equivalent to

$$h = h_U \ \wedge \ \neg(s,h)[a]\mathsf{wrong} \ \wedge \ (s,h)[a]\mathsf{wrong}$$

and this is equivalent to contradiction. Thus, this is an impossible case, and hence, the semantics $\rightsquigarrow_\eta$ for the basic client operations is equivalent to the semantics $\rightsquigarrow_{(\mathsf{emp},\eta)}$.

When c is a composition of c_1 and c_2, by the induction hypothesis, the semantics $\rightsquigarrow_\eta$ and $\rightsquigarrow_{(\mathsf{emp},\eta)}$ will yield the same executions for c_1 and c_2, and for example, if we apply the rule

$$\frac{c_1,(s,h) \rightsquigarrow (s'',h'') \quad c_2,(s'',h'') \rightsquigarrow (s',h')}{c_1;c_2,(s,h) \rightsquigarrow (s',h')}$$

in both semantics, we will get the same results. The reasoning is similar for the remaining two rules, bearing in mind that the rule for av in $\rightsquigarrow_\eta$ is impossible.

The remaining cases are proved similarly, using the induction hypothesis. $\square$

Before we go on and prove the main result of this chapter we need to introduce some additional notation. To specify that a certain module is defined and assumed to exist in the environment, we use *environment specification*. An environment specification $\Gamma : \mathsf{MOp} \rightharpoonup_{\mathsf{fin}} \mathsf{Pred} \times \mathsf{Pred}$ assigns a specification to each module operation. We say that a module environment η satisfies an environment specification Γ, denoted $\eta \models \Gamma$, just when meaning of each of the module operations in the environment η satisfies the intended specification given by Γ. More formally,

Definition 17. *A module environment η satisfies an environment specification Γ, denoted by $\eta \models \Gamma$, if and only if*

$$\forall f \in \mathsf{MOp}.\ \forall (p,q) = \Gamma(f).\ \{p\}\eta(f)\{q\}.$$

We express the fact that a client program is executed in an environment in which a certain module is defined, by using sequents. A sequent $\Gamma \vdash \{p\}c\{q\}$ says that if every specification in Γ is true of some module environment, then so is a specification of a client program c, $\{p\}c\{q\}$. More formally,

Definition 18. *A program c satisfies a specification $\{p\} - \{q\}$ in an environment Γ, denoted by $\Gamma \vdash \{p\}c\{q\}$, if and only if*

$$\forall \eta.\ (\eta \models \Gamma \Longrightarrow \{p\}c\{q\}).$$

We define an *expansion* of an environment specification Γ by a predicate p_0.

Definition 19. *An expansion of an environment specification Γ by a predicate p_0 is the environment specification $\Gamma' = \Gamma * p_0$ such that the following holds:*

$$\mathsf{dom}(\Gamma) = \mathsf{dom}(\Gamma')\ \wedge\ \forall f \in \mathsf{MOp}.\ \forall (p,q) = \Gamma(f).\ \Gamma'(f) = (p * p_0, q * p_0).$$

The first conjunct ensures that environments Γ and Γ' talk about the modules with the same set of operations. The second conjunct defines Γ' to be an environment in which each specification given by Γ is "expanded" by tacking on a predicate p_0 to the pre and post conditions. One can think of Γ as of an environment which hides the resources described by a predicate p_0, which are explicit in the environment Γ'.

Let η_0 be a module environment defined by:

$$\forall f \in \mathsf{MOp}. \eta_0(f) = \mathsf{hide}(\eta(f), p).$$

Actions r and $\mathsf{hide}(r,p)$ have a special relationship. This is also reflected in the fact that next lemma proves. Namely, there can always be found a growing relation R, by which action r simulates action $\mathsf{hide}(r,p)$. In fact, module (p,η) simulates module (emp,η_0) that hides its resources.

Lemma 12. *Module (p,η) simulates module (emp,η_0), i.e. for all actions r defined by η, r simulates $\mathsf{hide}(r,p)$ by some growing relation R^1, i.e.*

$$r\langle(p,\eta)\rangle[\mathsf{fsim}(R * \Delta)]\mathsf{hide}(r,p)\langle(\mathsf{emp},\eta_0)\rangle$$

Proof. Let R be a relation between states defined by

$$(s,h)[R](s,h') \iff \exists h_p \in p.\ (h\# h_p\ \wedge\ h \cdot h_p = h')$$

[1]Recall that $r\langle\alpha\rangle$ denotes the fact that action r is evaluated in semantics α.

It is easy to see that the relation R is growing. Let (s,h) and (s,h') be states related by $R*\Delta$. Then, by the definition of $*$, Δ and R, there exist heaps h_0, h_p, h_1 such that

$$h_0 \# h_1 \ \wedge\ (h_0 \cdot h_1) \# h_p \ \wedge\ (h_0 \cdot h_1) = h \ \wedge\ (h_0 \cdot h_1 \cdot h_p) = h'.$$

Suppose that

$$r, (s,h') \leadsto_{(p,\eta)} \mathsf{wrong} \ \vee\ r, (s,h') \leadsto_{(p,\eta)} \mathsf{av}.$$

Since r does not generate av, the first disjunct, $r, (s,h') \leadsto_{(p,\eta)} \mathsf{wrong}$, must hold. This is possible only when $(s,h')[r]\mathsf{wrong}$. Then, by the definition of $\mathsf{hide}(r,p)$, we have that $(s,h_0 \cdot h_1)[\mathsf{hide}(r,p)]\mathsf{wrong}$, and hence $\mathsf{hide}(r,p), (s,h) \leadsto_{(\mathsf{emp},\eta_0)} \mathsf{wrong}$. Thus, the first condition of the simulation holds. For the second condition, suppose that

$$\mathsf{hide}(r,p), (s,h) \not\leadsto_{(\mathsf{emp},\eta_0)} \mathsf{wrong} \ \wedge\ \mathsf{hide}(r,p), (s,h) \not\leadsto_{(\mathsf{emp},\eta_0)} \mathsf{av}.$$

Consider a final state (s_1, m_1) of r from the initial state (s,h'), i.e. $r, (s,h') \leadsto_{(p,\eta)} (s_1, m_1)$. By what we have just shown, neither $r, (s,h') \leadsto_{(p,\eta)} \mathsf{wrong}$ nor $r, (s,h') \leadsto_{(p,\eta)} \mathsf{av}$ is a possible computation. Since r preserves p and (s,h') is in $p * \mathsf{true}$, this safety of r at (s,h') implies that the final state (s_1, m_1) is also in $p * \mathsf{true}$. That is, there is a splitting $m_p \cdot m_0$ of m_1 such that $m_p \in p$. We will show that the state m_0 is required state in the second condition. Since $(s, h_0 \cdot h_1 \cdot h_p)[r](s_1, m_0 \cdot m_p)$ and h_p, m_p satisfy p, we have that

$$(s, h_0 \cdot h_1)[\mathsf{hide}(r,p)](s_1, m_0),$$

and by the definitions of $*$, Δ and R,

$$(s_1, m_0)[R * \Delta](s_1, m_0 \cdot m_p).$$

These two properties of m_0 show that m_0 is the required state. □

Finally, we state the main result, the theorem which gives a connection between the separation contexts and the logic.

Theorem 7. *Suppose $\Gamma \vdash \{p\}c\{q\}$. For all modules (p_0, η), if $\eta \models \Gamma * p_0$, then we have that*

$$\forall (s,h) \in p * p_0.\ \big(c, (s,h) \not\leadsto_{(p_0,\eta)} \mathsf{av} \ \wedge\ c, (s,h) \not\leadsto_{(p_0,\eta)} \mathsf{wrong}\big).$$

The theorem says that if a program c satisfies a triple $\{p\} - \{q\}$ whenever the specifications given by the environment Γ are true, and given a module which satisfies the environment Γ expanded by the predicate which describes its resource invariant, then program c is a separation context for that module and a precondition p. In other words, to prove that a client program is a separation context for a module and a precondition, it is enough to prove that it meets a corresponding specification assuming an environment in which the module is defined.

Proof. Let η_0 be a module environment defined by:

$$\forall f \in \mathsf{MOp}.\eta_0(f) = \mathsf{hide}(\eta(f), p_0).$$

Let $R = \{((s,[]),(s,h_{p_0})) \mid (s,h_{p_0}) \in p_0\}$ be a growing simulation relation. By lemma 12, module (p_0,η) simulates module (emp,η_0) by R. By the corollary 1 of the simulation theorem, we have that if the (abstract) execution of c in semantics $\rightsquigarrow_{(\mathsf{emp},\eta_0)}$ does not produce either av or wrong in some state, then starting from the related state, the (concrete) execution of c in semantics $\rightsquigarrow_{(p_0,\eta)}$ does not either. We will use this fact to prove the theorem. Let (s,h) be a state in $p * p_0$. Heap h can be split uniquely into some heaps $h_p \cdot h_{p_0}$, where $h_p \in p$ and $h_{p_0} \in p_0$. The substate (s,h_p) is related to the state (s,h) itself, by the relation $R * \Delta$. Thus, by the simulation theorem, to show that c produces neither wrong nor av in $\rightsquigarrow_{(p_0,\eta)}$ semantics starting from state (s,h), we only need to prove that

$$c,(s,h_p) \not\rightsquigarrow_{(\mathsf{emp},\eta_0)} \mathsf{wrong} \;\wedge\; c,(s,h_p) \not\rightsquigarrow_{(\mathsf{emp},\eta_0)} \mathsf{av}.$$

By lemma 11, the above is equivalent to

$$c,(s,h_p) \not\rightsquigarrow_{\eta_0} \mathsf{wrong} \;\wedge\; c,(s,h_p) \not\rightsquigarrow_{\eta_0} \mathsf{av}.$$

Since c never generates av in semantics $\rightsquigarrow_{\eta_0}$, we only need to prove the first conjunct. We prove it using the assumption of the theorem and lemma 10. Since $\eta \models \Gamma * p$, by lemma 10,

$$\eta_0 \models \Gamma.$$

This implies that the triple $\{p\}c\{q\}$ holds, because $\Gamma \models \{p\}c\{q\}$ by the assumption. The truth of the triple ensures the required conjunct, which says that c does not fault from (s,h_p) in the semantics $\rightsquigarrow_{\eta_0}$, because the initial state (s,h_p) satisfies the precondition p of the triple. $\square$

5.3 Examples

In Chapter 3 we introduced the concept of a separation context and illustrated it with several examples in Section 3.4.1. Here, we revisit those examples and give a proof in Separation logic or a counter example, for each of these examples.

We first consider a program

$$\begin{array}{l} \mathit{prog}_1 : \\ \quad \mathsf{new}(x); \\ \quad [x] := 47; \\ \quad \mathsf{dispose}(x) \end{array}$$

and prove that it satisfies a specification $\{\mathsf{emp}\} - \{\mathsf{emp}\}$ in an environment in which a memory manager with the free list resource environment is defined. By the frame rule it follows that for any predicate p, the program also satisfies specification $\{p\} - \{p\}$. In fact, from the frame rule follows that the program satisfies the most general specification $\{\mathsf{true}\} - \{\mathsf{true}\}$. Theorem 7 then ensures that program prog_1 is a separation context for memory manager module and the precondition of the specification, which is in this case true and means for any precondition.

We first give a proof of the program.

$$\begin{array}{l} \mathit{prog}_1 : \\ \quad \{\mathsf{emp}\} \\ \quad \mathsf{new}(x); \\ \quad \{x \mapsto -\} \\ \quad [x] := 47; \\ \quad \{x \mapsto 47\} \\ \quad \mathsf{dispose}(x) \\ \quad \{\mathsf{emp}\} \end{array}$$

We assumed a specification environment

$$\Gamma = \{\mathsf{emp}\}\ \mathsf{new}(x)\ \{x \mapsto -\}[x],\ \{x \mapsto -\}\ \mathsf{dispose}(x)\ \{\mathsf{emp}\}$$

and a module environment η which maps $\mathsf{new}()$ and $\mathsf{dispose}()$ to the greatest relations satisfying

$$\begin{array}{ll}\mathsf{new}(x): & \{\mathsf{list}(a\cdot\alpha,\mathsf{ls})\} - \{\mathsf{list}(\alpha,\mathsf{ls}) * x \mapsto a\}[x,\mathsf{ls}] \\ & \{\mathsf{list}(\varepsilon,\mathsf{ls})\} - \{\mathsf{list}(\varepsilon,\mathsf{ls}) * x \mapsto -\}[x,\mathsf{ls}] \\ \mathsf{dispose}(x): & \{\mathsf{list}(\alpha,\mathsf{ls}) * x \mapsto a\} - \{\mathsf{list}(a\cdot\alpha,\mathsf{ls})\}[\mathsf{ls}].\end{array}$$

If we denote by p_0 the predicate $\exists\alpha.\ \mathsf{list}\ (\alpha,\mathsf{ls})$, then the specifications can be rewritten as

$$\begin{array}{ll}\mathsf{new}(x): & \{p_0 * \mathsf{emp}\} - \{p_0 * x \mapsto a\}[x,\mathsf{ls}] \\ & \{p_0 * \mathsf{emp})\} - \{p_0 * x \mapsto -\}[x,\mathsf{ls}] \\ \mathsf{dispose}(x): & \{p_0 * x \mapsto a\} - \{p_0 * \mathsf{emp})\}[\mathsf{ls}].\end{array}$$

This shows that the module environment η satisfies the specification environment $\Gamma * \mathsf{list}\ (\alpha,\mathsf{ls})$ as required by Theorem 7. We will also use this fact in the examples that follow.

Lets consider now the following program

```
prog_2 :
  dispose(x);
  [x] := 47;
```

and its specification $\{x \mapsto -\} - \{\mathsf{true}\}$. We argue that this specification is incorrect, and in fact, no specification with precondition $x \mapsto -$ is correct.

```
prog_2 :
  {x ↦ −}
  dispose(x);
  {emp}
  [x] := 47;
  ???
```

The command $[x] := 47$ requires variable x to point to an allocated memory location in order not to fault. Since the precondition is emp, this is not the case and the command faults, meaning that the given specification cannot be proved. Actually, the specification is incorrect. By the similar reasoning we show that the program faults also for the precondition emp. In fact, there is no precondition for which this program does not fault, so it is never a separation context.

Finally, we consider program

$$prog_3 :$$
$$[81] := 42.$$

The proof that it satisfies the specification $\{81 \mapsto -\} - \{81 \mapsto 42\}$ is straightforward, and hence the program is a separation context for the memory manager module and precondition $\{81 \mapsto -\}$. However, by the reasoning similar as for $prog_2$ it follows that starting from a state satisfying precondition emp, the program $prog_3$ always faults, and hence it is not a separation context for that precondition.

5.4 Smallfoot and separation contexts

Smallfoot [22] is an automatic verification tool based on Separation logic. It is designed to check certain separation logic specifications of sequential and concurrent programs that manipulate recursive dynamically-allocated linked data structures.

As we have already mentioned, Smallfoot can be used to determine whether a given client program is a separation context or not. In this chapter we have presented a result which ensures that once we prove a certain specification of a program in an environment in which a module is assumed, then the program is a separation context. We use Smallfoot to automatically prove that a program meets given specification. We stress here that it is not us who developed Smallfoot nor it is a contribution of this thesis in any way, but because we find Smallfoot practically useful for our theory, we illustrate its usage here.

First, we give a Smallfoot implementation of a stack, given in Table 5.1. The stack is implemented by a linked list, with usual operations push() and pop() and operation bad() which returns to a client a pointer to the top of the stack. The parameters of the push() and pop() operations are values, while the parameter of the operation bad() is a reference. We assume that a list has two fields, hd for keeping data, in this case an integer value, and tl for keeping the pointer to the next node in the list. Stack is initially set to the empty stack, that is, pointer top is initially NULL. Operation push allocates a new node, saves into it a value kept in the parameter and links the node to the list. Operation pop returns to the client data kept in the top element of the stack and deallocates the node that kept that value. This is all done in an environment where stack with pointer top to its top is assumed to be a "protected" resource, which together with its operations forms a module and can be used by other programs. We achieve this by declaring a resource called stack and enclosing the body of each of the module operations, in our case bodies of operations pop(), push() and bad(), with the conditional critical region (with statement). Leaving out the pre and post conditions of the module operations means that we do not assume any extra memory apart from the resource invariant which must be preserved by the operations.

To consider a client of the module, one need to write a code of the client in the function main(). This code contains calls to the operations of the module. Then, the verification proceeds by verifying all the operations and checking the validity of the composition that consists of the operation init() and program given in main().

We now consider one client of the stack module. We add the following code to the file stack_sum_module.sf.

```
main()[emp]{
  local x, sum, y;
  push(5); push(42); push(17);
  sum = 0;
  pop(x);
  while(x! = NULL){
    sum = sum + x;
    pop(x); }
  push(sum);
}[emp]
```

The program main() first pushes three elements onto the stack which is initially empty and then it sums all the values kept in the stack by popping the values from the stack. Finally, the sum is put back onto the stack. The program uses only operations push() and pop() to handle the stack. When we run Smallfoot on the file stack_sum_module.sf, we get the following output.

```
ivanam: /Smallfoot/smallfoot ivanam$ bin/smallfoot-0.1_Darwin_8.3.0_
```

```
powerpc EXAMPLES/stack_sum_module.sf
   Function pop
   Function push
   Function bad
   Function main
   Function init
   Function compose_init_main
   Valid
```

This tells us that the all the module operations are correct, they respect the resource invariant. Program main() is also correct, it properly uses the module. According to our result, program main() is a separation context for the stack module, given in Table 5.1 and a precondition emp. Moreover, using the frame rule, we can conclude that it is a separation context for stack module and *any* precondition *r* which describes storage disjoint from the resource invariant of the stack module and in which variables x and sum do not figure. Namely, Smallfoot would validate program main() for any precondition that describes some storage disjoint from the resource invariant of the stack module.

Let us now consider the following code and add it to the file stack_sum_module.sf instead of the code for client1.

```
main()[emp]{
  local x, sum, z;
  push(5); push(42); push(17);
  sum = 0;
  bad(z;);
  if (z! = NULL)
    x = z → hd;
  while (x! = NULL){
    sum = sum + x;
    pop(x); }
  push(sum);
}[emp]
```

The program main() is similar to the previous program, except that it takes the first element off the stack using the bad(x). The top of the stack is placed into the variable *x* and the client then dereferences *x*, i.e. the top of the stack, to fetch the data kept in it. This is what Smallfoot outputs.

```
ivanam: /Smallfoot/smallfoot ivanam$ bin/smallfoot-0.1_Darwin_8.3.0_
powerpc EXAMPLES/stack_sum_module.sf
   Function pop
   Function push
   Function bad
   Function main File "EXAMPLES/stack_sum_module.sf", line 47,
characters 26-36:  ERROR: lookup z->hd in 0!=z
   Function init
   Function compose_init_main
   NOT Valid
```

Smallfoot validates all the operations of the module again, but this time there is a problem with the client program. Namely, Smallfoot cannot validate the dereferencing of the variable *z*, which points to the top of the stack. The program is clearly not a separation context, as it tampers with the memory that "belongs" to the module.

These examples with Smallfoot have been given just to underline the main point of this chapter: Theorem 7, which establishes a connection between separation contexts and logic, furnishes us with a method whereby we can use proof tools to, in some cases, automatically check for separation contexts.

Table 5.1: Contents of file stack_sum_module.sf

```
tl, hd;
init() {
    top = NULL;
}
resource stack (top) [list(top)]
pop(x){
    local t;
    with stack when (true){
        t = top;
        if (top != NULL){
            x = top → hd;
            top = top → tl;
            dispose t;
        }
        else
            x = NULL;
}}
push(y){
    local t;
    with stack when (true){
        t = new();
        t → hd = y;
        t → tl = top;
        top = t;
}}
bad(x; ){
    with stack when (true){
        x = top;
}}
```

6

Power Simulation

Proving data refinement in a low-level programming setting is particularly difficult. Swinging pointers between a module and a client program results in existence of cross-boundary pointers and dereferencing cross-boundary pointers results in exposure of the internal state of a module; that leads to broken abstraction. This can be worked out using a forward simulation method which is modified so that it only cares about programs which do not dereference cross-boundary pointers. However, the forward simulation method cannot deal with space optimizing refinements. This can be seen in our restriction to growing relations (Definition 14).

The power simulation method that we introduce in this chapter is a general and powerful method for proving data refinement. It enables proving space optimizing refinements by disallowing mechanisms by which the space optimizations can be detected. For instance, the equivalence of the doubly-linked and XOR-linked list representations of doubly-ended queues can be proved using power-simulation method, while forward simulation method cannot deal with this problem.

6.1 Motivation

So far, we have built a theory which gives us a sound forward simulation method for proving data refinement, even with memory manipulating constructs in the programming language. Tackling cross-boundary pointers, pointers from a client to the internal state of a module, became easy once we defined our method to ensure simulation only for non-faulting programs, i.e. separation contexts. Ownership transfer and similar important concepts in low-level programming, raised by the existence of cross-boundary pointers, are no longer a problem. Still, our forward simulation method falls short of handling space-optimizing refinements. This prevents us from proving, for instance, the equivalence of the doubly-linked list and XOR-linked list representations of the doubly ended queues which by intuition, should be the same. The problem arises when we want to prove that the XOR-linked list representation data refines and is a space-optimization of the doubly-linked list one. There are client programs, even separation contexts, which can detect the difference between the two representations and that breaks data refinement. To disallow such things, in our forward simulation method we restricted refinement relations only to the "growing" ones. In practice, that prevents us from even considering simulation for space-optimizing refinements.

We now examine what happens if we remove the restriction that our refinement relations have to be growing, and in that way allow reasoning about space-optimizing refinements. We illustrate the problem with a simple example, rather than with somewhat complex doubly-linked and XOR-linked lists.

We assume that program states consist only of heaps; this does not affect the point that we want to demonstrate here. Suppose that the module interface MOp is $\{\mathsf{init}, \mathsf{final}, f\}$. Let $(p, \eta), (q, \varepsilon)$ be semantic modules for MOp defined as follows:

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

$$\begin{array}{lrcl}
(q,\varepsilon): & h \in q & \overset{def}{\Leftrightarrow} & h = [] \\
& h[\varepsilon(\mathsf{init})]v & \overset{def}{\Leftrightarrow} & 1 \notin \mathsf{dom}(h) \ \wedge \ v = h \\
& h[\varepsilon(f)]v & \overset{def}{\Leftrightarrow} & v = h \\
& h[\varepsilon(\mathsf{final})]v & \overset{def}{\Leftrightarrow} & v = h
\end{array}$$

Module (q, ε) does not data-refine (p, η). To see the reason, consider a complete command that consists of a single atomic operation $\mathsf{cons}(2, 1)$, i.e. $c = \mathsf{init}; \mathsf{cons}(2, 1); \mathsf{final}$. Command $\mathsf{cons}(2, 1)$ allocates one new location initialized to 0 and assigns its address to location 2; in case that location 2 is not allocated initially, $\mathsf{cons}(2,1)$ generates wrong.

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

When the complete command c is run with module (q,ε) from $[2{\to}0]$, it can output $[1{\to}0, 2{\to}1]$:

$$c, [2{\to}0] \rightsquigarrow_{(q,\varepsilon)} [1{\to}0, 2{\to}1].$$

However, if the command is executed 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 location 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, if we used forward simulation to prove this, the abstraction would be broken as this example suggests. It is because, when R is a relation defined by

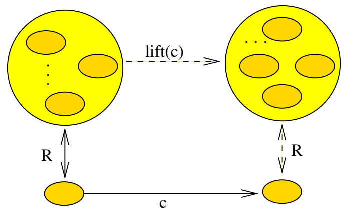
$$h[R]h' \overset{def}{\Leftrightarrow} h = [1{\to}2] \wedge h' = []$$

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

We develop a method for proving data refinement that handles both cross-boundary pointers and space-optimizing refinements at the same time, a *power simulation* method. A client program can detect the difference between the abstract module and its optimization by identifying which memory locations are allocated in the abstract module's internal state and not in the concrete module's one. This can be done using the allocation-status testing mechanism, explained in the introduction of the thesis. The key idea of a power simulation method is to allow the concrete module to space optimize only nondeterministically allocated memory in the abstract module. That makes it impossible for a client program to determine which locations are used, i.e. to successfully perform allocation status testing.

Power-simulation method works by ensuring relation preservation between a state and a set of states, see Figure 6.1. It is exactly this state set in the abstract program execution that enables us express the nondeterminism in memory allocation. Recall that forward simulation involves relations between pairs of states, unlike here.

Figure 6.1: Power simulation



6.2 Background

In this section we introduce the storage model which we will assume in the development of the power simulation method. In the first part of the thesis, in work with forward simulation, we assumed the underlying storage model was the most general one, a partial commutative monoid with injective operation. However, our work on power simulations is specific to a more concrete storage model, as it is addressing the problems that arise in this model. We also revisit the notion of a local action and add a new requirements for the actions (commands) to be considered in the power simulation method.

6.2.1 Storage Model and Finite Local Action

We take RAM for the storage model, which is an instance of the general (partial commutative monoid) model.

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

For simplicity, and without any loss of generality, we assume that state has only the heap component, i.e. that there is no stack. However, stack can be viewed as a preallocated part of the heap. Just as before, a state $h \in \mathsf{State}$ in the model is a finite mapping from locations to integer values; the domain of h denotes the set of currently allocated memory locations, and the "action" of h the contents of those allocated locations. The addresses are positive natural numbers, and so, they can be manipulated by arithmetic operations.

Recall that $(H, \cdot, [])$, with $\cdot$ defined as in Chapter 3 and $[]$ being empty heap, is a partial commutative monoid, and so all the properties introduced in Chapter 3 hold for this model, too. Here, we also use a resource preorder $\sqsubseteq_L$ parameterized by finite location set L: $h \sqsubseteq_L h'$ iff h' has at least as many allocated locations as h as far as L is concerned, i.e.

$$h \sqsubseteq_L h' \iff \mathsf{dom}(h) \cap L \subseteq \mathsf{dom}(h') \cap L.$$

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

$$p,q \in \mathsf{Pred} \overset{\text{def}}{=} \wp(\mathsf{State}) \qquad p*q \overset{\text{def}}{=} \{h_p \cdot h_q \mid h_p \in p \wedge h_q \in q\} \qquad \mathsf{true} \overset{\text{def}}{=} \mathsf{State}.$$

6.2.2 Finite local actions

We return to the notion of a local action. Recall that an *action* r is a relation from State to State $\cup$ {wrong}; action r outputs an error (wrong), if it accesses memory beyond the current state. Then, if $\neg h[r]\mathsf{wrong}$, state h contains all the locations that r dereferences, except the newly allocated locations. As in separation logic, we write $\mathsf{safe}(r,h)$ to indicate this (i.e., $\neg h[r]\mathsf{wrong}$).

We have assumed in work with forward simulations, that all the commands that we are dealing with are General local actions, i.e. actions satisfying safety monotonicity, frame property and contents independence. Here, we introduce a *finite local action*, which in addition to these three properties also needs to satisfy *finite access property*. Intuitively, these four properties mean that each execution of the action accesses only finitely many heap locations. Some of the locations are accessed directly by pointer dereferencing, so that the contents of the locations affects the execution, while the other remaining locations are accessed only indirectly by the allocation-status testing, so that the execution only depends on the allocation status of the memory locations, not their contents. The only new property here is the finite access property.

- **Finite Access Property**: For all h_0 and h_0', if $\mathsf{safe}(r,h_0)$ and $h_0[r]h_0'$, then

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

We again give the other three properties, but now formulated in the memory model assumed in this chapter.

- **Safety Monotonicity**: For all states h and h_1 such that $h \# h_1$, if $\neg h[r]\mathsf{wrong}$, then $\neg h \cdot h_1[r]\mathsf{wrong}$.
- **Frame Property**: For all states h, h' and h_1 with $h \# h_1$, if $\neg h[r]\mathsf{wrong}$ and $h \cdot h_1[r]h'$ then there is a substate $h'_0 \sqsubseteq h'$ such that $h'_0 \# h_1$, $h'_0 \cdot h_1 = h'$ and $h[r]h'_0$.
- **General Contents Independence** For all states h_0, h_1, h'_0 if

  $$h_1 \# h_0 \ \wedge\ \neg h_0[r]\mathsf{wrong} \ \wedge\ (h_0 \cdot h_1)[r](h'_0 \cdot h_1)$$

  then we have

  $$\forall h_2 \in H.\ h_2 \sqsubseteq h_1 \Longrightarrow (h_0 \cdot h_2)[r](h'_0 \cdot h_2).$$

The finite access property expresses the converse of the frame property; 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, expressed by the condition $h_1 \# h_0 \wedge h_1 \# h'_0$, or indirectly accessed locations, i.e. $\mathsf{dom}(h_1) \cap L = \emptyset$. Note that the finite set L here denotes the set of locations indirectly accessed by the computation $h_0[r]h'_0$.

The used-location set L in the definition of the finite access property plays a crucial role in the semantics of programs; it lets us interpret sequential composition of commands, by making finite access property preserved by the sequential composition. We explain this role of L by comparing the property with the following stronger version:

- **Stronger Finite Access Property**: For all h_0 and h'_0, if $\mathsf{safe}(r, h_0)$ and $h_0[r]h'_0$, then

  $$\forall h_1.\,(h_1 \# h_0 \wedge h_1 \# h'_0) \ \Rightarrow\ h_0 \cdot h_1[r]h'_0 \cdot h_1.$$

Note that this new definition does not use the set L of indirectly accessed locations, and expresses more directly that *all* computations from smaller states can be extended. The problem is that the new property is not preserved by the sequential composition. For local actions r and r′, let $\mathsf{seq}(r, r')$ be an action defined as follows:

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

It is straightforward to check that the action $\mathsf{seq}(r, r')$ is also local. However, seq does not preserve the stronger finite access property. To see this, consider the following two local actions r and r':

$$\begin{aligned} h[r]v &\iff 2 \notin dom(h) \wedge v = h \cdot [2 \mapsto 0] \\ h[r']v &\iff ((\exists h_0.\ h = h_0 \cdot [2 \mapsto 0]) \implies h_0 = v) \wedge \\ &\qquad (\neg(\exists h_0.\ h = h_0 \cdot [2 \mapsto 0]) \implies v = \mathsf{wrong}) \end{aligned}$$

The first local relation allocates location 2; if 2 is already in the initial state, the relation diverges. The second relation r' disposes location 2. Both r and r' satisfy the above stronger finite access property, but their sequential composition $\mathsf{seq}(r, r')$ does not. To understand why, consider the execution $[1 \mapsto 0]\mathsf{seq}(r, r')[1 \mapsto 0]$. Since executing $\mathsf{seq}(r, r')$ from $[1 \mapsto 0]$ is safe, by the stronger finite access property we can tack on any state disjoint from the input and output states of the command. However, it is not the case that $[1 \mapsto 0, 2 \mapsto 0]\mathsf{seq}(r, r')[1 \mapsto 0, 2 \mapsto 0]$, because the first command in the sequential composition requires location 2 not to be allocated. On the other hand, the original finite access property is preserved by the sequential composition. In the considered example, location l would be a member of L – a set of indirectly accessed locations, and the heap $[2 \mapsto -]$ does not fulfill the requirement that its domain is disjoint from L.

Definition 20 (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.*

We denote by $\mathcal{F}$ the poset of FLAs ordered by the "graph-subset" relation $\sqsubseteq$. Let $r, r' : \mathsf{State} \uplus \{\mathsf{wrong}\}$. Then,

$$r \sqsubseteq r' \iff \forall h \in \mathsf{State}. \forall v \in \mathsf{State} \cup \{\mathsf{wrong}\}.\, h[r]v \Rightarrow h[r']v.$$

In order to be able to interpret all the possible behaviors of the actions, including the "access-violation" which, as the reader will recall, happens when a client program accesses the internal state of a module with which it interacts, we need to broaden our domain. Let $\mathcal{F}'$ be defined as

$$\mathcal{F}' = \mathcal{F} \cup \{r \mid r : \mathsf{State} \to \mathsf{State} \uplus \{\mathsf{av}, \mathsf{wrong}\} \text{ is FLA}\}.$$

The order in $\mathcal{F}'$ is then

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

For each precise predicate p, we define a *p-protected* execution , a method for constructing new finite local actions. For each r in $\mathcal{F}$, the p-protected execution of r, is a function $\mathsf{prot}(-,p) : \mathcal{F} \to \mathcal{F}'$ defined as follows :

$$\begin{aligned}
h[\mathsf{prot}(r,p)]\mathsf{av} \quad &\Longleftrightarrow \quad \neg h[r]\mathsf{wrong} \;\wedge\; \exists h_0, h_p.\; h = h_0 \cdot h_p \;\wedge \\
&\qquad\quad h_p \in p \;\wedge\; h_0[r]\mathsf{wrong} \\
h[\mathsf{prot}(r,p)]h' \quad &\Longleftrightarrow \quad h[r]h' \\
h[\mathsf{prot}(r,p)]\mathsf{wrong} \quad &\Longleftrightarrow \quad h[r]\mathsf{wrong}.
\end{aligned}$$

Intuitively, $\mathsf{prot}(r,p)$ behaves the same as r, except that whenever r accesses the p-part of the input state, $\mathsf{prot}(r,p)$ generates av, thus indicating that there is an "access violation." That way, if p is the resource invariant of the module, $\mathsf{prot}(r,p)$ notifies all illegal accesses to the module internals, by generating av.

Lemma 13. *For every finite local action $r \in \mathcal{F}$ 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 $\mathsf{State} \times (\mathsf{State} \cup \{\mathsf{wrong}\})$. Note that all of the safety monotonicity, frame property, finite access property and contents independence only concern the state or wrong outputs, and that those properties are satisfied by r. Thus, they are also satisfied by $\mathsf{prot}(r,p)$. $\square$

The function $\mathsf{prot}(-,p)$ is not monotone. To see that, suppose we have two operations r and r' such that $r \sqsubseteq r'$ and let $h[\mathsf{prot}(r,p)]\mathsf{av}$. Then, by the definition of $\mathsf{prot}(r,p)$, we know that $\neg h[r]\mathsf{wrong}$ and there exists a splitting of the state h, $h = h_0 \cdot h_p$, where $h_p \in p$, such that $h_0[r]\mathsf{wrong}$. Then, because $r \sqsubseteq r'$, we also have $h[r']\mathsf{wrong}$, but to have $h[\mathsf{prot}(r',p)]\mathsf{av}$, we also need to know that $\neg h[r']\mathsf{wrong}$, which we do not. This actually means that r' might fault more often than r and this causes trouble. However, this does not affect the semantics of our language.

We redefine $\mathsf{seq}(r,r')$ to be a function from $\mathcal{F}' \times \mathcal{F}' \to \mathcal{F}'$

$$\begin{aligned}
h[\mathsf{seq}(r,r')]v \overset{def}{\Leftrightarrow} \quad &(\exists h'.\, h[r]h' \wedge h'[r']v) \vee \\
&(h[r]\mathsf{wrong} \wedge v{=}\mathsf{wrong}) \vee (h[r]\mathsf{av} \wedge v{=}\mathsf{av}).
\end{aligned}$$

We have shown that $\mathsf{seq}(r,r')$ does not preserve the Stronger finite access property. Nevertheless, it preserves the Finite access property. Moreover, it is a finite local action.

Lemma 14. *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 [80], 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 $\neg h_0[\mathsf{seq}(r, r')]\mathsf{wrong}$ 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 $\neg h_0[\mathsf{seq}(r, r')]\mathsf{wrong}$, by the definition of seq, we have that

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

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

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

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

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

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 $\neg h_0[\mathsf{seq}(r,r')]\mathsf{wrong}$, this substate m_0 should be a safe input for r': $\neg m_0[r']\mathsf{wrong}$. 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,

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

□

The poset $\mathcal{F}$ has a structure rich enough to interpret programs with low-level pointer operations. Particularly interesting are the low-level pointer operations, such as the memory update, allocation and deallocation of a location defined in Table 3.3 in Chapter 3, and a test operation:

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

The operation tests for a location l and integer set I whether l is allocated and it contains a value in I; in that case the test skips. If, on the other hand l is allocated but its value is not in I, the test blocks. In all other cases (i.e., if l is not allocated), the test outputs wrong.

Lemma 15. *The poset $\mathcal{F}$ contains all the concrete low-level operations, i.e. operations defined in Table 3.3 and operation* $\mathsf{test}()$.

Proof. It suffices to show that the operations obey all four properties of finite local actions. We have already proved for all operations in Table 3.3, that they satisfy safety monotonicity, frame property and content independence. It remains to prove that all of them also satisfy finite access property and that the operation test() satisfies all four properties. We only prove that $[l] := i$, where l denotes an address and i denotes an integer, satisfies the finite access property; it is straightforward to prove the remaining cases. Suppose that $\neg h[[l] := i]\mathsf{wrong}$ and $h[[l] := i]h'$. 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, $h \cdot h_1[[l] := i]h' \cdot h_1$. Thus, the required set L of indirectly accessed locations in the finite access property is the empty set. □

Lemma 16. *Both $\mathcal{F}$ and $\mathcal{F}'$ are complete lattices that have the set union as their join operator: for every family $\{r_i\}_{i\in I}$ in the poset, $\bigsqcup_{i\in I} r_i$ is $\bigcup_{i\in I} r_i$.*

Proof. Since both $\mathcal{F}$ and $\mathcal{F}'$ 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 only that $\mathcal{F}'$ is closed. It is well-known that the set of local actions are closed under union [80]. 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 $\neg h_0[r]\mathsf{wrong}$ and $h_0[r]h'_0$. By the definition of r, there is some r_j such that $\neg h_0[r_j]\mathsf{wrong}$ 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

$\mathsf{dom}(h_1) \cap L = \emptyset$. 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

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

Then, there exists r_j such that $\neg h_0 [r_j] \mathsf{wrong}$ 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. □

6.3 Programming Language

For the programming language, we choose Dijkstra's language of guarded commands [30], extended with low-level pointer operations and module operations. The syntax of the language is given by the grammar:

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

where f, a, P are, respectively, chosen from three disjoint sets $\mathsf{aop}, \mathsf{MOp}, \mathsf{pid}$ of identifiers. Set aop denotes a set of all client operations identifiers, set MOp denotes the set of all module operations identifiers and pid is a set of all program identifiers. The first construct f is a module operation declared in the "interface specification" MOp. Before a command gets executed, it is first "linked" to a specific module that implements the interface MOp. This linked module provides the meaning of command f. The second construct a is an atomic operation, which a client can execute without using 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 FLAs is given along with aop, and that under this interpretation, aop includes at least all the pointer operations in Lemma 15. 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 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*.

Table 6.1: Semantics of the Language

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

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

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

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

where seq and prot are defined as follows: for all $h \in \mathsf{State}$ and $v \in \mathsf{State} \cup \{\mathsf{wrong}, \mathsf{av}\}$,

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

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

Note that all the usual constructs of the while language can be expressed in this language, and in particular all the commands of the language defined in Chapter 3. For example, conditional statement **if** $(*l \in I)$ **then** C **else** C' can be expressed as $(\mathsf{test}(l,I);C)$ $[]$ $(\mathsf{test}(l,\bar{I});C')$, where $\bar{I}$ is the complement $\mathsf{Int}-I$ of I. And, the allocation-status testing check2 [1] 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 each atomic client operation accesses the internals of a module, and for such illegal accesses, the semantics generates access violation av. To implement the instrumentation, similarly as in Chapter 3, we parameterize the semantics by a module.

Let $\mathcal{E}$ be the poset of all functions from pid to $\mathcal{F}$ ordered pointwise. Given semantic module (p,η), we interpret command as a continuous function $[\![-]\!]_{(p,\eta)}$ 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 finalization of the module (p,η) before and after $([\![C]\!]_{(p,\eta)}\bot)$. The details of these two interpretations are shown in Table 6.1.

The most interesting part of the semantics lies in the interpretation of the atomic

[1] Code for check2 is given in Section 1.2.3. Recall that check2 implements the allocation status testing mechanism, which can detect space optimizing refinements.

client operations. For each atomic operation a, its interpretation first looks up the original meaning $[\![a]\!]_a \in \mathcal{F}$, which is given when the syntax of the language is defined. Then, the interpretation transforms the meaning into $\mathsf{prot}([\![a]\!]_a, p) \in \mathcal{F}'$, the p-protected execution of $[\![a]\!]_a$.

Lemma 17. *The interpretation in Table 6.1 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 13), 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 14). 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. $\square$

6.4 Power Simulation

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

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

$$\begin{array}{rcl} \mathsf{lft}(r) & : & \wp(\mathsf{State}) \leftrightarrow (\wp(\mathsf{State}) \cup \{\mathsf{wrong}, \mathsf{av}\}) \\ H[\mathsf{lft}(r)]V & \overset{def}{\Leftrightarrow} & (V \subseteq \mathsf{State} \wedge \forall h' \in V.\ \exists h \in H.\, h[r]h') \vee \\ & & ((V = \mathsf{av} \vee V = \mathsf{wrong}) \wedge \exists h \in H.\, h[r]V). \end{array}$$

Given an input state set H, the "lifted command" $\mathsf{lft}(r)$ runs r for all 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). For that purpose we define a *power relation* to be a relation between states and state sets.

$$\mathcal{R} \subseteq (\mathsf{State} \uplus \{\mathsf{wrong}, \mathsf{av}\}) \times (\wp(\mathsf{State}) \uplus \{\{\mathsf{wrong}\}, \{\mathsf{av}\}\}).$$

Intuitively, the power simulation method requires that, to prove this data refinement, we only need to find a "good" power relation $\mathcal{R}$. The power relation $\mathcal{R}$ then must be such that every concrete-module operation $\varepsilon(k)$ "forward-simulates" the corresponding lifted abstract-module operation $\mathsf{lft}(\eta(k))$ by $\mathcal{R}$. The formal 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 for a normal command to "forward-simulate" a lifted command.

To give an answer to the first question we have to define the *expansion* operator which expands the power relation so that it also takes care of the externals of the module, and *admissibility* condition for power relations which allows that the internal representation of the module does not depend on the identities of the locations in its build-up. For the second requirement, we use the operator psim that maps a power-relation pair to a relation on FLAs. We define these sub-components of power simulation, and use them to give the formal definition of power simulation.

6.4.1 The psim operator

The psim operator is similar to the fsim operator given in Definition 12, but it involves power relations instead of ordinary relations. We explain the operator psim. 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' should be possible outcomes of $\mathsf{lft}(r)$ from H up to $\mathcal{R}_1$.

Definition 21 (Power simulation). *Action r' power simulates another action r, denoted by*

$$r'[\mathsf{psim}(\mathcal{R}_0, \mathcal{R}_1)]r$$

iff for all states h' and state sets H,

1. *if $h'[\mathcal{R}_0]H \ \wedge\ \neg H[\mathsf{lft}(r)]\mathsf{wrong} \ \wedge\ \neg H[\mathsf{lft}(r)]\mathsf{av}$, then $\neg h'[r']\mathsf{wrong} \ \wedge\ \neg h'[r']\mathsf{av}$, and*
2. *if $h'[\mathcal{R}_0]H \ \wedge\ \neg H[r]\mathsf{wrong} \ \wedge\ \neg H[r]\mathsf{av}$ then $\forall h'_1.\, h'[r']h'_1 \ \Rightarrow\ \exists H_1.\, H[\mathsf{lft}(r)]H_1 \wedge h'_1[\mathcal{R}_1]H_1$.*

Similarly as for the forward simulation, these two conditions can be depicted by the following diagrams, for easier understanding.

$$\begin{tikzcd}
H \arrow[r, dashed, "\mathsf{lft}(r)"] \arrow[d, leftrightarrow, "\mathcal{R}_0"'] & \mathsf{wrong}\ (\mathsf{av}) \arrow[d, dashed, leftrightarrow, "\mathcal{R}_1"] \\
h' \arrow[r, "r'"'] & \mathsf{wrong}\ (\mathsf{av})
\end{tikzcd}$$

$$\begin{tikzcd}
H \arrow[r, dashed, "\mathsf{lft}(r)"] \arrow[d, leftrightarrow, "\mathcal{R}_0"'] & H_1 \arrow[d, dashed, leftrightarrow, "\mathcal{R}_1"] \\
h' \arrow[r, "r'"'] & h'_1
\end{tikzcd}$$

The first condition ensures that $\mathsf{lft}(r)$ faults more often than r'. The second condition establishes that if the abstract operation $\mathsf{lft}(r)$ does not fault, then the concrete one r' does not fault either and every computation of r' corresponds to some computation of $\mathsf{lft}(r)$.

Note that this definition is the lifted version of fsim in Sec. 4.4; 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 use psim to express the "forward-simulation" of a lifted command by a normal command.

6.4.2 Admissibility

An important subcomponent of the power simulation method is the admissibility condition for power relations. It ensures that the simulation between two modules does not fail only because of the identities of the locations in their representation. In some sense, it allows us to neglect those identities up to a certain point.

Definition 22. *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*[2]

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

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 locations" in these related state sets. It means that if $h[\mathcal{R}]H$, state set H collectively has at least as many free locations as h: for every finite collection L of free locations in h, set H contains states that do not have any of the locations in L, and, moreover, the set H_1 of such states itself collectively has as many free locations as h. To understand the second conjunct more clearly, consider the following power relation $\mathcal{R}_0$

$$h[\mathcal{R}_0]H \overset{def}{\Leftrightarrow} h{=}[3{\to}1] \wedge H{=}\{[3{\to}5]\}.$$

Relation $\mathcal{R}_0$ is admissible, because set $\{[3{\to}5]\}$ has only one state $[3{\to}5]$ that has the exactly same free locations, namely all locations other than 3, as state $[3{\to}1]$. On the other hand, $\mathcal{R}_1$ defined by

$$h[\mathcal{R}_1]H \overset{def}{\Leftrightarrow} h{=}[3{\to}1] \wedge H{=}\{[3{\to}5, 4{\to}5]\}$$

is not admissible. The unique state in $\{[3{\to}5, 4{\to}5]\}$ has an active location 4 that is not free in $[3{\to}1]$. Now, lets have a look at a little bit trickier relation $\mathcal{R}_2$

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

[2]Recall that $h_1 \sqsubseteq_L h$ iff $(\mathsf{dom}(h_1) \cap L) \subseteq (\mathsf{dom}(h) \cap L)$.

Relation $\mathcal{R}_2$ is admissible, even though for all $\mathcal{R}_2$-related h and H, every state in H has more active locations than h. The intuitive reason for this is that for every free location in $[3{\to}1]$, set H contains a state that does not contain the location, and so, it collectively has as many free locations as $[3{\to}1]$. In a sense, by having sufficiently many states, H hides the identity of the additional location n.

6.4.3 The expansion operator

Similarly to the forward simulation method where we extended each relation between two modules with the identity relation to take into account the externals of the module, we extend the relevant power relations, using the $-\otimes\Delta$ operator.

Definition 23. *The expansion $\mathcal{R}\otimes\Delta$ of a power relation $\mathcal{R}$ is a power relation defined as follows:*

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

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. We refer to $\mathcal{R}$ as a "coupling" power relation. It 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.

Lemma 18. *The expansion $\mathcal{R}\otimes\Delta$ of an admissible power relation $\mathcal{R}$ is admissible.*

Proof. Suppose that $H[\mathcal{R}\otimes\Delta]h$. Then, there exist a splitting $h_r * h_0$ of h and a state set H_r such that $H_r[\mathcal{R}]h_r$ and $H = H_r * \{h_0\}$. We first show that H cannot be empty. Let G be $\mathsf{dom}(h_0)$. Then, there exists a subset H_1 of H_r such that

$$H_1 \subseteq H_r \ \wedge\ H_1[\mathcal{R}]h_r \ \wedge\ \forall h_1 \in H_1.\, \mathsf{dom}(h_1) \cap G \subseteq \mathsf{dom}(h_r) \cap G.$$

Since $G = \mathsf{dom}(h_0)$ and $h_0 \# h_r$, the second conjunct of H_1 can be simplified as follows:

$$\forall h_1 \in H_1.\, h_1 \# h_0.$$

Thus, it suffices to show that H_1 is not empty. For this, we use the admissibility of $\mathcal{R}$. Since $H_1[\mathcal{R}]h_r$, by the admissibility of $\mathcal{R}$, set H_1 cannot be empty.

Next, we will prove that for any given finite set G of locations, there is a subset H_2 of H such that

$$H_2 \subseteq H \;\wedge\; H_2[\mathcal{R} \otimes \Delta]h \;\wedge\; \forall h_2 \in H_2.\, \mathsf{dom}(h_2) \cap G \subseteq \mathsf{dom}(h) \cap G.$$

Suppose that we are given a finite set G of locations. Since $H_r[\mathcal{R}]h_r$, there exists a state set H_1 such that

$$H_1 \subseteq H_r \;\wedge\; H_1[\mathcal{R}]h_r \;\wedge\; \forall h_1 \in H_1.\, \mathsf{dom}(h_1) \cap G \subseteq \mathsf{dom}(h_r) \cap G.$$

We will show that $H_1 * \{h_0\}$ is the state set that we are looking for. Since H_1 is a subset of H_r,

$$H_1 * \{h_0\} \subseteq H_r * \{h_0\} = H,$$

and since $H_1[\mathcal{R}]h_r$ and $h = h_r * h_0$, by the definition of $\mathcal{R} \otimes \Delta$,

$$H_1 * \{h_0\}[\mathcal{R} \otimes \Delta]h_r \cdot h_0.$$

Thus, we only need to show that

$$\forall h_2 \in H_1 * \{h_0\}.\, \mathsf{dom}(h_2) \cap G \subseteq \mathsf{dom}(h) \cap G.$$

We show that the required formula holds as follows:

$$\begin{array}{l}
\quad h_2 \in H_1 * \{h_0\} \\
\Longrightarrow \\
\quad \exists h_1.\, h_2 = h_1 \cdot h_0 \;\wedge\; h_1 \in H_1 \\
\Longrightarrow \\
\quad \exists h_1.\, h_2 = h_1 \cdot h_0 \;\wedge\; \mathsf{dom}(h_1) \cap G \subseteq \mathsf{dom}(h_r) \cap G \\
\Longrightarrow \; (\because h_2 = h_1 \cdot h_0 \;\wedge\; h = h_r \cdot h_0) \\
\quad \exists h_1.\, h_2 = h_1 \cdot h_0 \;\wedge\; \mathsf{dom}(h_2) \cap G \subseteq \mathsf{dom}(h) \cap G \\
\Longrightarrow \\
\quad \mathsf{dom}(h_2) \cap G \subseteq \mathsf{dom}(h) \cap G.
\end{array}$$

□

6.4.4 The reasoning principle

We have introduced all the components of the power simulation: admissibility condition, expansion operator and the psim operator. Using the expansion operator and admissibility condition, we can now define the criterion for deciding which power relations should be considered "good" for given modules (q, ε) and (p, η). The criterion is: a power relation should be the expansion $\mathcal{R}{\otimes}\Delta$ of an admissible coupling power relation $\mathcal{R}$ ($\mathcal{R} \subseteq q \times \wp(p)$). The following lemma, which we will prove later in Sec. 6.5, provides the justification of this criteria:

LEMMA 19: For all precise predicates q, p, and all power relations $\mathcal{R} \subseteq q \times \wp(p)$, if $\mathcal{R}$ is admissible, then $\forall r{\in}\mathcal{F}_{noav}.\ \mathsf{prot}(r,q)[\mathsf{psim}(\mathcal{R}{\otimes}\Delta, \mathcal{R}{\otimes}\Delta)]\mathsf{prot}(r,p)$.

To see the significance of this lemma, recall that the forward simulation without restriction to only growing relations fails to be sound mainly because some atomic client operations are not related to themselves by fsim. However, once we have agreed to consider only growing relations with forward simulation, the problem was resolved. The lemma suggests that when using admissible power relation $\mathcal{R}$, we do not have such a problem for psim either: 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 command executed in the concrete module environment power simulates itself when executed in the abstract module environment.

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: $h[\mathsf{ID}]H \overset{def}{\Leftrightarrow} H = \{h\}$.

Definition 24 (Power Simulation)**.** Module (q, ε) power-simulates (p, η) by $\mathcal{R}$ *iff*

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

The definition states that, in order to have one module power simulate another, all the operations of the concrete module need to power simulate the corresponding lifted operations of the abstract module. The init operations need to establish power relation $\mathcal{R} \otimes \Delta$ starting from "identical" "states", the final operations need to establish "identical" "states" from $\mathcal{R} \otimes \Delta$ and all other operations need to preserve the $\mathcal{R} \otimes \Delta$ power relation.

6.5 Soundness of the Power Simulation method

In this section we prove the soundness of the power simulation method. The soundness result ensures that if we prove that one module simulates another using the power simulation method, then we indeed have data refinement between the two modules.

We first need to prove several lemmas, which will be then all tied together in the proof of the soundness. We start with the lemma that guarantees that every atomic client operation is related to itself by psim with respect to some admissible power relation. This will ensure that, having two modules simulate one another, the client operation executed together with the concrete module will behave at least as good as when it is executed with the abstract one.

Lemma 19. *For all precise predicates q, p, and all power relations $\mathcal{R} \subseteq q \times \wp(p)$, if $\mathcal{R}$ is admissible, then* $\forall r{\in}\mathcal{F}_{noav}.\ \mathsf{prot}(r,q)[\mathsf{psim}(\mathcal{R}{\otimes}\Delta, \mathcal{R}{\otimes}\Delta)]\mathsf{prot}(r,p)$.

Proof. Let us denote $\mathsf{prot}(r,p)$ by r_p and $\mathsf{prot}(r,q)$ by r_q. We pick arbitrary $[\mathcal{R}{\otimes}\Delta]$-related h and H such that $\mathsf{lft}(r_p)$ does not generate an error from H. Since they are $\mathcal{R}{\otimes}\Delta$ related, 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 $\mathcal{R}$, set H_p contains a state h_p such that $h_p \sqsubseteq_{\mathsf{dom}(h_0)} h_q$. This condition on h_p ensures the disjointness of h_p and h_0, because $\mathsf{dom}(h_0) \cap \mathsf{dom}(h_q) = \emptyset$. 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 $\neg h_p \cdot h_0[r_p]\mathsf{wrong}\ \wedge\ \neg h_p \cdot h_0[r_p]\mathsf{av}$. This absence of errors of r_p ensures one important property of r: r cannot generate wrong from h_0. To see the reason, note that h_p is in p, and that $\neg h_p \cdot h_0[r]\mathsf{wrong}$ since $\neg h_p \cdot h_0[r_p]\mathsf{wrong}$. So, if $h_0[r]\mathsf{wrong}$, 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 $\neg h[r_q]\mathsf{wrong}$ and $\neg h[r_q]\mathsf{av}$. Since $h = h_0 \cdot h_q$ and $\neg h_0[r]\mathsf{wrong}$, by the safety monotonicity of r, we have that $\neg h[r]\mathsf{wrong}$. Thus, $\neg h[r_q]\mathsf{wrong}$ by the definition of prot. For $\neg h[r_q]\mathsf{av}$, we have to show that

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

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 $\neg h_0[r]\mathsf{wrong}$, it follows that $\neg m_0[r]\mathsf{wrong}$.

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 $\neg h_0[r]\mathsf{wrong}$, 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 $\neg h_0[r]\mathsf{wrong}$, 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 of indirectly accessed locations by the "computation" $h_0 \cdot h_q[r]h_0' \cdot h_q$, which is guaranteed to exist by the finite access property of r. Let L be the location set $L_0 \cup \mathsf{dom}(h_0) \cup \mathsf{dom}(h_0') \cup \mathsf{dom}(h_q)$. Since $h_q[\mathcal{R}]H_p$ and $\mathcal{R}$ is admissible, there is a subset H_1 of H_p such that

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

We note that since $\mathsf{dom}(h_q) \subseteq L$, the last conjunct in the above formula is equivalent to $\forall h_1 \in H_1.\, \mathsf{dom}(h_1) \cap L \subseteq \mathsf{dom}(h_q)$. 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 $m_1 \in H_p \wedge \mathsf{dom}(m_1) \cap L \subseteq \mathsf{dom}(h_q)$. 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 $\mathsf{dom}(n_1) \cap L = \emptyset$ and $\mathsf{dom}(n_2) = \mathsf{dom}(o_2)$. From this property, we obtain a new computation of r_p as follows:

$$
\begin{aligned}
h_q \cdot h_0[r]h_q \cdot h'_0 &\implies n_1 \cdot h_q \cdot h_0[r]n_1 \cdot h_q \cdot h'_0 && (\because r \text{ has FAP[3] property}) \\
&\implies n_1 \cdot o_2 \cdot o_3 \cdot h_0[r]n_1 \cdot o_2 \cdot o_3 \cdot h'_0 && (\because h_q = o_2 \cdot o_3) \\
&\implies n_1 \cdot n_2 \cdot o_3 \cdot h_0[r]n_1 \cdot n_2 \cdot o_3 \cdot h'_0 && (\because r \text{ has CI[4] property}) \\
&\implies n_1 \cdot n_2 \cdot h_0[r]n_1 \cdot n_2 \cdot h'_0 && ;\ (\because r \text{ has FP[5] property}) \\
&\implies m_1 \cdot h_0[r]m && (\because m_1 = n_1 \cdot n_2 \wedge m_1 \cdot h'_0 = m) \\
&\implies m_1 \cdot h_0[r_p]m && (\because \text{definition of } \mathsf{prot}(r,p))
\end{aligned}
$$

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

Now, we need to make sure that all compound commands preserve the property that the atomic operations have. Next lemma guarantees that sequential composition also has this property, i. e., that it preserves psim.

Lemma 20. *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 $r'_0[\mathsf{psim}(\mathcal{R}_0, \mathcal{R}_1)]r_0$ and $r'_1[\mathsf{psim}(\mathcal{R}_1, \mathcal{R}_2)]r_1$, 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:

$$
\begin{aligned}
&\neg H[\mathsf{lft}(\mathsf{seq}(r_0, r_1))]\mathsf{wrong} \wedge \neg H[\mathsf{lft}(\mathsf{seq}(r_0, r_1))]\mathsf{av} \\
\implies &(\because \text{the definition of } \mathsf{lft}(\mathsf{seq}(r_0, r_1))) \\
&\neg H[\mathsf{lft}(r_0)]\mathsf{wrong} \wedge \neg H[\mathsf{lft}(r_0)]\mathsf{av} \\
&\wedge\ (\forall H'.\, H[\mathsf{lft}(r_0)]H' \Rightarrow \neg H'[\mathsf{lft}(r_1)]\mathsf{wrong} \wedge \neg H'[\mathsf{lft}(r_1)]\mathsf{av}) \\
\implies &(\because h[\mathcal{R}_0]H \wedge r'_0[\mathsf{psim}(\mathcal{R}_0, \mathcal{R}_1)]r_0) \\
&\neg h[r'_0]\mathsf{wrong} \wedge \neg h[r'_0]\mathsf{av} \wedge (\forall h'.\, h[r'_0]h' \Rightarrow \exists H'.\, h'[\mathcal{R}_1]H' \wedge H[\mathsf{lft}(r_0)]H') \\
&\wedge\ (\forall H'.\, H[\mathsf{lft}(r_0)]H' \Rightarrow \neg H'[\mathsf{lft}(r_1)]\mathsf{wrong} \wedge \neg H'[\mathsf{lft}(r_1)]\mathsf{av}) \\
\implies & \\
&\neg h[r'_0]\mathsf{wrong} \wedge \neg h[r'_0]\mathsf{av} \\
&\wedge\ (\forall h'.\, h[r'_0]h' \Rightarrow \exists H'.\, h'[\mathcal{R}_1]H' \wedge \neg H'[\mathsf{lft}(r_1)]\mathsf{wrong} \wedge \neg H'[\mathsf{lft}(r_1)]\mathsf{av}) \\
\implies &(\because h'[\mathcal{R}_1]H' \wedge r'_1[\mathsf{psim}(\mathcal{R}_1, \mathcal{R}_2)]r_1)
\end{aligned}
$$

[3]Finite Access Property
[4]Content Independence
[5]Frame Property

$$\neg h[r'_0]\mathsf{wrong} \wedge \neg h[r'_0]\mathsf{av} \wedge (\forall h'.\, h[r'_0]h' \Rightarrow \neg h'[r'_1]\mathsf{wrong} \wedge \neg h'[r'_1]\mathsf{av})$$

$$\Longrightarrow$$

$$\neg h[\mathsf{seq}(r'_0, r'_1)]\mathsf{wrong} \wedge \neg h[\mathsf{seq}(r'_0, r'_1)]\mathsf{av}$$

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:

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

□

The following lemma ensures that the union of psim-preserving commands also preserve psim. This fact then implies that the nondeterministic choice and the recursive call will have "better" behavior when executed in an environment that contains the concrete module instead of the one that contains the abstract one, when we have simulation between the two modules.

Lemma 21. *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:

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

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:

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

□

We now consider the simulation theorem. As discussed before, the simulation theorem provides that the simulation between the modules can be lifted to the whole language. To prove the simulation theorem, we link the facts that we have proved so far.

Theorem 8 (Simulation theorem). *Let $(q,\varepsilon),(p,\eta)$ 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*

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

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 should hold because of Lemma 19. The remaining three cases follow from the property of $\mathsf{psim}(\mathcal{R}\otimes\Delta,\mathcal{R}\otimes\Delta)$ of being closed in Lemma 20 and 21. Namely, $\mathsf{psim}(\mathcal{R}\otimes\Delta,\mathcal{R}\otimes\Delta)$ is closed under arbitrary union and seq. This property directly implies that the induction step goes through for the cases of $C_1[]C_2$ and $C_1;C_2$. For $\mathsf{fix}x.C'$, we note that satisfying the property of being closed under arbitrary union implies that $\mathsf{psim}(\mathcal{R}\otimes\Delta,\mathcal{R}\otimes\Delta)$ is complete,[6] and that this completeness is what we need to prove the induction step for $\mathsf{fix}x.C'$. □

We now have almost everything we need to prove the soundness of the power simulation method. However, the following lemma provides us with one missing link in the proof. Namely, it states that to prove data refinement, it is enough to make sure that all the complete commands are psim-related with respect to the relation ID.

Lemma 22. *A module (q,ε) data-refines another module (p,η) iff for all complete commands C, we have that $[\![C]\!]^c_{(q,\varepsilon)}[\mathsf{psim}(\mathsf{ID},\mathsf{ID})][\![C]\!]^c_{(p,\eta)}$.*

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 $H = \{h\}$. 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

$$\big(\neg h[r']\mathsf{wrong}\ \wedge\ \neg h[r']\mathsf{av}\big)\ \wedge\ \big(\forall h'.h[r']h' \Longrightarrow \{h\}[\mathsf{lft}(r)]\{h'\}\big).$$

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. More specifically,

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

$\mathsf{lft}(r)$ does not generate an error from $\{h\}$ iff r does not generate an error from h and $\mathsf{lft}(r)$ 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' neither generates 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.

$$\neg h[[\![C]\!]^c_{(p,\eta)}]\mathsf{wrong} \wedge \neg h[[\![C]\!]^c_{(p,\eta)}]\mathsf{av} \implies$$
$$\neg h[[\![C]\!]^c_{(q,\varepsilon)}]\mathsf{wrong} \wedge \neg h[[\![C]\!]^c_{(q,\varepsilon)}]\mathsf{av} \wedge (\forall h'.\ h[[\![C]\!]^c_{(q,\varepsilon)}]h' \implies h[[\![C]\!]^c_{(p,\eta)}]h').$$

The resulting unrolled statement proves the lemma, because both sides of "iff" in the statement are the same. □

Finally, we present the proof of the soundness of the power simulation method. It relies on the results we have proved so far.

Theorem 9 (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, $[\![C]\!]_{(q,\varepsilon)}\ [\mathsf{psim}(\mathsf{ID},\mathsf{ID})]\,[\![C]\!]_{(p,\eta)}$, because, then, module (q,ε) should data-refine (p,η) by Lemma 22.

We pick an arbitrary complete program C. Let μ be an environment that maps all program identifiers to the empty relation. Since the empty relation is a FLA, μ is a well-defined environment. By Lemma 21, $\mu(P)[\mathsf{psim}(\mathcal{R}\otimes\Delta,\mathcal{R}\otimes\Delta)]\mu(P)$ 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 8)

$[\![C]\!]_{(q,\varepsilon)}\mu[\mathsf{psim}(\mathcal{R}\otimes\Delta,\mathcal{R}\otimes\Delta)]\,[\![C]\!]_{(p,\eta)}\mu$

$\implies$ ($\because$ Lemma 20)

$\mathsf{seq}(\mathsf{seq}(\varepsilon(\mathsf{init}),[\![C]\!]_{(q,\varepsilon)}\mu),\varepsilon(\mathsf{final}))[\mathsf{psim}(\mathsf{ID},\mathsf{ID})]\mathsf{seq}(\mathsf{seq}(\eta(\mathsf{init}),[\![C]\!]_{(p,\eta)}\mu)),\eta(\mathsf{final}))$

$\implies$ ($\because$ the definition of $[\![-]\!]^c$)

$[\![C]\!]^c_{(q,\varepsilon)}[\mathsf{psim}(\mathsf{ID},\mathsf{ID})]\,[\![C]\!]^c_{(p,\eta)}$

This complete the soundness proof by the remark we made in the beginning of the proof. □

6.6 Example

We demonstrate power simulation using the semantic modules (q,ε) and (p,η) that, respectively, correspond to counter2 and counter3 in Table 1.7. 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 locations, namely location 1 and a newly allocated one, to track the value of the counter, while counter2 uses only location 1 for the same purpose. The corresponding semantic modules, (q,ε) for counter2 and (p,η) for counter3, are defined in Table 6.2. Note that the resource invariant p indicates that counter3 uses two locations 1 and n internally, and the invariant q shows that counter2 uses only one location, and that is 1, internally. We will show that the space saving in counter2 is correct, by proving that (q,ε) power-simulates (p,η).

We first need to make sure that we are dealing only with finite local actions. For that we prove the following theorem.

Theorem 10. *All the operations defined in Table 6.2 are finite local actions.*

Proof. We only prove that $\eta(\mathsf{inc})$ satisfies finite access property, one of the more complex cases. It can be similarly proved that the other operations also satisfy the finite access property. We leave out the other properties, as they should be easier to prove.

Let h_0 be a state such that $\mathsf{safe}(\eta(\mathsf{inc})$. The definition of $\eta(\mathsf{inc})$ then tells us that $1 \in \mathsf{dom}(h_0)$ and $h_0(1) \in \mathsf{dom}(h_0)$. Let h_0 be such that $h_0[\eta(\mathsf{inc})]h_0'$, i.e., let $h_0' = h_0[h_0(1) \mapsto (h_0(h_0(1))+1)]$.

Allow L, the set from the finite access property, to be an empty set, $L = \emptyset$. Let h_1 satisfy the condition of the finite access property, i.e. $h_1 \# h_0 \wedge h_1 \# h_0'$. Clearly, $h_1 \# L$, as L is the empty set.

Now, by the definition of $\eta(\mathsf{inc})$, since $1 \in \mathsf{dom}(h_0)$ and $h_0(1) \in \mathsf{dom}(h_0)$, $\mathsf{safe}(\eta(\mathsf{inc}), h_0 \cdot h_1$ is true and

$$h_0 \cdot h_1[\eta(\mathsf{inc})]h_0 \cdot h_1[h_0 \cdot h_1(1) \mapsto (h_0 \cdot h_1(h_0 \cdot h_1(1))+1)].$$

$$
\begin{array}{rcl}
h \in p & \overset{def}{\Leftrightarrow} & \exists n,n'.\, n' \neq 1 \land n \geq 0 \land h = [1 \to n', n' \to n] \\
h[\eta(\mathsf{init})]v & \overset{def}{\Leftrightarrow} & \text{if } (1 \notin \mathsf{dom}(h)) \text{ then } v = \mathsf{wrong} \text{ else } \exists n.\, n \notin \mathsf{dom}(h) \land v = h[1 \to n] \cdot [n \to 0] \\
h[\eta(\mathsf{inc})]v & \overset{def}{\Leftrightarrow} & \text{if } (1 \notin \mathsf{dom}(h) \lor h(1) \notin \mathsf{dom}(h)) \text{ then } v = \mathsf{wrong} \text{ else } v = h[h(1) \to (h(h(1)) + 1)]) \\
h[\eta(\mathsf{read})]v & \overset{def}{\Leftrightarrow} & \text{if } (1 \notin \mathsf{dom}(h) \lor h(1) \notin \mathsf{dom}(h) \lor 3 \notin \mathsf{dom}(h)) \text{ then } v = \mathsf{wrong} \text{ else } v = h[3 \to h(h(1))] \\
h[\eta(\mathsf{final})]v & \overset{def}{\Leftrightarrow} & \text{if } (1 \notin \mathsf{dom}(h) \lor h(1) \notin \mathsf{dom}(h)) \text{ then } v = \mathsf{wrong} \\
 & & \text{else } \exists h_0.\, v = h_0[1 \to 0] \land h = h_0 \cdot [h(1) \to h(h(1))] \\
h \in q & \overset{def}{\Leftrightarrow} & \exists n.\, n \geq 0 \land h = [1 \to n] \\
h[\varepsilon(\mathsf{init})]v & \overset{def}{\Leftrightarrow} & \text{if } (1 \notin \mathsf{dom}(h)) \text{ then } v = \mathsf{wrong} \text{ else } v = h[1 \to 0] \\
h[\varepsilon(\mathsf{inc})]v & \overset{def}{\Leftrightarrow} & \text{if } (1 \notin \mathsf{dom}(h)) \text{ then } v = \mathsf{wrong} \text{ else } v = h[1 \to (h(1) + 1)] \\
h[\varepsilon(\mathsf{read})]v & \overset{def}{\Leftrightarrow} & \text{if } (1 \notin \mathsf{dom}(h) \lor 3 \notin \mathsf{dom}(h)) \text{ then } v = \mathsf{wrong} \text{ else } v = h[3 \to h(1)] \\
h[\varepsilon(\mathsf{final})]v & \overset{def}{\Leftrightarrow} & \text{if } (1 \notin \mathsf{dom}(h)) \text{ then } v = \mathsf{wrong} \text{ else } v = h[1 \to 0]
\end{array}
$$

Table 6.2: Definition of Module (p, η) and (q, ε)

Because both 1 and $h_0(1)$ are in the h_0 part of the heap, this is equivalent to

$$h_0 \cdot h_1[\eta(\mathsf{inc})]h_0[h_0(1) \mapsto (h_0(h_0(1)) + 1)] \cdot h_1.$$

which is $h_0 \cdot h_1[\eta(\mathsf{inc})]h_0' \cdot h_1$ and that is exactly what we wanted to prove. □

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

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

Intuitively, $h[\mathcal{R}]H$ means that all the states in H and state h represent the same counter having the same value. Before we can go on, we need to prove that $\mathcal{R}$ is admissible.

Lemma 23. *Power relation $\mathcal{R}$ defined as above is admissible.*

Proof. Let h and H be related by $\mathcal{R}$. Then there exist a finite set L and an integer $n \geq 0$ such that $h = [1 \to n]$ and $H = \{[1 \to n', n' \to n] \mid n' \notin L \cup \{1\}$. Firstly, H is not empty, as the set L is finite.

Secondly, let G be any finite set of locations. Then, set of heaps

$$H_1 = \{(1 \mapsto n', n' \mapsto n | n' \notin L \cup G \cup \{1\}\}.$$

is a subset of H such that $H_1 \neq \emptyset$, $h[\mathcal{R}]H_1$ and for all $h_1 \in H_1$ it is true that $\mathsf{dom}(h_1) \cap G \subseteq \mathsf{dom}(h) \cap G$. □

The next step is to show that all the corresponding module operations of (q, ε) and (p, η) are related by psim.

Lemma 24. *Module (q, ε) power simulates module (p, η).*

Proof. Here we only show that $\varepsilon(\mathsf{init})$ and $\eta(\mathsf{init})$ are $\mathsf{psim}(\mathsf{ID}, \mathcal{R}\otimes\Delta)$-related. The other operations can be examined in a similar fashion.

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 $\varepsilon(\mathsf{init})$ 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 error from $\{h\}$. Then, $\eta(\mathsf{init})$ cannot output wrong from h, and so, location 1 should be in $\mathsf{dom}(h)$. From this, it follows that the concrete initialization $\varepsilon(\mathsf{init})$ does not generate an error from h either. We now check the non-error outputs of $\varepsilon(\mathsf{init})$. When started from h, the concrete initialization $\varepsilon(\mathsf{init})$ has only one non-error output, namely state $h[1{\to}0]$. We split this output state $h[1{\to}0]$ into $[1{\to}0]$ and the remainder h_0. Let L be $\mathsf{dom}(h_0)$. It is easy to check, using the definition of $\mathcal{R}$, that the first part $[1{\to}0]$ of the splitting is $\mathcal{R}$-related to $H_r = \{[1{\to}n', n'{\to}0] \mid n' \notin L\}$. Thus, extending $[1{\to}0]$ and H_r by the remainder h_0 gives $\mathcal{R}\otimes\Delta$-related state $[1{\to}0] \cdot h_0 = h[1{\to}0]$ and state set $H_r * \{h_0\}$. Note here that both these compositions are defined, as we made sure that there are no states in H_r that contain locations from h_0.

Now, the state set $H_r * \{h_0\}$ is equal to $\{h[1{\to}n'] \cdot [n'{\to}0] \mid n' \notin \mathsf{dom}(h)\}$, 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 $h[1{\to}0]$ 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 location 2:

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

Then, (q,ε) no longer data-refines (p,η);[7] by testing the allocation status of location 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 $\varepsilon(\mathsf{init})[\mathsf{psim}(\mathsf{ID},\mathcal{R}\otimes\Delta)]\eta(\mathsf{init})$, then $\mathcal{R}$ cannot be admissible. To see the reason, suppose that $\varepsilon(\mathsf{init})[\mathsf{psim}(\mathsf{ID},\mathcal{R}\otimes\Delta)]\eta(\mathsf{init})$. When $\varepsilon(\mathsf{init})$ and $\mathsf{lft}(\eta(\mathsf{init}))$ are run from ID-related $[1{\rightarrow}0]$ and $\{[1{\rightarrow}0]\}$, $\varepsilon(\mathsf{init})$ 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 $\mathcal{R}\otimes\Delta$-related to $\{[1{\rightarrow}2, 2{\rightarrow}0]\}$ or $\emptyset$. Then, by the definition of $\mathcal{R}\otimes\Delta$, state $[1{\rightarrow}0]$ is $\mathcal{R}$-related to $\{[1{\rightarrow}2, 2{\rightarrow}0]\}$ or $\emptyset$. In either case, $\mathcal{R}$ is not admissible; the first case violates the second conjunct about the free locations in the admissibility condition, and the second case violates the first conjunct about the non-emptiness.

6.7 Discussion

The work in this chapter is joint work with Hongseok Yang and was presented at the conference APLAS [53]. The material presented in the following chapter is also a part of this joint work.

Work on power simulations assumes a concrete RAM memory model, rather the general memory model used in work on forward simulations. One reason for that is that work on power simulations was inspired by the problems that arise exclusively from the features of the low-level programming languages. For example, the notion of a finite local action would be very tricky to formulate. Namely, the finite access property heavily relies on the assumptions of the RAM model. We suspect that it would be possible to carry the results in this, low-level model to the general memory model, but we leave this generalization as a possible future direction.

[7] 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).

7

State-based representation of the Power Simulation Method

We have introduced so far two methods for proving data refinement that are sound even in presence of pointers: forward simulation and power simulation. Even though power simulation is a more general method which enables us to prove data refinement for a broader range of data structures, the forward simulation method is more intuitive and easier to use. As the methods are based on similar principles, the question of whether there is some connection between them poses itself. Indeed, in certain circumstances power simulation can be reduced to the simple forward simulation, and in some other, to forward simulation with "backtracking".

The goal of this chapter is to introduce an important connection between forward simulation and power simulation methods and to give a state based representation of the power simulation method. In Chapter 1 we discussed the pitfalls of the forward simulation method and illustrated them with inability of the forward simulation method to deal with the equivalence of the doubly-linked list and XOR-linked list representations of doubly-ended queues. Here, we prove this equivalence using the power simulation method and its state-based representation.

7.1 State-based Representation of the Power Simulation Method

The soundness result for power simulation method 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 consider two special cases of power simulation. In the first case, a coupling power relation is built from the growing relation between the states. In the second case a coupling power relation arises from a standard coupling relation on *states* and, we 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).

7.1.1 Growing Relation between States

In the first case the coupling power relation is built from a *growing* relation between states. An important fact of this construction is that for such coupling relations, the power simulation collapses to the usual forward simulation on states. Then, the soundness of the power simulation method also indirectly implies the soundness of the usual forward simulation for the growing refinement relation.

Recall that relation $S \subseteq \mathsf{State} \times \mathsf{State}$ between states is *growing* if and only if

$$h[S]h' \implies \mathsf{dom}(h) \subseteq \mathsf{dom}(h').$$

An interesting feature of growing relations is that they give rise to admissible power relations. Let S be a growing relation. We define a power relation $\mathsf{lft}(S)$ as follows:

$$H[\mathsf{lft}(S)]h \iff \exists h_0 \in \mathsf{State}.\, H = \{h_0\} \,\wedge\, h_0[S]h.$$

The lifted relation $\mathsf{lft}(S)$ is admissible. Intuitively, the unique state h_0 in H uses so little memory, that it allows all the possible allocations from the other state h. We formally prove this property below:

Lemma 25. *The power relation* $\mathsf{lft}(S)$ *is admissible.*

Proof. In order to prove that $\mathsf{lft}(S)$ is admissible, we need to prove that $\mathsf{lft}(S)$ satisfies the following two conditions: for all $H \in \wp(\mathsf{State})$ and all $h \in \mathsf{State}$ such that $H[\mathsf{lft}(S)]h$,

- $H \neq \emptyset$; and
- for all $L \subseteq_{\mathsf{fin}} \mathsf{Loc}$, there exists a subset H_1 of H such that

$$H_1 \neq \emptyset \;\wedge\; H_1[\mathsf{lft}(S)]h \;\wedge\; \forall h_1 \in H_1.\, h_1 \subseteq_L h.$$

Let set H and heap h be related by $\mathsf{lft}(S)$. Then, by the definition of $\mathsf{lft}(S)$, there exists $h_0 \in \mathsf{State}$ such that $H = \{h_0\}$ and $h_0[S]h$, and hence $H \neq \emptyset$, which makes the first condition of admissibility true. To prove the second condition of admissibility, note that since $h_0[S]h$ and S is growing, we have $dom(h_0) \subseteq dom(h)$. Therefore, for every set $L \subseteq_{\mathsf{fin}} \mathsf{Loc}$, we have $dom(h_0) \cap L \subseteq dom(h) \cap L$. Hence, the set H itself is the required non-empty subset H_1 in the second condition. $\square$

The main result in this case is that for each growing relation S, the power simulation for $\mathsf{lft}(S)$ can equivalently be formulated as a usual forward simulation for S. Let (p, η) and (q, ε) be two modules, and let S be a growing relation such that $S \subseteq p \times q$. Let $S * \Delta$ be a relation between states such that

$$h[S*\Delta]h' \iff \exists h_0, h_1, h_0'.\, \Big(h_0 * h_1 = h \;\wedge\; h_0' * h_1 = h' \;\wedge\; h_0[S]h_0'\Big)$$

We give the precise statement of the result as follows:

Proposition 1. *For all finite local actions r and r', the action r' simulates r by $\mathsf{lft}(S)$ if and only if*

1. *for all states h, h', if*

   $$h[S*\Delta]h' \;\wedge\; (h'[r']\mathsf{wrong} \;\vee\; h'[r']\mathsf{av})$$

   then $(h[r]\mathsf{wrong} \vee h[r]\mathsf{av})$; and

2. *for all states h, h', m', if*

   $$h[S*\Delta]h' \;\wedge\; \neg h[r]\mathsf{wrong} \;\wedge\; \neg h[r]\mathsf{av} \;\wedge\; h'[r']m'$$

   then there exists a state m such that

   $$h[r]m \;\wedge\; m[S*\Delta]m'.$$

For simplicity we will refer to the first condition as FS_1, and to the second as FS_2.

Proof. We prove that for the coupling relation $\mathsf{lft}(S)$, the conditions PS_1 and PS_2 of the corresponding power-simulation, are equivalent to FS_1 and FS_2, respectively.

Before proving the equivalence, we note one important property of power relation $\mathsf{lft}(S) \otimes \Delta$: for all state sets H and all states h',

$$H[\mathsf{lft}(S) \otimes \Delta]h' \iff \exists h.\, H = \{h\} \wedge h[S * \Delta]h'.$$

This property gives a simple characterization of $\mathsf{lft}(S) \otimes \Delta$: it is the "lifting" of $S * \Delta$. We will use this characterization extensively in the proof of this proposition. The proof of this property itself is straightforward, mainly done by unrolling the definitions and using the assumption that S is growing. We give the detailed proofs below:

$H[\mathsf{lft}(S) \otimes \Delta]h'$

$\iff$ ($\because$ the definition of $S \otimes \Delta$)

$\exists H_s, h_0, h'_s.\ \big(H = H_s * \{h_0\} \wedge h' = h'_s * h_0 \wedge H_s[\mathsf{lft}(S)]h'_s\big)$

$\iff$ ($\because$ the definition of $\mathsf{lft}(S)$)

$\exists H_s, h_0, h'_s, h_s.\ \big(H = H_s * \{h_0\} \wedge h' = h'_s * h_0 \wedge H_s = \{h_s\} \wedge h_s[S]h'_s\big)$

$\iff$ ($\because$ the rules from classical logic)

$\exists h_0, h'_s, h_s.\ \big(H = \{h_s\} * \{h_0\} \wedge h' = h'_s * h_0 \wedge h_s[S]h'_s\big)$

$\iff$ ($\because$ S is growing, so $h_s[S]h'_s \wedge h'_s \# h \Rightarrow h_s \# h$)

$\exists h_0, h'_s, h_s.\ \big(h_s \# h_0 \wedge H = \{h_s\} * \{h_0\} \wedge h' = h'_s * h_0 \wedge h_s[S]h'_s\big)$

$\iff$ ($\because$ the definition of $H_0 * H_1$)

$\exists h_0, h'_s, h_s, h.\ \big(h = h_s * h_0 \wedge H = \{h\} \wedge h' = h'_s * h_0 \wedge h_s[S]h'_s\big)$

$\iff$ ($\because$ the definition of $S * \Delta$)

$\exists h.\ \big(H = \{h\} \wedge h[S * \Delta]h'\big)$

Using the above property of $\mathsf{lft}(S) \otimes \Delta$, we first prove the equivalence between PS_1 and FS_1:

$\forall H, h'.\ \big(H[\mathsf{lft}(S) \otimes \Delta]h' \wedge (h'[r']\mathsf{wrong} \vee h'[r']\mathsf{av})\big)$
$\qquad \Longrightarrow \big(H[\mathsf{lft}(r)]\mathsf{wrong} \vee H[\mathsf{lft}(r)]\mathsf{av}\big)$

$\iff$ ($\because$ the shown property of $\mathsf{lft}(S) \otimes \Delta$)

$$\forall H, h'.\ \big(\exists h.\ H = \{h\}\ \wedge\ h[S * \Delta]h'\ \wedge\ (h'[r']\mathsf{wrong}\ \vee\ h'[r']\mathsf{av})\big)$$
$$\implies \big(H[\mathsf{lft}(r)]\mathsf{wrong}\ \vee\ H[\mathsf{lft}(r)]\mathsf{av}\big)$$

$\iff$ ($\because$ the rules from classical logic)

$$\forall H, h, h'.\ \big(H = \{h\}\ \wedge\ h[S * \Delta]h'\ \wedge\ (h'[r']\mathsf{wrong}\ \vee\ h'[r']\mathsf{av})\big)$$
$$\implies \big(H[\mathsf{lft}(r)]\mathsf{wrong}\ \vee\ H[\mathsf{lft}(r)]\mathsf{av}\big)$$

$\iff$ ($\because$ the rules from classical logic)

$$\forall h, h'.\ \big(h[S * \Delta]h'\ \wedge\ (h'[r']\mathsf{wrong}\ \vee\ h'[r']\mathsf{av})\big)$$
$$\implies \big(\{h\}[\mathsf{lft}(r)]\mathsf{wrong}\ \vee\ \{h\}[\mathsf{lft}(r)]\mathsf{av}\big)$$

$\iff$ ($\because$ the definition of $\mathsf{lft}(r)$)

$$\forall h, h'.\ \big(h[S * \Delta]h'\ \wedge\ (h'[r']\mathsf{wrong}\ \vee\ h'[r']\mathsf{av})\big) \implies \big(h[r]\mathsf{wrong}\ \vee\ h[r]\mathsf{av}\big)$$

Next we prove the equivalence between PS_2 and FS_2:

$$\forall H, h', m'.\ \big(H[\mathsf{lft}(S) \otimes \Delta]h'\ \wedge\ \neg H[\mathsf{lft}(r)]\mathsf{wrong}\ \wedge\ \neg H[\mathsf{lft}(r)]\mathsf{av}\ \wedge\ h'[r']m'\big)$$
$$\implies \exists M.\ \big(M[\mathsf{lft}(S) \otimes \Delta]m'\ \wedge\ H[\mathsf{lft}(r)]M\big)$$

$\iff$ ($\because$ the shown property of $\mathsf{lft}(S) \otimes \Delta$)

$$\forall H, h', m'.\ \big(\exists h.\ H = \{h\}\ \wedge\ h[S * \Delta]h' \wedge \neg H[\mathsf{lft}(r)]\mathsf{wrong} \wedge \neg H[\mathsf{lft}(r)]\mathsf{av} \wedge h'[r']m'\big)$$
$$\implies \exists M, m.\ \big(M = \{m\} \wedge m[S * \Delta]m' \wedge H[\mathsf{lft}(r)]M\big)$$

$\iff$ ($\because$ the rules from classical logic)

$$\forall H, h, h', m'.\ \big(H = \{h\}\ \wedge\ h[S * \Delta]h' \wedge \neg H[\mathsf{lft}(r)]\mathsf{wrong} \wedge \neg H[\mathsf{lft}(r)]\mathsf{av} \wedge h'[r']m'\big)$$
$$\implies \exists M, m.\ \big(M = \{m\} \wedge m[S * \Delta]m' \wedge H[\mathsf{lft}(r)]M\big)$$

$\iff$ ($\because$ the rules from classical logic)

$$\forall h, h', m'.\ \big(h[S * \Delta]h' \wedge \neg\{h\}[\mathsf{lft}(r)]\mathsf{wrong} \wedge \neg\{h\}[\mathsf{lft}(r)]\mathsf{av} \wedge h'[r']m'\big)$$
$$\implies \exists m.\ \big(m[S * \Delta]m' \wedge \{h\}[\mathsf{lft}(r)]\{m\}\big)$$

$\iff$ ($\because$ the definition of $\mathsf{lft}(r)$)

$$\forall h, h', m'.\ \big(h[S * \Delta]h' \wedge \neg h[r]\mathsf{wrong} \wedge \neg h[r]\mathsf{av} \wedge h'[r']m'\big)$$
$$\implies \exists m.\ \big(m[S * \Delta]m' \wedge h[r]m\big)$$

$\square$

7.1.2 Power relation induced by per

The previous case is based on the assumption that the standard coupling relation $R \subseteq q \times p$ is growing. However, any 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: a partial equivalence relation E on p, and an operator rs, which imposes certain restrictions on relations defined on states.

Partial equivalence relation E denotes which "abstract" states can be considered the same and satisfies the property

$$R; E = R.$$

The other data, rs, sets a restriction on a relation E, using pre-given finite location set L and state h. More formally, for a given set L and state h, we define $\mathsf{rs}(E, L, h)$ to be a relation defined as

$$h'[\mathsf{rs}(E, L, h)]h'_0 \overset{def}{\Leftrightarrow} h'[E]h'_0 \wedge h'_0 \sqsubseteq_L h.$$

Note that $\mathsf{rs}(E, L, h)$ then satisfies

$$(\forall L'.\, \mathsf{rs}(E, L \cup L', h) \subseteq \mathsf{rs}(E, L, h))$$

since $\sqsubseteq_{-}$ is anti-monotone with respect to its subscript.

Definition 25. *The pair (R, E) is a coupling pair if and only if for all R-related states h and h' and all finite sets L of locations, there exists a state h_0 such that $h_0[E]h$ and $h_0 \sqsubseteq_L h'$.*

From a coupling pair (R, E) and restriction rs, we define a power relation as follows:

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

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

Lemma 26. *Power relation $\mathsf{pw}(R, E, \mathsf{rs})$ 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

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

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 $L \subseteq_{\mathsf{fin}} \mathsf{Loc}$ and $h \in \mathsf{State}$ 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 fact that the (R,E) is a coupling pair. 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, $h \sqsubseteq_{L\cup L_1} h_1$. Since $\sqsubseteq_-$ is anti-monotone with respect to its subscript, it follows that $h \sqsubseteq_{L_1} h_1$. □

Before we present the main result of this section, we need to prove the following important property of $\mathsf{pw}(R,E,\mathsf{rs})$.

Lemma 27. *A state set H and a state h' are related by $\mathsf{pw}(R,E,\mathsf{rs}) \otimes \Delta$ if and only if they are related by $\mathsf{pw}(R*\Delta,E*\Delta,\mathsf{rs})$.*

Proof. Let H be a state set and h' a state. We first unroll the definition of $h'[\mathsf{pw}(R,E,\mathsf{rs}) \otimes \Delta]H$ and simplify the result using the rules from classical logic:

$$h'[\mathsf{pw}(R,E,\mathsf{rs}) \otimes \Delta]H$$

$$\iff$$

$$\exists H_R, h_0, h'_R.\ \Big(H = H_R * \{h_0\} \ \wedge\ h' = h'_R * h_0 \ \wedge\ h'_R[\mathsf{pw}(R,E,\mathsf{rs})]H_R\Big)$$

$$\iff$$

$$\exists H_R, h_0, h'_R, L, h_R.$$
$$\Big(H = H_R * \{h_0\} \ \wedge\ h' = h'_R * h_0 \ \wedge\ h_R[R]h'_R \ \wedge\ H_R = \{m_R \mid m_R[E]h_R \wedge m_R \sqsubseteq_L h'_R\}\Big)$$
$$\iff$$
$$\exists h_0, h'_R, L, h_R.\ \Big(H = \{m_R * h_0 \mid m_R \# h_0 \wedge m_R[E]h_R \wedge m_R \sqsubseteq_L h'_R\} \ \wedge\ h' = h'_R * h_0 \ \wedge\ h_R[R]h'_R\Big)$$

We designate this last formula with φ. Now, we need to show that φ is equivalent to $h'[\mathsf{pw}(R*\Delta, E*\Delta, \mathsf{rs})]H$, i.e.,

$$\exists h, L.\ h'[R*\Delta]h \ \wedge\ H = \{h_0 \mid h[E*\Delta]h_0 \ \wedge\ h_0 \sqsubseteq_L h'\}.$$

which we designate by ψ.

The proof about the equivalence uses one important fact about state sets that are constructed in a specific way:

FACT 1: for all heaps h_R, h_0, h'_R, if $h_R \# h_0 \ \wedge\ h'_R \# h_0$, then

$$\{m_R * h_0 \mid m_R \# h_0 \wedge m_R[E]h_R \wedge m_R \sqsubseteq_L h'_R\} = \{m \mid m[E*\Delta]h_R * h_0 \wedge m \sqsubseteq_L h'_R * h_0\}.$$

The proof of this fact involves unfolding the definition of $E*\Delta$ and exploiting the preciseness of E, as $E \subseteq p \times p$ and p is precise. We omit the details of the proof.

Now, we prove the equivalence between the formulas φ and ψ. To show the right implication, let h_0, h'_R, h_R be states and L a finite set of locations such that

$$H = \{m_R * h_0 \mid m_R \# h_0 \wedge m_R[E]h_R \wedge m_R \sqsubseteq_L h'_R\} \ \wedge\ h' = h'_R * h_0 \ \wedge\ h_R[R]h'_R.$$

Since (R, E) is a coupling pair and $h_R[R]h'_R$, there exists n_R such that

$$n_R[E]h_R \ \wedge\ n_R \sqsubseteq_{\mathsf{dom}(h_0)} h'_R.$$

Since $R;E = R$, state n_R is related to h'_R by R, and since h'_R is disjoint from h_0, state n_R is also disjoint from h_0. Thus, we have

$$n_R * h_0[R*\Delta]h'_R * h_0. \tag{7.1}$$

Moreover, since E is per and $n_R[E]h_R$,

$$H = \{m_R * h_0 \mid m_R \# h_0 \wedge m_R[E]n_R \wedge m_R \sqsubseteq_L h'_R\}.$$

By Fact 1, this implies that

$$H = \{m \mid m[E * \Delta]n_R * h_0 \wedge m \sqsubseteq_L h'_R * h_0\}. \tag{7.2}$$

Formula ψ follows from 7.1 and 7.2.

For the other implication, consider a state h and a finite set L of locations such that

$$H = \{m \mid m[E * \Delta]h \ \wedge\ h \sqsubseteq_L h'\} \ \wedge\ h[R * \Delta]h'.$$

By the definition of $R * \Delta$, there exist splitting $h_R * h_0 = h$ of h and $h'_R * h'_0 = h'$ of h' such that

$$h_R[R]h'_R \ \wedge\ h_0[\Delta]h'_0.$$

By the definition of Δ, $h_0 = h'_0$. We now apply Fact 1 to h_R, h_0, h'_R, and get the following equality:

$$H = \{m_R * h_0 \mid m_R \# h_R \ \wedge\ m_R[E]h_R \ \wedge\ h_R \sqsubseteq_L h'_R\}.$$

This equality together with $h' = h'_R * h_0$ and $h_R[R]h'_R$ implies the formula φ. $\square$

Finally, we prove the main result of this section, that is, that the power relation has a state-based characterization.

Proposition 2. *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 $h'[R{*}\Delta]h \wedge (\forall h_0.\ h[\mathsf{rs}(E, L_0, h')]h_0 \Rightarrow \neg h_0[r]\mathsf{av} \wedge \neg h_0[r]\mathsf{wrong})$, then*

1. $\neg h'[r']\mathsf{wrong} \wedge \neg h'[r']\mathsf{av}$, *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 states that every such output state m' from the given concrete input h' should be $R{*}\Delta$-related to some "backtrack-able 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 "backtrack-able abstract computation," instead of a normal abstract computation.

Proof. 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 $\alpha_i = \forall \ldots \varphi_i {\Rightarrow} \psi_i$ and $\beta_i = \forall \ldots \varphi_i {\Rightarrow} \psi_i'$ where the universal quantifications are precisely the ones in the original formulas 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:

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

$$\begin{aligned}&\forall h', h, L_0.\\ &\big(h'[R{*}\Delta]h \wedge (\forall h_0.\, h[\mathsf{rs}(E{*}\Delta, L_0, h')]h_0 \Rightarrow \neg h_0[r]\mathsf{av} \wedge \neg h_0[r]\mathsf{wrong})\big)\\ &\Rightarrow (\neg h'[r']\mathsf{av} \wedge \neg h'[r']\mathsf{wrong})\end{aligned}$$

The second fact can be proved as follows:

$$\begin{aligned}&\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{wrong} \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)\\ \iff\ & (\because \mathsf{pw}(R,E,\mathsf{rs}){\otimes}\Delta = \mathsf{pw}(R{*}\Delta, E{*}\Delta, \mathsf{rs}))\\ &\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{wrong} \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)\\ \iff\ & (\because \text{the definition of } \mathsf{pw}(R{*}\Delta, E{*}\Delta, \mathsf{rs}))\\ &\forall h', H, m'.\\ &\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{wrong} \wedge h'[r']m'\end{array}\right)\\ &\ \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)\\ \iff\ &\\ &\forall h', H, m', h, L_0.\\ &\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{wrong} \wedge h'[r']m'\end{array}\right)\\ &\ \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)\\ \iff\ &\\ &\forall h', m', h, L_0.\\ &\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)\\ &\ \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)\\ \iff\ & (\because \text{the definition of } \mathsf{lft}(r))\\ &\forall h', m', h, L_0.\\ &\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{wrong}\big)\Big)\\ &\ \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)\end{aligned}$$

$$\Longleftrightarrow$$

$$\forall h', h, L_0.$$

$$\Big(h'[R{*}\Delta]h \wedge \big(\forall h_0.\ h[\mathsf{rs}(E{*}\Delta, L_0, h')]h_0 \Rightarrow \neg h_0[r]\mathsf{av} \wedge \neg h_0[r]\mathsf{wrong}\big)\Big)$$

$$\Rightarrow$$

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

$\square$

7.2 Example

In this section, we show that the *XOR-linked* list representation of the doubly ended queues is equivalent to its *doubly-linked* list representation. First we give the definition of equivalence between two modules.

Definition 26 (Equivalence)**.** *Two modules (p, η) and (q, ε) are equivalent if and only if they data refine each other.*

We formally define predicates xlist and dlist. Let α be a sequence. Predicates xlist and dlist are defined inductively on the sequence α. Empty sequence is denoted by ε, and $\cdot$ conses an element to the front of the sequence. The definitions are taken from [71]

$$\begin{array}{rcl}\mathsf{dlist}\ \varepsilon(i,i',j,j') & \equiv & \mathsf{emp} \wedge i = j \wedge i' = j' \\ \mathsf{dlist}\ a\cdot\alpha(i,i',k,k') & \equiv & \exists j.i \mapsto a, j, i' * \mathsf{dlist}\ \alpha(j,i,k,k')\end{array}$$

$$\begin{array}{rcl}\mathsf{xlist}\ \varepsilon(i,i',j,j') & \equiv & \mathsf{emp} \wedge i = j \wedge i' = j' \\ \mathsf{xlist}\ a\cdot\alpha(i,i',k,k') & \equiv & \exists j.i \mapsto a, (j \oplus i') * \mathsf{xlist}\ \alpha(j,i,k,k')\end{array}$$

We define the dlist module (p,η) with the following.

$$\begin{aligned}
h \in p \quad &\iff \quad h \in \exists\alpha.\ \mathsf{dlist}\ \alpha(i,nil,nil,m)\\
h[\eta(\mathsf{insert}(a))]v \quad &\iff \quad (\exists l,j,\alpha,\beta.\ a \notin \alpha\cdot\beta \ \wedge\ \mathsf{dlist}\ \alpha(i,nil,l,j) * \mathsf{dlist}\ \beta(l,j,nil,m) *\\
&\qquad \mathsf{true} \ \wedge\ \exists k,k+1,k+2 \notin \mathsf{dom}(h).\ v = h[j+1 \mapsto k, l+2 \mapsto k]\cdot\\
&\qquad k \mapsto a,l,j) \ \vee\ (a \in \alpha\cdot\beta \ \wedge\ v = \mathsf{wrong})\\
h[\eta(\mathsf{delete}(a))]v \quad &\iff \quad (\exists l,j,k,\alpha,b,\beta.\ b = a \ \wedge\ h \in \mathsf{dlist}\ \alpha(i,nil,k,j) * k \mapsto b,l,j *\\
&\qquad \mathsf{dlist}\ \beta(l,k,nil,m) * \mathsf{true} \ \wedge\ \exists v'.\ h = v'\cdot k \mapsto b,l,j \ \wedge\\
&\qquad v = v'[j+1 \mapsto l, l+2 \mapsto j]) \ \vee\ (a \notin \alpha\cdot b\cdot\beta \ \wedge\ v = \mathsf{wrong})
\end{aligned}$$

Similarly, we define the xlist module – (q,ε).

$$\begin{aligned}
h \in q \quad &\iff \quad h \in \exists\alpha.\ \mathsf{xlist}\ \alpha(i,nil,nil,m)\\
h[\varepsilon(\mathsf{insert}(a))]v \quad &\iff \quad (\exists l,j,\alpha,\beta.\ a \notin \alpha\cdot\beta \ \wedge\ \mathsf{xlist}\ \alpha(i,nil,l,j) * \mathsf{xlist}\ \beta(l,j,nil,m) *\\
&\qquad \mathsf{true} \ \wedge\ \exists k,k+1 \notin \mathsf{dom}(h).\ v = h[j+1 \mapsto k \oplus l \oplus h(j+1),\\
&\qquad l+1 \mapsto k \oplus j \oplus h(l+1)]\cdot k \mapsto a, l \oplus j) \ \vee\ (a \in \alpha\cdot\beta \ \wedge\\
&\qquad v = \mathsf{wrong})\\
h[\varepsilon(\mathsf{delete}(a))]v \quad &\iff \quad (\exists l,j,k,\alpha,b,\beta.\ b = a \ \wedge\ h \in \mathsf{xlist}\ \alpha(i,nil,k,j) * k \mapsto b, l \oplus j *\\
&\qquad \mathsf{xlist}\ \beta(l,k,nil,m) * \mathsf{true} \ \wedge\ \exists v'.\ h = v' * k \mapsto b, l \oplus j \ \wedge\\
&\qquad v = v'[j+1 \mapsto k \oplus l \oplus v'(j+1), l+1 \mapsto k \oplus j \oplus v'(l+1)])\\
&\qquad \vee\ (a \notin \alpha\cdot b\cdot\beta \ \wedge\ v = \mathsf{wrong})
\end{aligned}$$

We can assume for i,m – the delimiters of the lists, that they are some fixed locations and for simplicity that they are always in the domain of the heap. We leave out initialization and finalization operations (we can always impose that they are empty relations, and that lists are always constructed by the client using the insert() operation). All of these operations are finite local actions.

Lemma 28. *All the operations of the both modules (p,η) and (q,ε) are finite local actions.*

The proof is based on routine checking that all four properties: safety monotonicity, frame property, finite access property and content independence hold.

The proof of equivalence of the XOR-linked and doubly-linked modules is somewhat complex and therefore for the sake of better organization and understanding we divide it into two parts. We separately prove the two directions of data refinement between the modules, from which it will result their equivalence.

7.2.1 Power-simulation of doubly-linked list module by the XOR-list module

We now prove that the XOR-linked list module data refines the doubly-linked list one. To do that we need to provide the refinement (power) relation and prove for both $\mathsf{insert}()$ and $\mathsf{delete}()$ operations of the XOR-linked list module that they power simulate the corresponding operations of the doubly-linked list module.

We define the refinement relation R to be

$$\begin{aligned} h[R]H \iff & \exists L, \alpha.\ h \in \mathsf{xlist}\ \alpha(i, nil, nil, m)\ \wedge \\ & H = \{h' \mid h' \in \mathsf{dlist}\ \alpha(i, nil, nil, m)\ \wedge\ h' \sqsubseteq_L h\} \end{aligned}$$

It is easy to check that $R : q \leftrightarrow \mathcal{P}(p)$. We need to make sure that R is admissible.

Lemma 29. *R, defined as above, is admissible.*

Proof. let h and H be such that $h[R]H$. Then there exist a finite set of locations L and a sequence α such that $h \in \mathsf{xlist}\ \alpha(i, nil, nil, m)$ and $H = \{h' \mid h' \in \mathsf{dlist}\ \alpha(i, nil, nil, m)\ \wedge\ h' \sqsubseteq_L h\}$. We first prove that $H \neq \emptyset$, by constructing a heap h' such that $h' \in H$. Let n be a length of sequence α, i.e. there exist $a_1, \ldots, a_n$ such that $\alpha = a_1 \cdot \ldots \cdot a_n$. Let $l_i + j \in \mathsf{Loc}$ be any locations such that $l_i + j \neq l_{i'} + j'$ and $l_i + j \notin \mathsf{dom}(h) \cup L$, for $i, i' = 1, \ldots, n$, $i \neq i'$ $j, j' = 0, \ldots, 2$. Such locations exist, since Loc is infinite and $\mathsf{dom}(h)$ and L are finite. We construct h' by setting its domain to $\mathsf{dom}(h') = \{l_i + j \mid i = 1, \ldots, n,\ j = 0, 1, 2\}$, setting $h'(i) = l_1$, $h'(m) = l_n$ and letting $h' : \mathsf{Loc} \rightharpoonup \mathsf{Loc}$ to be the map defined by:

$$h'(l) = \begin{cases} a_i, & l = l_i,\ i = 1, \ldots, n \\ l_i + 1, & l = l_i + 1,\ i = 1, \ldots, n-1 \\ nil, & l = l_n + 1 \\ l_i - 1, & l = l_i + 2,\ i = 2, \ldots, n \\ nil, & l = l_1 + 2 \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

Then, $h' \in \mathsf{dlist}\ \alpha(i, nil, nil, m)$. Since $\mathsf{dom}(h') \cap (\mathsf{dom}(h) \cup L) = \emptyset$, i.e. $\mathsf{dom}(h') \cap L = \emptyset$, it follows that $\mathsf{dom}(h') \cap L \subseteq \mathsf{dom}(h) \cap L$, i.e. $h' \sqsubseteq_L h$. Therefore, $h' \in H$ and $H \neq \emptyset$.

We now check the other condition of admissibility. Let L_1 be any finite set of locations. Then,

$$H_1 = \{h' \mid h' \in \mathsf{dlist}\ \alpha(i, nil, nil, m)\ \wedge\ h' \sqsubseteq_{L \cup L_1} h\}$$

is a subset of H, as $\sqsubseteq_{-}$ is anti-monotone. H_1 is not empty, as we can always construct a heap h_1 that belongs to H_1 in a similar manner as above. Since L_1 is an arbitrary finite set of locations, this is true for any such set. Heap h is power-related to H_1 by the definition of $\mathcal{R}$. We have proved

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

i.e., relation R is admissible. □

We now prove that the $\mathsf{insert}()$ operation of the XOR-linked list module power simulates the $\mathsf{insert}()$ operations of the doubly-linked list module, with respect to the power relation $R \otimes \Delta$.

Lemma 30. *The following holds:*

$$\varepsilon(\mathsf{insert}(a))[\mathsf{psim}(R \otimes \Delta, R \otimes \Delta)]\eta(\mathsf{insert}(a)).$$

Proof. Let h and H be such that $h[R \otimes \Delta]H$. Let $\neg H[\mathsf{lft}(\eta(\mathsf{insert}(a)))]\mathsf{wrong}$. We first show that $\neg h[\varepsilon(\mathsf{insert}(a))]\mathsf{wrong}$. From $h[R \otimes \Delta]H$, it follows that there exist H_1, h_1 and h_0 such that

$$h_1[R]H_1 \text{ and } H = H_1 * \{h_0\} \text{ and } h = h_1 \cdot h_0 \text{ and } h_1 \# h_0.$$

Let α and L be such that $h_1 \in \mathsf{xlist}\ \alpha(i, nil, nil, m)$ and

$$H_2 = \{h_2 \mid h_2 \in \mathsf{dlist}\ \alpha(i, nil, nil, m) \ \wedge \ h_2 \sqsubseteq_{\mathsf{dom}(h_0)} h_1\},$$
$$H_1 = \{h_2 \mid h_2 \in \mathsf{dlist}\ \alpha(i, nil, nil, m) \ \wedge \ h_2 \sqsubseteq_L h_1\}.$$

Then, because $h_1 \# h_0$ and $h_2 \sqsubseteq_{\mathsf{dom}(h_0)} h_1$, H can be written as

$$H = \{h_2 * h_0 \mid h_2 \# h_0 \ \wedge \ h_2 \in \mathsf{dlist}\ \alpha(i, nil, nil, m) \ \wedge \ h_2 \sqsubseteq_L h_1\}.$$

Since the lifted execution of $\eta(\mathsf{insert}(a))$ does not go wrong from set H, this means that for all h_3 in H, $h_3 \in \mathsf{dlist}\ \alpha(i, nil, nil, m) * \mathsf{true}$ and $a \notin \alpha$. On the other hand $h \in \mathsf{xlist}\ \alpha(i, nil, nil, m) * \mathsf{true}$ and since $a \notin \alpha$ the execution of $\varepsilon(\mathsf{insert}(a))$ does not go wrong on h either.

Now, we prove that the second condition of the power simulation method holds. Let g be an output state of $\varepsilon(\mathsf{insert}(a))$ from h. Then, since h satisfies $\mathsf{xlist}\ \alpha(i, nil, nil, m) * \mathsf{true}$ then there exist β and γ (possibly empty) such that $\alpha = \beta \cdot \gamma$, and l, j such that

$h \in \mathsf{xlist}\ \beta(i,nil,l,j) * \mathsf{xlist}\ \gamma(l,j,nil,m) * \mathsf{true}$. State g is the output state, and so there exist $k,k+1 \notin \mathsf{dom}(h)$ such that

$$g = h[j+1 \mapsto k \oplus l \oplus h(j+1), l+1 \mapsto k \oplus j \oplus h(l+1)] \cdot k \mapsto a, l \oplus j.$$

We have that $h = h_1 \cdot h_0$ (recall that $h_1 \in \mathsf{xlist}\ \alpha(i,nil,nil,m)$) and $\neg h_1[\varepsilon(\mathsf{insert}(a))]\mathsf{wrong}$ and since $h[\varepsilon(\mathsf{insert}(a))]g$, then by the frame property there exists a state g_1, such that $h_1[\varepsilon(\mathsf{insert}(a)]g_1$, and $g_1\#h_0$ and $g = g_1 \cdot h_0$. Then $g_1 \in \mathsf{xlist}\ \beta \cdot a \cdot \gamma(i,nil,nil,m)$, i.e.

$$g_1 = h_1[j+1 \mapsto k \oplus l \oplus h(j+1), l+1 \mapsto k \oplus j \oplus h(l+1)] \cdot k \mapsto a, l \oplus j.$$

Now, we construct the set G such that $H[\mathsf{lft}(\eta(\mathsf{insert}(a)))]G$ and $g[R * \Delta]G$. We know that

$$\begin{aligned} H_1 &= \{h_2 \mid \ h_2\#h_0 \ \wedge\ h_2 \in \mathsf{dlist}\ \alpha(i,nil,nil,m) \ \wedge\ h_2 \sqsubseteq_L h_1\} \\ &= \{h_2 \mid \ \exists n.\ h_2\#h_0 \ \wedge\ h_2 \in \mathsf{dlist}\ \beta(i,nil,n,m) * \mathsf{dlist}\ \gamma(i,n,nil,m) \ \wedge \\ &\qquad\qquad h_2 \sqsubseteq_L h_1 \ \wedge\ \alpha = \beta \cdot \gamma\}. \end{aligned}$$

Now, let

$$\begin{aligned} G_1 = \{g_2 \cdot k \mapsto a,l,j \mid \ & g_2 = h_2[j+1 \mapsto k, l+2 \mapsto k] \ \wedge\ h_2 \in H_1 \ \wedge\ g_2\#h_0 \ \wedge \\ & k,k+1 \notin \mathsf{dom}(h_2) \cup \mathsf{dom}(h) \cup L\} \end{aligned}$$

Then, for all $g_2 \in G$, since $h_2 \sqsubseteq_L h_1$ and $k,k+1 \notin \mathsf{dom}(h)$ and $h_1\#h_0$, we have that $G_1 * \{h_0\}$ is defined. Set $G = G_1 * \{h_0\}$ is the one we are looking for. By construction, G is a possible output of $\mathsf{lft}(\eta(\mathsf{insert}(a)))$ from H:

$$\begin{aligned} G &= \left\{ g_2 \cdot h_0 \mid \begin{array}{l} g_2 = h_2[j+1 \mapsto k, l+2 \mapsto k] \cdot k \mapsto a,l,j \ \wedge\ g_2\#h_0 \ \wedge \\ h_2 \in H_1 \ \wedge\ k,k+1 \notin \mathsf{dom}(h) \cup L \end{array} \right\} \\ &= \left\{ g_2 \cdot h_0 \mid \begin{array}{l} g_2 = h_2[j+1 \mapsto k, l+2 \mapsto k] \cdot k \mapsto a,l,j \ \wedge\ h_2 \sqsubseteq_L h_1 \ \wedge\ g_2\#h_0 \ \wedge \\ h_2 \in \mathsf{dlist}\ \beta \cdot \gamma(i,nil,nil,m) \ \wedge\ k,k+1 \notin \mathsf{dom}(h) \cup L \end{array} \right\} \\ &= \left\{ g \mid \begin{array}{l} g = h[j+1 \mapsto k, l+2 \mapsto k] \cdot k \mapsto a,l,j \ \wedge \\ h \in H \ \wedge\ k,k+1 \notin \mathsf{dom}(h) \cup L \end{array} \right\}. \end{aligned}$$

It is easy to see that for all states g in G there exists a state h in H, such that $h[\eta(\mathsf{insert}(a))]g$, i.e. $H[\mathsf{lft}(\eta(\mathsf{insert}(a))]G$.

Finally, we prove $g[R \otimes \Delta]G$. We already know $g = g_1 * h_0$, and $g_1 \in \mathsf{xlist}\ \beta \cdot a \cdot \gamma(i,nil,nil,m)$

and $G = G_1 * \{h_0\}$ and

$$\begin{aligned} G_1 &= \left\{ g_2 \cdot k \mapsto a,l,j \;\middle|\; \begin{array}{l} g_2 = h_2[j+1 \mapsto k, l+2 \mapsto k] \;\wedge\; k,k+1 \notin \mathsf{dom}(h) \cup L \\ \wedge\; h_2 \in \mathsf{dlist}\ \beta \cdot \gamma(i,nil,nil,m) \;\wedge\; h_2 \sqsubseteq_L h_1 \end{array} \right\} \\ &= \{g_3 \mid g_3 \in \mathsf{dlist}\ \beta \cdot a \cdot \gamma(i,nil,nil,m) \;\wedge\; g_3 \sqsubseteq_L h_1 \;\wedge\; g_3 \# h_0\} \end{aligned}$$

Hence, there exist L and α' such that $g_1 \in \mathsf{xlist}\ \alpha'(i,nil,nil,m)$ and

$$G_1 = \{g_2 \mid g_2 \in \mathsf{dlist}\ \alpha'(i,nil,nil,m) \;\wedge\; g_2 \sqsubseteq_L g_1 \;\wedge\; g_2 \# h_0\}.$$

Therefore, $g_1[R]G_1$, and so $g[R \otimes \Delta]G$. □

We now prove that the $\mathsf{delete}()$ operation of the XOR-linked list module power simulates the $\mathsf{delete}()$ operation of the doubly-linked list module, with respect to the power relation $R \otimes \Delta$.

Lemma 31. *The following holds:*

$$\varepsilon(\mathsf{delete}(a))[psim(R \otimes \Delta, R \otimes \Delta)\eta(\mathsf{delete}(a)).$$

Proof. Let h and H be such that $h[R \otimes \Delta]H$. From this assumption we have that there exist h_0, h_1 and H_1, α and L such that

$$h_1[R]H_1 \text{ and } h = h_1 \cdot h_0 \text{ and } H = H_1 * \{h_0\},$$

where $h_1 \in \mathsf{xlist}\ \alpha(i,nil,nil,m)$ and

$$H_1 = \{h_2 \mid h_2 \in \mathsf{dlist}\ \alpha(i,nil,nil,m) \;\wedge\; h_2 \sqsubseteq_L h_1 \;\wedge\; h_2 \# h_0\}.$$

i.e.

$$H = \{h_2 \cdot h_0 \mid h_2 \in \mathsf{dlist}\ \alpha(i,nil,nil,m) \;\wedge\; h_2 \sqsubseteq_L h_1 \;\wedge\; h_2 \# h_0\}.$$

We first show that the first condition of the power simulation holds. Suppose $\neg H$ $[\mathsf{lft}(\eta(\mathsf{delete}(a)))]$ wrong. Then, for all $h' \in H$, it is true that $\neg h'[\eta(\mathsf{delete}(a))]\mathsf{wrong}$ and hence, there exist l, j, k, β, γ, such that $\alpha = \beta \cdot a \cdot \gamma$, i.e. $h' = h_2 \cdot h_0$ and $h_2 \in H_1$. On the other hand, $h = h_1 \cdot h_0$, and $h_1[R]H_1$, so $h_1 \in \mathsf{xlist}\ \alpha(i,nil,nil,m)$, and since $a \in \alpha$, we have $\neg h[\varepsilon(\mathsf{delete}(a))]\mathsf{wrong}$.

Now, we prove that the second condition of the power simulation holds. Let g be an output state of $\varepsilon(\mathsf{delete}(a))$ when run in state h, i.e. $h[\varepsilon(\mathsf{delete}(a))]g$. Then, let l, j, k_1, β, γ and g' be such that $\alpha = \beta \cdot a \cdot \gamma$, i.e.

$$h \in \mathsf{xlist}\ \beta(i, nil, k_1, j) * k_1 \mapsto a, l \oplus j * \mathsf{xlist}\ \gamma(l, k_1, nil, m) * \mathsf{true}$$

and

$$g = g'[j+1 \mapsto k_1 \oplus l \oplus g'(j+1), l+1 \mapsto k_1 \oplus j \oplus g'(l+1)],$$

where $h = g' \cdot k_1 \mapsto a, l \oplus j$ (g is a heap obtained from h by taking away the element that points to a and in which all the pointers are set correctly to form a list again). From $h = h_1 \cdot h_0$ and $\neg h_1[\varepsilon(\mathsf{delete}(a))]\mathsf{wrong}$, it follows by the frame property that there exists a state g_1 such that $g = g_1 * h_0$ and $h_1[\varepsilon(\mathsf{delete}(a))]g_1$, i.e. $g_1 \in \mathsf{xlist}\ \beta \cdot \gamma(i, nil, nil, m)$. We now construct G. State set H can be represented as $H = H_1 * \{h_0\}$, where

$$\begin{aligned} H_1 = \{h_2 \mid\ & h_2 \in \mathsf{dlist}\ \beta(i, nil, k, j) * k \mapsto a, l, j * \mathsf{dlist}\ \gamma(l, k, nil, m) \wedge \\ & h_2 \# h_0 \ \wedge\ h_2 \sqsubseteq_L h_1\}. \end{aligned}$$

We define G_1 to be

$$G_1 = \{g_2 \mid \exists g_1'.\ g_2 = g_1'[j+1 \mapsto k, l+2 \mapsto k] \ \wedge\ h_2 = g_1' * k \mapsto a, l, j \ \wedge\ h_2 \in H_1 \ \wedge\ h_2 \# h_0\}.$$

Then, since for all $g_2 \in G_1$, there exists $h_2 \in H_1$, such that $\mathsf{dom}(g_2) \subseteq \mathsf{dom}(h_2)$ and since for all $h_2 \in H_1$ it is true that $h_2 \# h_0$, then $g_1 \# h_0$ also, i.e. set $G = G_1 * \{h_0\}$ is defined and non-empty (because by the admissibility condition H_1 must be non-empty). We claim that G is the set we are looking for.

By construction, G is a possible output state set from H:

$$G = \{g_2 \cdot h_0 \mid \exists g_1'.\ g_2 = g_1'[j+1 \mapsto k, l+2 \mapsto k] \ \wedge\ h_2 = g_1' \cdot k \mapsto a, l, j \ \wedge\ h_2 \cdot h_0 \in H \ \wedge\ h_2 \# h_0\}.$$

For all $g' \in G$, there exists $h' \in H$ such that $g'[\eta(\mathsf{delete}(a))]h'$. We know that $h_2 \# h_0$ and $h_2 \sqsubseteq_L h_1$. Let L_1 be equal to $L \setminus \{k_1, k_1+1\}$ (locations that were deleted from h by running $\varepsilon(\mathsf{delete}(a))$). Let h' be any state in H, then $h' \sqsubseteq_L h$, i.e. $\mathsf{dom}(h') \cap L \subseteq h \cap L$, and for any $l, l+1, l+2 \in \mathsf{dom}(h')$,we have that $\mathsf{dom}(h') \setminus \{l, l+1, l+2\} \cap L_1 \subseteq h \setminus \{k, k+1\} \cap L_1$, i.e., $g' \sqsubseteq_{L_1} g$. Since $g \in \mathsf{xlist}\ \beta \cdot \gamma(i, nil, nil, m) * true$, i.e., $g = g' * h_0$ and $g' \in \mathsf{xlist}\ \beta \cdot \gamma(i, nil, nil, m)$, and also for all $g_2 \in G$, $g_2 = g_1 * h_0$ and $g_1 \in \mathsf{dlist}\ \beta \cdot \gamma(i, nil, nil, m)$ and $g_2 \sqsubseteq_{L_1} g$, we have $g[R \otimes \Delta]G$. □

Finally, we can conclude by what we have proved so far and by the soundness result for power simulation method that the XOR-linked list module data refines the doubly-linked list module.

Theorem 11. *Module (p, η) data refines module (q, ε).*

Proof. The proof follows from lemma 30 and lemma 31 and the soundness theorem 9. □

7.2.2 Power simulation of XOR-linked list module by the doubly-linked list module

Finally we prove that doubly-linked list module also simulates the XOR-linked list one. It can be easily noticed that a XOR-linked list takes less space than a doubly-linked list to represent the same sequence. This indicates that it might be possible to find a growing refinement relation and instead of proving the power simulation between the modules, to prove their forward simulation.

We give the refinement relation

$$\begin{aligned} h_1[R]h_2 \quad \Longleftrightarrow \quad & \exists\alpha.\ h_1 \in \mathsf{xlist}\ \alpha(i, nil, nil, m) \ \wedge\ h_2 \in \mathsf{dlist}\ \alpha(i, nil, nil, m)\ \wedge \\ & \mathsf{dom}(h_1) \subseteq \mathsf{dom}(h_2) \end{aligned}$$

We now prove that the $\mathsf{insert}()$ operation of the doubly-linked list module power simulates the $\mathsf{insert}()$ operation of the XOR-linked list module, with respect to the power relation $R * \mathsf{Id}$.

Lemma 32. *The following holds:*

$$\eta(\mathsf{insert}(a))[\mathsf{fsim}(R * \mathsf{Id}, R * \mathsf{Id})]\varepsilon(\mathsf{insert}(a)).$$

Proof. Let $h_1[R * \mathsf{Id}]h_2$. Then there exist states h_1', h_2' and h_0 such that $h_1 = h_1' \cdot h_0$ and $h_2 = h_2' \cdot h_0$ and $h_1'[R]h_2'$. That means that $h_1' \in \mathsf{xlist}\ \alpha(i, nil, nil, m)$ and $h_2' \in \mathsf{dlist}\ \alpha(i, nil, nil, m)$.

We first prove the first condition of forward simulation, so let $\neg h_1[\varepsilon(\mathsf{insert}(a))]\mathsf{wrong}$. Then, a is not in the sequence α already, so neither $\eta(\mathsf{insert}(a))$ goes wrong on state h_2.

Now we prove that the second condition of the forward simulation holds. Let g_2 be such that $h_2[\eta(\mathsf{insert}(a)]g_2$ and let $\neg h_1[\varepsilon(\mathsf{insert}(a))]\mathsf{wrong}$. Then by the definition of the operation,

$$g_2 = h_2[j+1 \mapsto k, l+2 \mapsto k] \cdot k \mapsto a, l, j$$

for some $k, k+1, k+2 \notin \mathsf{dom}(h_2)$ and $l, j \in \mathsf{dom}(h_2)$. Then l and j are the delimiters of the two sublists of sequences β and γ such that $\alpha = \beta \cdot \gamma$. Let

$$g_1 = h_1[j_1 + 1 \mapsto n \oplus l_1 \oplus h(j+1), l_1 + 1 \mapsto n \oplus j_1 \oplus h(l_1+1)] \cdot n \mapsto a, l_1 \oplus j$$

where $\{n, n+1\} \subseteq \{k, k+1, k+2\}$ and $l_1, j_1 \in \mathsf{dom}(h_1)$, such that l_1 and j_1 are the delimiters of the sublists that represent sequences β and γ. Then we claim that g_1 is the state we are looking for. Firstly, $h_1[\eta(\mathsf{insert}(a))]g_1$ follows from the construction of g_1. Secondly, $g_1[R * \mathsf{Id}]g_2$ because, in both states g_1 and g_2 the same sequence $\beta \cdot a \cdot \gamma$ is represented, and by the construction of g_2, we also have that $\mathsf{dom}(g_1) \subseteq \mathsf{dom}(g_2)$. The Id part follows from the frame property. □

We now prove that the $\mathsf{delete}()$ operation of the doubly-linked list module simulates the $\mathsf{delete}()$ operation of the XOR-linked list module, with respect to the refinement relation $R * \mathsf{Id}$.

Lemma 33. *The following holds:*

$$\eta(\mathsf{delete}(a))[\mathsf{fsim}(R * \mathsf{Id}, R * \mathsf{Id})]\varepsilon(\mathsf{delete}(a)).$$

Proof. Let $h_1[R]h_2$. Then there exist states h'_1, h'_2 and h_0 such that $h_1 = h'_1 \cdot h_0$ and $h_2 = h'_2 \cdot h_0$ and $h'_1[R]h'_2$. That means that $h'_1 \in \mathsf{xlist}\ \alpha(i, nil, nil, m)$ and $h'_2 \in \mathsf{dlist}\ \alpha(i, nil, nil, m)$.

We first prove the first condition of forward simulation, so let $\neg h_1[\varepsilon(\mathsf{delete}(a))]\mathsf{wrong}$. Then, a is already in the sequence α, so neither $\eta(\mathsf{delete}(a))$ goes wrong on state h_2.

Now we prove that the second condition of the forward simulation holds. Let g_2 be such that $h_2[\eta(\mathsf{delete}(a)]g_2$ and let $\neg h_1[\varepsilon(\mathsf{delete}(a))]\mathsf{wrong}$. Then by the definition of the operation,

$$g_2 = h''_2[j+1 \mapsto l, l+2 \mapsto j]$$

where h''_2 is such that $h_2 = h''_2 \cdot k \mapsto a, l, j$. Then l and j are the delimiters of the two sublists of sequences β and γ, such that the original sequence was $\alpha = \beta \cdot a \cdot \gamma$. Let

$$g_1 = h''_1[j_1 + 1 \mapsto l_1, l_1 + 1 \mapsto j_1]$$

where h''_1 is such that $h_1 = h''_1 \cdot n \mapsto a, l_1 \oplus j_1$ and, l_1 and j_1 are the delimiters of the sublists that represent sequences β and γ. Then we claim that g_1 is the state that we are looking for. Firstly, $h_1[\eta(\mathsf{insert}(a))]g_1$ follows from the construction of g_1. Secondly, $g_1[R * \mathsf{Id}]g_2$ because, in both states g_1 and g_2 the same sequence $\beta \cdot \gamma$ is represented, and by the construction of g_2, we also have that $\mathsf{dom}(g_1) \subseteq \mathsf{dom}(g_2)$. The Id part follows from the frame property. □

Similarly as in the previous case, we can now conclude according to the previous two lemmas and soundness of the forward simulation with growing relations, that the doubly-linked list module data refines the XOR-linked list one.

Theorem 12. *Module (q, ε) data refines module (p, η).*

Proof. The proof follows from lemma 32 and lemma 33 and soundness result for the forward simulation, Theorem 6. □

Finally, we can prove the equivalence of the doubly-linked and XOR-linked representations of the doubly-ended queues, using the results we have shown so far.

Theorem 13. *Modules (p, η) and (q, ε) are equivalent.*

Proof. The proof follows from Theorem 11 and Theorem 12. □

8

Conclusion and Future Work

It has long been known that pointers cause great difficulties in the treatment of data abstraction [40, 39, 10, 20]. This has lead on to a non-trivial body of research. The focus of our work on problems caused by low-level pointer operations, sets it apart from all this other research.

In order to be able to consider different data representations of an abstract data structure and allow pointers and their manipulations, we defined a class of programs which are guaranteed to have a "nice" behavior. Namely, separation contexts are client programs which, using the services of a certain module, change or read the state of a module only indirectly, that is through the operations that module provides through its interface. We introduced a concept of a separation context semantically. We tweaked the standard semantics of a small while-language by parameterizing it by a module and by adding additional rules which can detect any illegal attempts of a client to dereference module's part of the state. Then, it can be checked in the semantics whether a certain client program is a separation contexts. However, there is a more practical way of checking if a program is a separation context. That can be done using a program logic, in particular separation logic. As we showed, by proving that a program satisfies certain specification in an environment which contains a module, we obtain that the program then also has to be a separation context for the relevant precondition and assumed module. Our hopes are that, as static checking tools like Smallfoot develop, this can give rise to static checking methods for imperative modularity.

We have introduced two methods for proving data refinement: forward simulation method and power simulation method. Once simulation is proved between two modules, we are interested in ability to lift the simulation to the whole language. Both lifting and soundness largely depend on whether the programs in question are separation contexts for given modules. In fact, neither of the methods put any requirements on programs that are not separation contexts.

The forward simulation method employs a binary relation between the states of the abstract program and its concrete counterpart. It is based on a requirement that the refinement relation is preserved by the operations of the module. In order to prove that one operation forward simulates another, one must ensure that starting from related states, what ever step the concrete operation makes, the abstract one is able to perform the same step in such a manner that the refinement relation between the output states still holds. For lifting and soundness results, we had to restrict refinement relations only to the growing ones, i.e. where the abstract state is in a certain sense smaller than the concrete one. This is the crucial restriction which ensures the soundness of the method. The downfall of this restriction is that certain examples, such as the equivalence of the doubly-linked and XOR-linked list representations of the doubly ended queues, cannot be tackled using the forward simulation method.

For more advanced applications, we introduced a more general method, a power simulation method. The method is based on using the so called power relations, relations between a concrete state and a set of abstract states. The principle on which it is based is the same as in the forward simulation method, except that here instead of the ordinary execution of the abstract program on a single state, we have its lifted execution on a set of states. An important presumption for the power simulation method and its soundness is that all the involved power relations are admissible. While the admissibility condition ensures the soundness of the method, it removes the restriction that the state has to grow as it gets more concrete. This allows us to consider a great deal of examples that the forward simulation method could not handle.

The difficulty with the power simulation method is that it consists of complex objects, such as power relations, sets of states, etc., which are harder to deal with than the simpler data of forward simulation method. The lucky circumstance is that in some special cases, which are of great importance, the power simulation method has a state based representation which is by far easier to manipulate.

Now, it may seem that the problems that we target here arise only because of language bugs. Indeed, previous work has relied strongly on protection mechanisms of high-level, garbage collected languages. However, we would counter that a comprehensive approach to abstraction cannot be based on linguistic restrictions, particularly on limiting cross-boundary pointers. 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. There is no inherent reason why the *idea* of refinement of modules should not be applicable to it, as demonstrated in this work.

There are several directions in which our work on data refinement could expand.

Completeness. We introduced a power simulation method in the thesis and proved its soundness. We suspect that the power simulation method is also complete, but to prove that is far more challenging. We leave that for future work.

General theory of data refinement. It would be useful to produce a general theory of data refinement which would give us a way to connect two heap models, each a partial commutative monoid. This would allow us to refine the heap and hence the resources, in stages. To illustrate, one might start with a sequence-based representation of a queue data type, refine it with a linked-list representation in a heap where pointer identities are opaque (like in ML), and then refine again to a heap where the addresses are integers (like in C or assembly).

Unconstrained multiple modules. We have considered a setting in which a client program interacts with only one module. A possible future direction could be to extend our setting so that a client may interact with several modules. We suspect that the extension where modules are completely independent would be straightforward. However, more interesting situation is where modules might share resources. For example, if one module "owns" a tree and another "owns" a list, it is possible that the list itself consists of the leaves of the tree of the tree-module.

Multiple-instance classes. In our work we considered only single-instance classes and we do not examine inheritance and behavioral subtyping, concepts of importance for the object-oriented programming. It would be worthwhile to extend our setting to include some of the features of the object-oriented approach.

Concurrency. Our work on data refinement relies heavily on Separation logic. There has been a significant progress in Separation logic in work on concurrency [59], in particular, shared-memory programs. One possible direction for future work is to use ideas from concurrent separation logic and apply them in the field of data refinement in a concurrent setting.

Refinement calculus. Our work is still in a conceptual stage, analogous to earlier Hoare's work [35], which set down a conceptual framework, but which came before the refinement calculus. One future direction that imposes itself is a development of a refinement calculus, which would give a more practical approach to data refinement.

Tool. Finally, in today's world where automatic and semi-automatic tools are mushrooming, it is impossible not to think of developing a tool as a possible future direction. In fact, there are several refinement tools that are used for development of (safety-critical) software [27, 46, 1], and that adds to the motivation.

Index

Bibliography

[1] `http://www.vdmbook.com/tools.php`.

[2] J. R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[3] J. R. Abrial. Event based sequential program development: application to constructing a pointer program. In *FME 2003*, pages 51 – 74. Springer, 2003.

[4] P.S. Almeida. Balloon types: Controlling sharing of state in data types. In *ECOOP97Object-Oriented Programming, 11th European Conference*, volume 1241 of *LNCS*, pages 32–59. Springer-Verlag: Jyväskylä, Finland, 1997.

[5] A.Rezazadeh and M. Butler. Some guidelines for formal development of web-based applications in B-method. In Helen Treharne, Steve King, Martin C. Henson, and Steve Schneider, editors, *ZB05*, volume 3455 of *Lecture Notes in Computer Science*, pages 472–492. Springer, 2005.

[6] R. J. Back. On the correctness of refinement steps in program development. Technical Report A-1978-4, Department of Computer Science, University of Helsinki, 1978.

[7] R. J. Back, X. Fan, and V. Preoteasa. Reasoning about pointers in refinement calculus. In *Proceedings of the Tenth Asia-Pacific Software Engineering Conference (APSEC'03)*, 2003.

[8] A. Banerjee and D. Naumann. Ownership confinement ensures representation independence in object-oriented programs. *Journal of the ACM*, 52(6):894 – 960, November 2005.

[9] A. Banerjee and D. Naumann. State based ownership, reentrance and encapsula-

tion. In *Proceedings of the Nineteenth European Conference on Object-oriented Programming (ECOOP)*, volume 3586 of *LNCS*, pages 387 – 411. Springer-Verlag, July 2005.

[10] A. Banerjee and D. A. Naumann. Representation independence, confinement and access control [extended abstract]. In *Proceedings of 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL02)*, pages 166–177. ACM Press, 2002.

[11] Anindya Banerjee and David A. Naumann. Ownership: Transfer, sharing, and encapsulation.

[12] M. Barnett, R. DeLine, M. Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *JOT*, 3(6), 2004.

[13] J. Berdine, C. Calcagno, and P. W. O'Hearn. A decidable fragment of separation logic. In K. Lodaya and M. Mahajan, editors, *FSTTCS 2004*, volume LNCS 3328, pages 97–109, December 2004.

[14] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In K. Yi, editor, *APLAS 2005*, volume LNCS 3780, pages 52–68, November 2005.

[15] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO 2006*, November 2006.

[16] R. Bornat. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction*, 2000.

[17] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.

[18] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *POPL'03*, 2003.

[19] C. Boyapati and M. Rinard. A parameterized type system for race-free java programs. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2001.

[20] M. Butler. Calculational derivation of pointer algorithms from tree operations. *Science of Computer Programming*, 33:221–260, 1999.

[21] M. J. Butler and M. M. R. Meagher. Performing algorithmic refinement before data refinement in B. In *Proceedings of Proc. ZB2000: Formal Specification and Development in Z and B*, pages 324–343, 2000.

[22] C. Calcagno, J. Berdine, and P. W. O'Hearn. `http://www.dcs.qmul.ac.uk/research/logic/theory/projects/smallfoot/index.html`.

[23] D. Clarke and T. Wrigstad. External uniqueness is unique enough, 2003.

[24] D. G. Clarke and S. Drossopoulou. Ownership, encapsulation and disjointness of type and effect. In *OOPSLA 2002*, November 2002.

[25] D. G. Clarke, J Noble, and J. M. Potter. Simple ownership types for object containment. In *Proc. European Conference on Object-Oriented Programming*, June 2001.

[26] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1998.

[27] ClearSy. `http://www.atelierb.societe.com/index_uk.html`.

[28] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, second edition, January 2001.

[29] Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Number 47 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, UK, November 1998.

[30] E. Dijkstra. *Discipline of Programming*. Prentice-Hall, 1976.

[31] P. H. Gardiner and C. C. Morgan. Data refinement of predicate transformers. *Theoretical Computer Science*, 87:143–162, 1991.

[32] P. H. B. Gardiner and C. Morgan. A single complete rule for data refinement. *Formal Aspects of Computing*, 5:367 – 382, 1993.

[33] J. He, C. A. R. Hoare, and J. W. Sanders. Data refinement refined (resume). In B. Robinet and R. Wilhelm, editors, *ESOP 86, European Symposium on Programming*, volume 213 of *Lecture Notes in Computer Science*, pages 187 – 196. Springer Verlag, 1986.

[34] C. A. R. Hoare. An axiomatic approach to computer programming. *Communications of the ACM*, 12(583):576 – 580, 1969.

[35] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271 – 281, 1972.

[36] C. A. R. Hoare, J. He, and J. W. Sanders. Prespecification in data refinement. *Information Processing Letters*, 25(2):71 – 76, May 1987.

[37] CAR Hoare and J. He. The weakest prespecification. Technical Report PRG-44, Oxford University Computing Laboratory, June 1985.

[38] CAR Hoare and J. He. Data refinement in a categorical setting. Technical Report PRG-90, Oxford University Computing Laboratory, November 1990.

[39] J. Hogg. Islands: Aliasing protection in object-oriented languages. In *OOPSLA'91*, 1991.

[40] J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt. The geneva convention on the treatment of object aliasing. *OOPS Messenger*, 1992.

[41] S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Principles of Programming Languages*, volume 28, London, 2001. ACM - SIGPLAN.

[42] C. B. Jones. *Software development: a rigorous approach*. Prentice-Hall, 1980.

[43] N. Krishnaswami and J. Aldrich. Permission-based ownership: encapsulating state in higher-order typed languages. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 96 – 106, 2005.

[44] K. Rustan M. Leino and Peter Müler. A verification methodology for model fields. In *ESOP*, pages 115 – 130, 2006.

[45] K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In Marting Odersky, editor, *Proceedings of ECOOP*, volume 3086 of *LNCS*, pages 491 – 516. Springer-Verlag, 2004.

[46] M. Leuschel and M. Butler. ProB: A model checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.

[47] N. Lynch. Simulation techniques for proving properties of real-time systems. Technical Report MIT/LCS/TM-494, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1993.

[48] N. Lynch and F. Vaandrager. Forward and backward simulations — part i: Untimed systems. Technical Report MIT/LCS/TM-486, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1993.

[49] N. Lynch and F. Vaandrager. Forward and backward simulations — part ii: Timing-based systems. Technical Report MIT/LCS/TM-487b, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1993.

[50] U. Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley Publishing Company, 1989.

[51] I. Mijajlovic and N. Torp-Smith. Refinement in a separation context. In *SPACE*, 2004.

[52] I. Mijajlovic, N. Torp-Smith, and P. O'Hearn. Refinement and separation contexts. In K. Lodaya and M. Mahajan, editors, *FSTTCS*, volume LNCS 3328, pages 421–433, 2004.

[53] I. Mijajlovic and H. Yang. Data refinement with low-level pointer operations. In Kwangkeun Yi, editor, *Programming Languages and Systems*, volume LNCS 3780, pages 19–36, November 2005.

[54] C. Morgan, K. Robinson, and P. Gardiner. On the refinement calculus. Technical Report PRG-70, Oxford University Computing Laboratory, October 1988.

[55] P. Müller and A. Rudich. Ownership transfer in Universe Types. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2007. To appear.

[56] P. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2), June 1999.

[57] P. O'Hearn, D. J. Pym, and H. Yang. Possible worlds and resources: The semantics of BI. *Theoretical Computer Science*, 315(1):257–305, 2003.

[58] P. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *POPL'04*, 2004.

[59] P. W. O'Hearn. Resources, concurrency and local reasoning. In *Proceedings of CONCUR'04*, volume 3170, pages 49 – 67. LNCS, 2004.

[60] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of 15th Annual Conference of the European Association for Computer Science Logic: CSL 2001*, Lecture Notes in Computer Science, Berlin, 2001. Springer-Verlag.

[61] M. Parkinson. *Local reasoning for Java*. PhD thesis, University of Cambridge, 2005.

[62] M. Parkinson, R. Bornat, and C. Calcagno. Variables as resource in hoare logic. In *Proceedings of LICS*, 2006.

[63] P.Behm, P.Benoit, A. Faivre, and J.-M. Meynadier. Mtor: A successful application of b in a large project. In *FM'99 - Formal Methods: World Congress on Formal Methods in the Development of Computing Systems*, volume 1708 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1999.

[64] G. D. Plotkin. Lambda definability and logical relations. Technical Report SAI-RM-4, School of Artificial Intelligence, University of Edinburgh, 1973.

[65] A. Potanin and J. Noble. Checking ownership and confinement properties, 2002.

[66] A. Potanin, J. Noble, D. Clarke, and R. Biddle. Generic ownership for generic java. In *Proceedings of the 2006 OOPSLA Conference*, volume 41, 2006.

[67] U. S. Reddy and H. Yang. Correctness of data representations involving heap data structures. In P. Degano, editor, *Proc. of the 12th European Symposium on Programming, ESOP 2003*, pages 223 – 237. Springer Verlag, 2003.

[68] Uday S Reddy. Talk at MFPS, 2000.

[69] J. C. Reynolds. *Types, Abstraction and Parametric Polymorphism*, pages 513 – 523. Elsevier Science Publishers, 1983.

[70] J. C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

[71] J. C. Reynolds. Lists and arrays, 2002. Slides from the Course Reasoning about Low-Level Programming Languages held at Carnegie Mellon University, Spring 2002. Available from the course's home page.

[72] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of Logic in Computer Science*, volume 17, pages 55 – 74, Copenhagen, July 2002. IEEE.

[73] J. C. Reynolds. Precise, intuitionistic, and supported assertions in separation logic, May 2005. Slides from the invited talk given at the MFPS XXI Available at authors home page: `http://www.cs.cmu.edu/ jcr/`.

[74] J. Schwarz. Generic commands - a tool for partial correctness formalisms. *Computer Journal*, 10(2):151 – 155, 1977.

[75] R. D. Tennent. *Semantics of Programming Languages*. Prentice-Hall International (UK) Ltd., 1991.

[76] J. Vitek and B. Bokowski. Confined types. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 82–96. ACM Press, 1999.

[77] Alan Wills. Capsules and types in fresco: Program verification in smalltalk. In *ECOOP 1991*, volume LNCS 512, pages 313–323. Springer-Verlag, July 1991.

[78] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. Foundations of Computing. MIT Press, 4th edition, 1997.

[79] N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, April 1971.

[80] H. Yang. *Local Reasoning for Stateful Programs*. PhD thesis, University of Illinois, 2001.

[81] H. Yang and P. O'Hearn. A semantic basis for local reasoning. In *Proceedings of FOSSACS 2002*, 2002.