DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Verification of the Decrease-Key Operation in Fibonacci Heaps in Imperative HOL

Simon Griebel

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Verification of the Decrease-Key Operation in Fibonacci Heaps in Imperative HOL

Verifikation der Decrease-Key Operation in Fibonacci Heaps in Imperative HOL

Author:	Simon Griebel
Supervisor:	Prof. Tobias Nipkow, Ph.D.
Advisor:	Maximilian P. L. Haslbeck
Submission Date:	15.05.2020

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Erding, 14.05.2020 Simon Griebel

Abstract

In this thesis, we present a proof of the functional correctness of the Fibonacci heap decrease-key operation using a separation logic framework implemented for Imperative HOL, an imperative verification framework built on top of Isabelle/HOL. This is a follow-up on Daniel Stüwe's Master's thesis [17], where he verified all Fibonacci heap operations (most notably delete-min) except for decrease-key and delete, which require a handle to the element to be changed or removed, respectively. His approach is based on first verifying a functional version of the Fibonacci heap and then showing that the imperative implementation correctly models the functional one. Using an augmented version of his approach, which also works for decrease-key, turned out to be too complex. Thus, the decrease-key operation was proven with an alternative approach where the heap is modeled as a map from addresses to (abstract) heap nodes. Together with Daniel Stüwe's thesis, this means that the complete functionality of the Fibonacci heap was verified, even though the two different approaches are not directly compatible. For that reason, we also proved the functional correctness of make-heap, heap-union, get-min, and of a function creating a heap with a single element with the help of the new approach.

Contents

1 Introduction

One of the most important aspects of programming is the avoidance of bugs in code. Normally this is achieved with the help of tests and code inspection, but these are generally not enough to guarantee the complete absence of bugs in code. This is especially problematic for systems that are deployed in very critical environments like medical or aerospace systems. Formal methods are powerful enough to ensure the correctness of a program, but the used approaches either only work for smaller systems (or only parts of the program) and/or take quite a lot of time. Because of that, these methods are not widely used and why further research is needed to improve these methods. One way to prove the correctness of a program is to use a theorem prover. This approach is quite powerful, because there are generally very few (theoretical) restrictions as to what can be shown, but currently proving programs with theorem provers requires a lot of manual work. This is especially a problem when imperative programs are proven, which are generally much more difficult to describe mathematically than functional programs.

One currently used approach to verify imperative programs is separation logic, which was first presented by Reynolds [16]. It is an extension to standard logic that includes the concept of addressable heap memory. The goal of this thesis was to verify an imperative implementation of a Fibonacci heap, to observe how well (basic) separation logic is able to deal with the problems posed by the data structure.

A Fibonacci heap is a heap data structure (also known as a priority queue), which was first presented by Fredman and Tarjan [8]. The implementation for this thesis is heavily based on the version of Fibonacci heaps found in the standard algorithm textbook by Cormen et al. [5]. The main reason for the development of the Fibonacci heap was that it efficiently implements decrease-key (with a $O(1)$ amortized time complexity), which means that if it is used for example in Dijkstra's pathfinding algorithm [6], it reduces the running time complexity from $O((V + E)\log(V))$ (for standard heaps like a binary heap) to $O(V\log(V) + E)$, where V is the number of nodes and E the number of edges of the graph. Despite of that, Fibonacci heaps are not widely used, which is probably mainly due to the pointer based structure (compared to the array based binary heap), which means that the complexity reduction probably does not translate well into actual running time improvements [5]. Still, the Fibonacci heap is an interesting structure to verify, as its analysis is quite complicated, while its interface is relatively simple. A main difficulty factor in the analysis are the two functions decrease-key and delete, which both require a handle (a pointer) to the node that should be changed/removed. Since decrease-key is the more important of the two functions and since delete can be directly implemented with decrease-key, it was the main focus of this work. The reason why the verification of decrease-key is so difficult is that its functionality is not easily representable in a purely

functional setting [3]. This means that the standard trick of verifying an imperative algorithm by first verifying a functional representation of the same algorithm and then showing that the functional representation correctly abstracts the imperative version does not work as easily.

This thesis is a direct follow-up on Stüwe's master's thesis [17]. In his thesis, Stüwe presents the verification of the functional correctness and of the running time bounds of an imperative Fibonacci heap which implements all heap functions except for decrease-key and delete. For the verification he used a functional tree data structure as an abstract representation of the heap. As extending his approach for the correctness proof for decrease-key did not work as anticipated, an alternative approach was developed as part of this thesis, with which it was possible to verify decrease-key. Here the abstract representation of the heap is simply a map from addresses to heap nodes. The new approach made it necessary to reverify all of the heap functions that were already verified by Stüwe. Due to the time constraints on this thesis only get-min, heap-union, the functions that create an empty heap (make-heap), and a heap with only one element (heap-singleton) were verified in addition to decrease-key. The running time proofs and proving more complex algorithms that make use of the Fibonacci heap are left for future work.

The implementation of the Fibonacci heap was done in Imperative HOL [4], an imperative verification framework implemented in Isabelle/HOL [13]. The used separation logic framework was built on top of Imperative HOL by Lammich and Meis [11].

First we introduce the Fibonacci heap data structure (Chapter 2) and the used verification framework (Chapter 3). This is followed by the main part of this work, where the used verification approaches are described (Chapter 4). Here we summarize Stüwe's approach (Section 4.1), as the first approach that was developed as part of the work for this thesis (Section 4.2) is heavily based on it. As this approach did not work as expected, we developed another approach where the Fibonacci heap is represented as a map from addresses to nodes (Section 4.3). With it we were able to verify the functional correctness of decrease-key (Section 4.3.7) and of the other above mentioned heap functions (Section 4.3.8). Before finishing the the thesis with an outlook on future work (Chapter 5) and the conclusion (Chapter 6), a third approach, that stayed mostly a thought experiment, is presented (Section 4.4). Here the heap is represented as a composition of paths from the root to the single nodes.

2 Fibonacci Heap

In this chapter we summarize the Fibonacci heap data structure, which was introduced by Fredman and Tarjan [8]. The implementation used in this thesis is mostly based on the version found in the algorithms textbook by Cormen et al. [5]. First the interface of the Fibonacci heap is presented (Section 2.1). After that the structure of the heap (Section 2.2) and its invariant (Section 2.3) will be explained in more detail, before finally the implementation of the functions will be described (Section 2.4). The concrete implementations in Imperative HOL can be found later, where these functions are also verified (Section 4.3.4 for decrease-key, Section 4.3.8 for the other heap functions).

2.1 Interface

The (minimum) Fibonacci heap supports the following basic heap operations, all of which, except for delete-min, return the changed heap object (more on that in the next subsection):

make-heap() Returns an empty heap.

insert(h, el) Inserts a new element el into the given heap h. The element has to contain its priority. In the simplest case the element itself is the priority.

get-min(h) Returns the element in the heap with minimum priority.

delete-min(h) Deletes the element with minimum priority from the given heap.

heap-union(h1, h2) Merges the two given heaps to one heap

In addition to the above functions the Fibonacci heap defines the following two operations, which both require a handle to the element in question. In this implementation the handle is just the address of the heap node, which contains the element.

decrease-key(h, handle, new_val) Decreases the priority of the element defined by the handle to the given value. The new value must be smaller than the already stored one.

delete(h, handle) Deletes the element indicated by the given handle

Theoretically Fibonacci heaps offer better time bounds for many operations than other heaps like array-based binary heaps as shown in Table 2.1. The problem is that due to their complex, pointer-based structure, which is explained in more detail in the next subsection, these theoretical bounds most likely do not translate to the real world, where modern processors are much more efficient in dealing with array based structures.

	get-min	delete-min	insert	union	decrease-key
Binary Heap	$O(1)$	$O(\log n)$	$O(\log n)$	$O(n)$	$O(\log n)$
Fibonacci Heap	$O(1)$	$O(\log n)$	$O(1)$	$O(1)$	$O(1)$

Table 2.1: running time comparison between a binary heap and a Fibonacci heap. The Fibonacci heap running times require an amortized analysis. Information taken from [5].

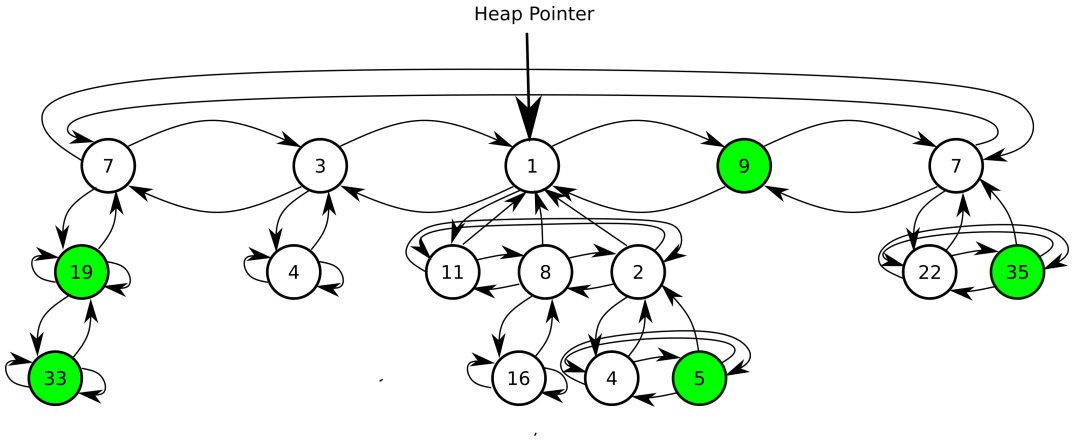

Figure 2.1: A visualization of a minimum Fibonacci heap with all pointers (that are not null). Note that the nodes "5" and "35" have to be marked, otherwise the shown structure would not be a valid Fibonacci heap.

2.2 Structure

The Fibonacci heap is a circular linked list of Fibonacci trees. This list will from now on be called *root list* as it stores the root nodes of the trees. A Fibonacci tree can be defined recursively as a root node, that can have an arbitrary amount of Fibonacci trees as children, which are again stored in a circular doubly linked list (the *child list*). Each heap node contains a Boolean *mark*, a (non-negative) integer *rank*, the stored value and 4 pointers: a pointer to the parent node, a pointer to the first node in the child list, a pointer to the previous node and a pointer to the next node. The next and previous pointers are used to concatenate the child or the root lists. The rank is just another name for size of the child list and the mark denotes whether the node has lost a child, while it was not in the root list. These last two attributes are solely needed for the amortized analysis of the data structure and they are described in more detail in the next section where the heap invariant is discussed. The parent pointer of nodes in the root list is set to be the null pointer. A visualization of this structure can be seen in Figure 2.1. The object that represents the heap simply contains a pointer to the node containing the element with minimum priority in the root list (see the next subsection for more details). This pointer will from now on be called *heap pointer*. If desired, the heap object can also contain the number of stored elements in the heap as an integer. This is only required

for the implementation of an efficient size function as the other heap operations do not make use of it. For that reason the size attribute is left out for the heap implemented as part of this thesis (though the Fibonacci heap implementation by Stüwe [17] presented in Section 4.1.2 does contain it). The elements stored in the heap normally either consist of a priority-value-pair or the elements have some way of exposing their priority (for example with a function provided to the heap implementation). For simplicity's sake the implementation used in this thesis assumes that the stored value is the priority.

2.3 Invariant

Apart from the heap structure, there are two main parts of the heap invariant. The first part is responsible for the order of elements in the heap. Here the basic heap invariant, also found in other heap structures, holds. It states that the priority of each child is either smaller or bigger than the priority of the parent depending on whether a min or max heap is implemented. From now on it is assumed that the heap is a min heap. The root list itself is not ordered, but the pointer stored in the heap object is guaranteed to always point to the node containing the element with minimum priority (in the root list and due to transitivity the element with minimum priority in the whole heap). The second part of the invariant is responsible for ensuring the running time behavior of the heap. Here the rank of each node has to be equivalent to the number of nodes in its child list. Additionally the rank of each node in a child list has to be bigger or equal to its index in the child list (starting with 0 for the first node). If a node is marked, its rank has to be bigger or equal to $i - 1$, where i is the index. The mark indicates that the node has lost a child during the execution of decrease-key. These two rules, that connect the index of a node to its rank, are called *child rank invariant* in this thesis. This child rank invariant does not apply to nodes in the root list, which can have an arbitrary amount of children. While there may be marked nodes in the root list, this mark has no effect, and each of these nodes is unmarked before they are readded to a child list. Note that even though the restrictions on the rank will be proven in the verification, the amortized analysis, for which these restrictions are required, was not verified in this thesis.

2.4 Implementation Overview

The Fibonacci heap can be seen as a somewhat "lazy" data structure, where each operation is as cheap as possible and most of the work is done in delete-min, which restructures the tree. The basic operations, of which only get-min and heap-union were verified as part of this thesis, work as follows:

insert(h, el) A new node containing the given element is created and appended to the root list. If the new element is smaller than the old minimum, reset the heap pointer to the new node.

get-min(h) Return the value stored in the node pointed to by the heap pointer.

delete-min(h) Deletes the minimum of the tree (indicated by the heap pointer). The children of the deleted node and all of the trees stored in the root list are merged together to a new heap according to relatively strict rules. For more details see the algorithm textbook [5].

heap-union(h1, h2) Concatenates the two root lists. The pointer to the smaller of the two minimum elements becomes the new heap pointer of the merged heap.

Decrease-key is makes use of two helper functions: cut and cascading-cut. They work as follows:

cut(H,child-addr, parent-addr) Removes the node indicated by the child address from the child list of the parent and adds it to the root list. The removed child node is unmarked and its parent pointer is set to the null pointer.

cascading-cut(H, addr) This function is used to restore the child rank invariant of the given node, which just lost a child, (indicate by its address).

- If the node is in the root list nothing has to be done.
- If the node is not marked, it is marked and cascading-cut returns.
- If the node is marked, it is cut from its parent and cascading-cut is called for the parent.

decrease-key(h, handle, new_val) There are three main cases that have to be considered for decrease-key:

- If the new priority of the node indicated by the handle is still bigger or equal to the priority of the parent node, it can simply be reduced without affecting the validity of the heap.
- If the node has no parent, it is in the root list, so it only has to be checked whether the new priority is smaller than the old minimum of the tree and if so the heap pointer has to be reset.
- If the node whose priority will be decreased has a parent and the new priority is smaller than the priority of the parent, then the node has to be cut from its parent. It is subsequently added to the root list, so that it can be decreased there. This works as in the previous case. Since cut might violate the child rank invariant of the parent, cascading-cut is called for the parent, to restore the invariant.

A visualization decrease-key can be seen in Figure 2.2.

This implementation differs from the one described in standard algorithm book [5] as there the actual decrease of the priority is done before the cascading-cut is called (while in the presented implementation it is done afterwards). The reason for this change is that doing the decrease after cascading-cut leads to a slightly easier verification. This will be explained when the verification is done in Section 4.3.5. As only the order and not the amount of operations is changed, this does obviously not affect the running time.

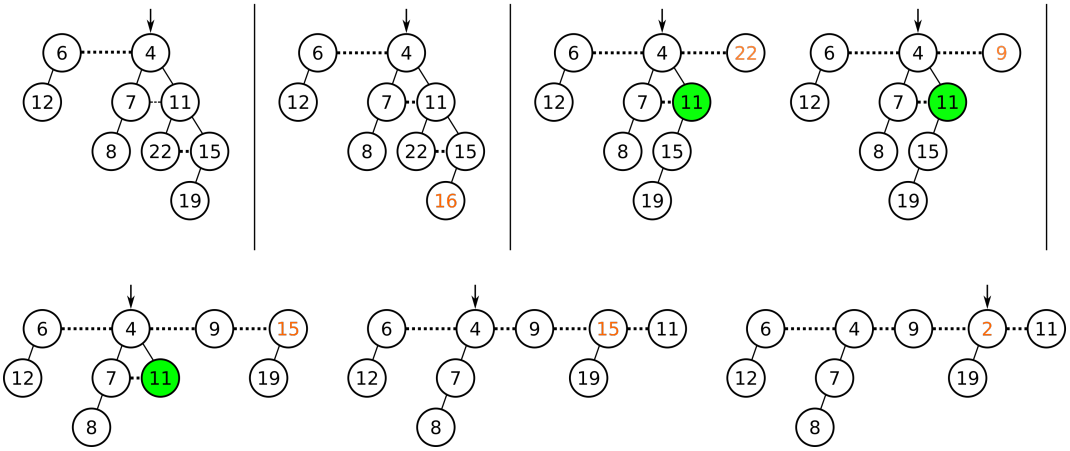

Figure 2.2: A visualization of the decrease-key operation for three different nodes. The values of the nodes which will be decreased are orange. Each shown heap represents a step of decrease-key. Separate decrease-key operations are separated by a vertical line. The first heap shows the initial state.

The function delete can be implemented with the help of decrease-key and delete-min in the following way:

delete First the priority of the element to be deleted is decreased to a value smaller than the current minimum and then delete-min is called.

The actual implementations of the verified functions in Imperative HOL are shown directly before their verification. For decrease-key this is in Section 4.3.4 and for the other heap functions in Section 4.3.8.

3 Verification Framework

In this chapter the tools used for the verification are presented. The basis is Isabelle/HOL [13] a general higher-order logic framework (Section 3.1). On top of that Bulwahn et al. [4] implemented Imperative HOL, which allows reasoning about imperative programs (Section 3.2). As the verification support offered by Imperative HOL was quite basic, the framework was extended by Lammich and Meis to support separation logic (Section 3.3), which is an extension to predicate logic that adds the notion of addressable heap memory.

3.1 Isabelle/HOL

Isabelle is an interactive theorem prove, that is based on a fragment of higher-order logic [15]. On top of this so called meta-logic a more expressive higher order logic framework called Isabelle/HOL is built [13]. It supports standard logical operators known from predicate logic. Due to the described structure, there are two versions of the most central ones. The two that appear in this paper are implication $\Longrightarrow$ and a version of the universal quantifier $\bigwedge$. There are two equivalent ways to define mathematical lemmas with assumptions in Isabelle/HOL:

```
lemma example: "A ⟹ B ⟹ C"
```

```
lemma example:
assumes "A" "B"
shows "C"
```

Easier lemmas can be proven by applying certain proof methods that transform the statements (for example with the help of induction) so that resulting statements can then be proven automatically by the solvers supported by Isabelle/HOL. For more complex proofs there is a proof language called Isar [18], which makes it possible to split an otherwise monolithic proof into several smaller proofs of subgoals, which can again be proven with proof methods and solvers.

Isabelle/HOL also supports some concepts known from functional programming. Types can be defined with the keywords **datatype** or **record**. As an example, the Isabelle/HOL implementation of a functional list looks as follows:

```
datatype (set: 'a) list =
  Nil  ("[]")
| Cons (hd: 'a) (tl: "'a list")  (infixr "#" 65)
```

Certain expressions can be given names with the **abbreviation** and **definition** keywords. Potentially recursive functions that make use of pattern matching can be defined with the **fun**, **function** and **partial_function**. The former two may contain pattern matching. The key property of functions defined with the **fun** keyword is that they are

required to be (automatically) provably total. In turn the framework provides an induction scheme for the given function. As an example, the definition of and the automatically generated induction scheme resulting from the function that removes adjacent duplicates from a list look as follows:

```
fun remdups_adj :: "'a list ⇒ 'a list" where
"remdups_adj [] = []" |
"remdups_adj [x] = [x]" |
"remdups_adj (x # y # xs) = (if x = y then remdups_adj (x # xs)
                                      else x # remdups_adj (y # xs))"

lemma
assumes
   "P []"
   "⋀ x. P [x]"
   "⋀ x y xs. (x = y ⟹ P (x # xs)) ⟹ (x ≠ y ⟹ P (y # xs)) ⟹ P (x#y#xs)"
shows "P li"
```

Two other important structures that will appear during this thesis are indicated with **inductive** and **inductive_set**, which offer an inductive way of defining either Boolean predicates or sets, respectively. The general structure of these two look as follows:

```
inductive P :: "'a ⇒ bool" where
base_case: "precond1 ⟹ P x" |
inductive_step: "precond2 ⟹ P y
                          ⟹ P x"
```

```
inductive_set S :: "'a ⇒ 'b set"
for a :: 'a where
base_case: "precond1 ⟹ x ∈ S a" |
inductive_step: "precond2 ⟹ y ∈ S a
                          ⟹ x ∈ S a"
```

The given rules are independent of each other (so the `x` in the base cases is not the same as the `x`). The variable `a` is fixed across all cases. The preconditions may be arbitrary predicates over the used variables and in the inductive step they have to define the inductive relationship between `x` and `y`. There may of course be fewer or more than two rules for each inductive definition.

The Isabelle theories that were created as part of this thesis can be found here [7].

3.2 Imperative HOL

Imperative HOL is a framework implemented in Isabelle/HOL, which supports the verification of imperative programs. It was developed by Bulwahn et al. [4]. The heap memory of a program is represented with the following type:

```
record heap =
arrays :: "typerep ⇒ addr ⇒ heap_rep list"
refs :: "typerep ⇒ addr ⇒ heap_rep"
lim  :: addr
```

As can be seen this implementation differentiates between references to arrays and references to single elements. The Fibonacci heap implementation does not contain any arrays, so from now on only references to single objects will be used. The values stored, indicated by the `heap_rep` type, and the addresses used are simply aliases for positive

integers (starting at 0). Technically each type (indicated by `typerep`) has an independent address range, but the implementation of the allocation ensures that for each address there is only a single value or array of any type stored. This is done with the help of the limit `lim`. If a new value is allocated, it gets the address represented by the current value of the limit, which is subsequently incremented by 1. As this limit is global for the heap, this ensures no two elements of different types are stored at the same numerical address. Even though there may only be one element of any type at a single numerical address, for retrieving the value stored at a certain address the type information is also needed. This is achieved by introducing the following reference type (together with a helper function that returns the numerical address):

```
datatype 'a ref = Ref addr

primrec addr_of_ref :: "'a ref ⇒ addr" where
"addr_of_ref (Ref x) = x"
```

The function that is responsible for dereferencing a certain reference uses the generic value of `ref` to determine the `typerep` needed to retrieve the stored element. Nullable pointers are represented by the `'a ref option` type.

Any value of a (user defined) Isabelle/HOL type can be stored, as long as it is countable, meaning that there is a bijective function (that has to be provided by the user) that is able to transform the given type into a positive integer (the `heap_rep`). Values cannot be freed, so this can be thought of as a memory model using a (implicit) garbage collector. In addition to allocation, which is implemented with the `ref` function, there are read (`!`) and write (`:=`) operations, that can interact with the heap. Note that only whole Isabelle/HOL objects and not their members individually can be accessed with these functions. Functions that do not interact with the heap are implemented as normal Isabelle/HOL functions. To model the effect of all functions that interact with the heap, the heap monad, which is the return type of these functions, is introduced:

```
datatype 'a Heap = Heap "heap ⇒ ('a × heap) option
```

So each function models a heap action, that takes a heap as argument and potentially returns a value (the standard return value of the function) and the changed heap. This pair is wrapped in an option so that abnormal program termination (for example with assertions) can be modeled. The functions and heap operations can be chained with the monadic bind operator `>>=`. When the left hand side of it returns `None`, then also the bind operator returns `None`. This means that an abnormal termination in any part of the program leads to the whole program terminating abnormally, as `None` is passed along every bind operator to the end of the program. If the left hand side exits normally by returning a value (together with the changed heap), then the bind operator passes these two along to the next function (with the return value being passed before the heap). With these simple semantics a swap function that exchanges the value of two memory cells can be implemented as follows:

```
definition swap' :: "('a::heap) ref ⇒ 'a ref ⇒ unit Heap" where
"swap' x y = !x >>= (λ a. !y >>= (λ b. x := b >>= (λ _. y := a)))"
```

The heap in the type annotation of 'a must not be confused with the heap record defined above. Here it means that the annotated type must be storable on the heap as discussed above.

This definition is not very readable so Imperative HOL supports a Haskel like donotation [12]. With it the swap can be written as follows:

```
definition swap :: "('a::heap) ref ⇒ 'a ref ⇒ unit Heap" where
"swap x y = do {
   a ← !x;
   b ← !y;
   x := b;
   y := a
}"
```

The ← operator stores the return value of the function on the right hand side in the variable on the left. The value of the last expression is the return value of the function, which in this case is just a unit. Note that contrary to the normal imperative swap implementation, two auxiliary variables are needed, because the result of !b cannot be directly be stored at the memory location a, since it "returns" the heap monad, which first needs to be unfolded with the ← operator.

3.3 Separation Logic in Imperative HOL

Lammich and Meis extended the basic Imperative HOL framework to also support separation logic [11]. Separation logic itself was first introduced by Reynolds [16] and it extends normal predicate logic by the concept of addressable heap memory. A key new idea in separation logic (hence its name) is to include an operator, the *separating conjunction*, that indicates that two formulas over two address ranges do not apply (partly) to the same addresses. This will make it easier to deal with Hoare triples for imperative programs as will be shown later in this section. In the version of separation logic implemented by Lammich and Meis, the separation logic assertions are functions from a partial heap to a Boolean. A partial heap is simply a pair of a heap (as defined above) and an address set, which indicates the parts of the of the heap for which the assertion has to hold. To enforce this, a valid assertion must not depend on any value at an address outside of address set. Additionally each assertion may only be true for a partial heap, whose addresses are below the limit of the heap (as these addresses were not allocated yet, see the description in the previous subsection). The latter restriction is expressed by the function in_range (h,as). A valid assertion A is said to model a partial heap ph, written as A ⊨ ph, simply iff A(ph) = True.

According to this definition, any arbitrary function from a partial heap to a Boolean can be an assertion as long as it satisfies the two mentioned rules, but generally they are built from the following atoms and operators defined in Reynolds work on separation logic. The three most basic atoms are true, false and emp. The atom true is satisfied by any valid partial heap, the atom false by no partial heap and emp by the empty partial heap. In Isabelle/HOL this is expressed as follows:

```
lemma mod_true: "ph ⊨ true ⟷ in_range ph"
lemma mod_false: "¬ ph ⊨ false"
lemma mod_emp: "ph ⊨ emp ⟷ snd ph = {}"
```

The next important atom states what is stored at a single address. It contains an additional check for validity of the partial heap. With `get` being the dereferencing function of the heap, this atom is defined as follows:

```
lemma mod_sngr: "ph ⊨ r ↦_r x ⟷
      get (fst ph) r = x ∧ (snd ph) = {addr_of_ref r} ∧ addr_of_ref r < lim (fst ph)"
```

Note that this assertion can only hold for partial heaps that contain the address `r` in their address set and nothing else. This assertion will be referred to as *points-to* assertion during this thesis.

Boolean predicates that do not depend on the heap can also be made to assertions in the following way:

```
lemma mod_pure: "ph ⊨ ↑b ⟷ (ph ⊨ emp) ∧ b"
```

This type of assertion is also called a *pure* assertion.

These atoms can be connected by the standard operators and quantifiers lifted from predicate logic:

```
lemma mod_ex_dist: "ph ⊨ (∃_A x. P x) ⟷ (∃ x. ph ⊨ P x)"
lemma mod_and_dist: "ph ⊨ P ∧_A Q ⟷ ph ⊨ P ∧ ph ⊨ Q"
lemma mod_or_dist: "ph ⊨ P ∨_A Q ⟷ ph ⊨ P ∨ ph ⊨ Q"
```

Out of these the existential quantifier seems to be the most useful one for data structure analysis, as it can be used to hide internal state of a data structure from the caller of its functions. The path-based approach described in Section 4.4 actually uses $\wedge_A$, but the usage of this operator was one of the problems of the approach. There is no formal reason why the universal quantifier could not be lifted as well. It was just not done in this framework.

Now comes the most important operator: the already mentioned separation conjunction, which is defined as follows:

```
lemma mod_star_conv: "ph ⊨ A*B  ⟷
    (∃_A h as1 as2. ph=(h,as1 ∪ as2) ∧ as1 ∩ as2={} ∧ (h,as1)⊨ A ∧ (h,as2) ⊨ B)"
```

The separating conjunction states that a partial heap satisfying two assertions connected by it has to be splittable into two disjoint parts, each of which has to fulfill one of the two assertions. The separating conjunction is commutative and associative. The term "Π `x ∈ S. F x`", where F is a function returning an assertion, will be used to indicate a concatenation of the assertions generated from all the elements in S with the separating conjunction. It will be called *separating product* in this thesis. It is equivalent to the *iterated separating conjunction* defined by Reynolds [16].

The main way to prove the semantics of a program with the help of separation logic are Hoare triples [9]. A Hoare triple consists of a precondition, a postcondition, and a program. Instead of standard boolean predicates, a separation logic Hoare triple has a separation logic assertion as pre- and postcondition. A Hoare triple states that if the

program is executed on a state for which the precondition holds, the state after the program execution fulfills the postcondition. For the previously implemented swap a sensible separation logic Hoare triple looks as follows:

```
lemma swap_rule: "<x ↦r a * y ↦r b> swap x y <λ _. x ↦r b * y ↦r a>"
```

Note that the postconditon is a function, that takes the return value (which is ignored here, since it is just a `unit`) and returns an assertion. Since the separating conjunction is used, this Hoare triple is only applicable, when the two addresses `x` and `y` are not equal.

As the Hoare triple is defined here, the pre- and postcondition only match the partial heaps that contains only the addresses `x` and `y`, but of course swap also works for bigger heaps. This does not need to be stated explicitly as there is a fundamental lemma that holds for all separation logic Hoare triples: the frame rule. It states that the pre- and postconditions of any Hoare triple for any function may be extended with the help of the separating conjunction, without changing the validity of the triple. The corresponding Isabelle/HOL lemma can be defined as follows:

```
lemma frame_rule:
assumes "<P> c <λ x. Q x>"
shows "<P*R> c <λx. Q x * R>"
```

To be able to prove Hoare triples for functions that consist of many instructions, it is key to split the proof into smaller parts. In Imperative HOL a program consists of operations, that are connected with the monadic bind operator `>>=`, so the following lemma shows how the proof for a Hoare triple fo two operations connected with a monadic bind can be split into two Hoare triples, one for each used operation:

```
lemma bind_rule:
assumes "<P> f <λ x. R x>" "⋀x. <R x> g x <Q>"
shows "<P> f >>= g <Q>"
```

Assuming that a program for which a Hoare triple should be verified consists of only two operations connected with a monadic bind `>>=`, then the precondition of the first operation and the postcondition of the last operation are given as they are the pre- and poscondition stated in the Hoare triple for the whole program. This means that only the postcondition of the first and the precondition of the second operation, which are equal, have to be given.

As it turns out, it is relatively easy to generate this missing assertion, when a Hoare triple for the first operation is given. The reason for this is the frame rule. If the given precondition of the whole program contains terms that are connected by the separating conjunction, then the first step is to determine the terms that match the precondition of the given Hoare triple for the first operation. The postcondition of the operation in the context of the program is then just the postcondition of the Hoare triple of that operation plus the terms that didn't match the precondition. So for the swap, assuming the precondition `A * a ↦r x * B * b ↦r y * C`, where A, B, and C can be arbitrarily complex assertions, one can directly follow that the postcondition must be `A * a ↦r y * B * b ↦r x * C`.

Of course this approach also works for a program with more than two operations: Starting from the precondition of the first operation one can generate its postcondition, which is the precondition for the second operation, for which the postcondition can again be generated and so on. In the end it needs to be shown that the determined postcondition for the last operation in the program matches the postcondition supplied in the Hoare triple for the whole program that should be verified. These two postconditions do not have to exactly match. It is possible to weaken the postcondition. To express this, one first needs to lift the implication from predicate logic into separation logic, with the following definition:

```
definition entails :: "assn ⇒ assn ⇒ bool"
where "P ⟹_A Q ≡ ∀h. h⊨P ⟶ h⊨Q"
```

With this concept of entailment one can state, that one can arbitrarily "weaken" the postcondition of a Hoare triple:

```
lemma cons_post_rule:
<P> c <λ x. Q x> ⟹ (⋀x. Q x ⟹_A Q' x) ⟹ <P> c <λ x. Q' x>
```

The precondition of a Hoare triple can be arbitrarily "strengthened".

To enable the described automation for the three standard operators that interact with the heap (allocation, read and write), it is necessary to define Hoare triples for these, which are defined as follows:

```
lemma lookup_rule: "<p ↦_r x> !p <λr. p ↦_r x * ↑(r = x)>

lemma update_rule: "<p ↦_r y> p := x <λr. p ↦_r x>"

lemma ref_rule: "<emp> ref x <λr. r ↦_r x>"
```

Even though the described automation is quite powerful, it can only generate a postcondition for a read from or a write to a memory cell with address `x` if the precondition is of the form "`... * x ↦_r r * ...`" for arbitrary `r`. This will become relevant for the assertions for the imperative list and the Fibonacci heap, that will be introduced later, as they are defined with functions, which return the assertions describing these data structures. To get to the required assertion form, it is necessary to split the return values of these functions, so that the relevant points-to assertions are exposed.

4 Verification

We considered three approaches for the verification of decrease-key. The first one is based on the work done by Stüwe in his Master's thesis [17]. Here the imperative Fibonacci heap is represented by an abstract functional Fibonacci heap, which is used to generate the assertion that describes the former. In his approach, two key features of the heap are missing as they are not needed for the verification of the non-handle-based heap operations: the parent pointer of the nodes and the enforcement of fixed addresses of the heap nodes. That's why his approach had to be augmented for this thesis (Section 4.2). To better understand the augmentation, the structures used by Stüwe in his approach are presented first (Section 4.1). As the augmented version did not work as well as expected, another approach was developed (Section 4.3). Here the heap is modeled as a map from addresses to abstract heap nodes. With this approach decrease-key (Section 4.3.7) and a few other heap functions (Section 4.3.8) could be verified. For the verification a formalization of the concept of footprints, which are the sets of addresses for which an assertion can hold on any heap, is introduced (Section 4.3.1). Before the end of this chapter another approach is shortly described (Section 4.4). Here the heap representation is a composition of paths from the root list to the nodes. As this structure was not very easy to deal with, it stayed mostly a thought experiment.

4.1 Previous Work

The first approach that was tried for the verification of the decrease-key operation is just an extension of the one presented in Stüwe's Master's thesis [17], where he verified all heap functions, except for decrease-key and delete, in Imperative HOL. In this section his work is shortly summarized, so that the changes made as part of the augmentation of his approach can be compared to the original approach.

He modeled the imperative heap data structure and its operations with a functional abstraction. The latter was verified first and then it was shown that the functional representation actually models the imperative program. This method works here since all non-handle operations are easily representable in a functional setting, as they only operate on the root list and the direct children of the nodes in this list.

This chapter begins with the definition of the assertion Stüwe used to described a circular doubly linked list, the basic building block of Fibonacci Heaps, in Imperative HOL (Section 4.1.1). Directly after the definition it is explained, why this approach is not sufficient for the verification of decrease-key. Then his separation logic assertion for the heap plus the heap invariants, that are defined for the functional heap representation, are shown (Section 4.1.2).

4.1.1 Circular Doubly Linked List

Assertion

As mentioned already in Section 2.2, the main building block of Fibonacci heaps are circular doubly linked lists of heap nodes. The functional representation of a (circular) linked list is a simple functional list. This list contains the functional abstractions of the concrete values that are stored in the imperative list on the actual heap.

To represent a doubly linked list in Imperative HOL, list nodes need to be defined, which are represented with the following datatype:

```
datatype 'a dll = Cell (val: 'a) (following: "'a dll ref option")
                                 (previous: "'a dll ref option")
```

Note that the attribute storing a pointer to the following node is the first of the two pointer attributes. In the list node definition used in the augmented version of this approach (presented in Section 4.2.1), it will be the other way round.

The next step is to formulate the separation logic assertion for circular doubly linked lists. For that so called list segments are introduced, which describe a list from a certain node (not necessarily from the start) to the end of the list. This is the standard approach of verifying doubly linked lists in separation logic, which was already used by Reynolds in the original paper on separation logic [16]. The advantage of defining list segments is that they can be used to describe both circular and non-circular linked lists. For Fibonacci heaps only the former will be needed. A heap state that is a list segment is described by the following function that generates a separation logic assertion:

```
fun dll_seg :: "('a ⇒ 'b::heap ⇒ assn) ⇒ 'a list ⇒ 'b dll ref option ⇒
                'b dll ref option ⇒ 'b dll ref option ⇒ 'b dll ref option ⇒ assn"
  "dll_seg _ [] start start_pre end end_next = ↑(start = end_next ∧ start_pre = end)"
| "dll_seg R (x#xs) (Some start') start_pre  end end_next =
   ( ∃A start_next x'. start' ↦r Cell x' start_next start_pre
                     * R x x'
                     * dll_seg R xs  start_next (Some start')  end end_next)"
| "dll_seg _ _  _ _  _ _ = false"
```

The parameters `start` and `end` are the addresses of the first and last nodes of the segment respectively. The parameter `start_pre` is the address of the node that comes before the first node (so before `start`), and the parameter `end_next` is the address of the next node after the last node (`end`) of the list segment. The second parameter is the list of abstract list elements mentioned above. The first parameter is a relation that has to check whether a concrete value is a correct refinement of the given abstract value. It returns an assertion, which means that is not necessarily only responsible for the contents that are stored in the list directly, but it can, for example, also generate the assertion for a larger structure that is represented by a pointer stored in the list. To use the example of the Fibonacci tree: Here only the contents of the root node are directly stored in the list, but the refinement function `R` can also generate the assertion for the whole subtree which is rooted in that node. How this is implemented will be presented in the next subsection.

With this definition of list segments, it is now possible to generate the assertion for

circular doubly linked lists:

```
type_synonym 'a cdll = "'a dll ref option"

fun cdll :: "('a ⇒ 'b::heap ⇒ assn) ⇒ 'a list ⇒ 'b cdll ⇒ assn" where
  "cdll _ [] None = emp"
| "cdll R (x#xs) (Some p) = (∃A end. dll_seg R (x#xs) (Some p) end end (Some p))"
| "cdll _ _ _ = false"
```

The first two parameters are the same as for the segment assertion (the refinement relation and the abstract list), the third parameter is the object, that represents the whole list in a program, which in this case is just a pointer to the first element. The assertion for non-circular linked lists is almost the same, except that the values of the parameters `start_pre` and `end_next` would be set to `None` instead of `end` and `Some p` respectively.

Disadvantages of This List Representation

A crucial problem of this approach is that all of the addresses (except for the address of the list head) are existentially quantified. That means that the assertion does not state which values are stored at which addresses in the list. When using this assertion as pre- and postcondition of a list modifying function (for example prepend), it is not guaranteed that the function does not change the addresses of the list nodes. This was unintentionally shown by Stüwe, as in his presented version of the prepend function, the newly added element is placed in a new node stored at the position of the old list head node, while the modified old list head node is stored at the newly allocated address.

The reason for implementing it in that way is due to a problem with the representation of the heap in imperative-HOL. It does not support overwriting single fields of objects stored on the heap. For every change to an object, the whole object has to be overwritten. This means that the implementation where the addresses are preserved is actually not shorter than the one implemented by Stüwe.

The potential address change of certain elements is quite problematic. A key reason for using linked lists over dynamic arrays in imperative languages is the pointer/iterator stability, which is guaranteed as long as the pointed to element is not removed from the list. This directly affects the verification of Fibonacci heaps, because as described in Section 2.4 (and implemented in Section 4.3.4), the cut function used by decrease-key has to remove an element, that is identified by its address, from a list. This can only be verified if it is known which values are stored at which addresses in the list.

4.1.2 Fibonacci Heap

The chosen functional data structure that represents the Fibonacci trees in the heap is the so called rose tree [1], which models a tree where each node can have an arbitrary amount of children, which are stored in a list. Adapted to the needs of the Fibonacci heap implementation, it is defined as follows:

```
datatype ('a ::linorder) rosetree = Node (rank: nat) (marked: bool) (val: 'a)
                                         (children: "'a rosetree list")
```

The type annotation of the generic type ensures that the elements stored in the heap behave as expected, when they are compared (especially the transitivity of comparisons is essential for the correctness proof of the heap).

The functional abstraction of the heap itself is then defined as:

```
datatype 'a roseheap = Heap (count: nat) (nodes: "'a rosetree list")
```

Note that this heap contains a size attribute, which the other implementations presented in this thesis will not have.

The concrete imperative version of a heap node is the following:

```
datatype 'a rosetree_imp = Rose (cval: 'a) (sub: "'a rosetree_imp cdll")
type_synonym 'a fibtree_imp = "(nat × bool × 'a) rosetree_imp"
```

In this implementation the parent pointer is missing. The reason for this is, that the pointer is only needed by the handle-based operations, which as mentioned were not implemented in the presented thesis. The pointers to the previous and next element are taken care of by the list.

The concrete version of the heap object, which represents the whole heap, is defined as follows:

```
type_synonym 'a fibheap_imp = "nat × 'a fibtree_imp cdll"
```

The function creating the assertion from the functional representation is presented next. It uses the recursive power of the refinement relation:

```
fun fibtree_imp :: "'a::{linorder, heap} rosetree ⇒ 'a fibtree_imp ⇒ assn" where
"fibtree_imp (Node r m v ts) (Rose (r', m', v') c')
= cdll fibtree_imp ts c' * ↑(r' = r ∧ m' = m ∧ v' = v)"

definition fibheap_imp :: "'a::{heap,linorder} roseheap ⇒ 'a fibheap_imp ⇒ assn"
where
"fibheap_imp h hi =
     (case (h, hi) of ((Heap n ts), (n', p)) ⇒ cdll fibtree_imp ts p * ↑(n = n'))"
```

This defines the basic structural invariant of the heap, but additional invariants are needed to be able to prove the correctness and the running time bounds of the heap. For the correctness only the order of elements needs to be ensured and for the running time the rank of the nodes is restricted as described in Section 2.3. First the invariants for a single tree are defined, which are then used for the heap invariant:

```
inductive fibtree :: "'a::linorder rosetree ⇒ bool" where
"∀ i ∈ {0..< length ts}. let t = nth ts i in
       i ≤ rank t + of_bool (marked t) ⟹ ∀ t ∈ set ts. fibtree t ⟹
       r = length ts ⟹ fibtree (Node r _ _ ts)"

fun min_tree :: "'a rosetree ⇒ bool" where
"min_tree (Node _ _ v ts) ⟷ (∀ t ∈ set ts. min_tree t ∧ v ≤ val t)"
```

```
fun min_heap :: "'a roseheap ⇒ bool" where
"min_heap (Heap _ ts) ⟷ (∀t∈set ts. min_tree t)"

fun fibheap :: "'a roseheap ⇒ bool" where
"fibheap (Heap c []) ⟷ c = 0" |
"fibheap (Heap c ts) ⟷ c = size (mset_heap_list ts)
                         ∧ (∀ t ∈ set ts. val (hd ts) ≤ val t ∧ fibtree t)"

definition invar :: "'a roseheap ⇒ bool" where
"invar h ⟷ fibheap h ∧ min_heap h"
```

As shown by Stüwe in his thesis, with these definitions it is possible to prove the functional correctness of the non-handle based heap operations, but as mentioned for the verification of decrease-key this approach is not sufficient. The augmentations needed for that verification are presented in the following section.

4.2 Augmented Functional Approach

The main problem with the previous approach, as detailed at the end of Section 4.1.1, is that the invariants do not ensure pointer stability, which is essential for the handle-based operations. So the first idea for verifying decrease-key was to reuse Stüwe's approach by augmenting it with explicitly stored addresses in the functional representation of the heap. This approach ultimately did not work very well due to some fundamental problems that seem not to be solvable. For that reason another approach was developed, which is explained in Section 4.3, but still it is quite interesting to see what the introduction of explicit addresses does to the assertions and where this approach falls short.

As in the previous chapter, the separation logic assertion for the circular doubly linked list is described first, which is then used to verify the list functions needed for the Fibonacci heap implementation (Section 4.2.1). This is followed by the description of the assertion for the Fibonacci Heap, which is accompanied by an in depth description of the problems that the assertion has, when it is used for the verification of decrease-key (Section 4.2.2). The assertion for the circular doubly linked list contains a major change (in addition to the incorporated explicit addresses) when compared to the assertion described in the previous chapter: The refinement relation, that relates a concrete imperative value to its functional abstraction, was removed and replaced with a function with much less expressive power. The reasons for this change are explained after the description of the Fibonacci heap assertion (Section 4.2.3), as the explanation is somewhat complicated and it is not required for the understanding of the changed heap assertion. This chapter ends with a summary of the major takeaways from this approach (Section 4.2.4).

4.2.1 Circular Doubly Linked List

This subsection begins with an introduction to the modified list assertion. Then the list functions that are needed for the verification of decrease-key are verified, and in the end potential problems of this approach are discussed.

Separation Logic Assertion

To be able to guarantee the pointer stability of the linked list, the abstract, functional list representation was modified so it additionally stores the address of the corresponding list node for each value. This address-value-pair is stored in its own datatype, which is defined as follows:

```
datatype ('conc, 'abs) dll_abs = Dll_Abs_Node (dll_abs_addr: "'conc dll ref")
                                              (dll_abs_val: 'abs)
```

Note that this definition has two type parameters: the first is the type of the abstract representation of the concrete values that are stored on the heap, and the second type is the type for the concrete values themselves. The latter is needed for the type parameter of the reference type.

The type of the concrete nodes stored on the heap, is the following:

```
datatype 'a dll = Dll_Node (dll_val: 'a) (dll_prev: "'a dll ref option")
                                         (dll_next: "'a dll ref option")
```

A list segment can now be defined as follows (using the presented helper function):

```
abbreviation node_to_ptr :: "('a::heap, 'b) dll_abs ⇒ 'a dll ref option" where
   "node_to_ptr x ≡ Some (dll_abs_addr x)"

abbreviation dll_seg_single where
"dll_seg_single F x pre next ≡
                     (dll_abs_addr x) ↦r (Dll_Node (F (dll_abs_val x)) pre next)"

fun dll_seg :: "('a ⇒ 'b::heap) ⇒ ('b, 'a) dll_abs list ⇒ 'b dll ref option ⇒
                'b dll ref option ⇒ assn" where
  "dll_seg _ [] _ _ = emp"
| "dll_seg F [x] pre end_next = dll_seg_single F x pre end_next"
| "dll_seg F (x#y#xs) pre end_next = dll_seg_single F x pre (node_to_ptr y)
                                     * dll_seg F (y#xs) (node_to_ptr x) end_next"
```

As can be seen the refinement relation was replaced with a function, that, given an abstract representation of a value, returns the concrete representation of that value, that is stored on the heap. This means that the list assertion is now only responsible for its direct contents and not for potentially complex data structures, which are represented by the abstract representation of the stored value. As mentioned the reasons for this change are detailed in Section 4.2.3. Together with the introduction of addresses into the abstract representation of the list, this completely eliminated the need for existential quantifiers.

There are two lemmas connected to these segments which are needed for later proofs, which state that a list segment can be split apart (equivalent lemmas can already be found in Stüwe's thesis [17]):

```
lemma dll_seg_Cons:
   "dll_seg F (x # xs) pre end_next =
    dll_seg_single F x pre (first_or_default_ptr end_next xs)
    * dll_seg F xs (node_to_ptr x) end_next"
```

```
lemma dll_seg_append:
   "dll_seg F (xs @ ys) pre end_next =
    dll_seg F xs pre (first_or_default_ptr end_next ys)
    * dll_seg F ys (last_or_default_ptr pre xs) end_next"
```

These lemmas are used to extract a points-to assertion from the list assertion, which is required so that the automation can apply the Hoare triples defined for reading from and writing to a certain address (as described in Section 3.3). The used functions `first_or_default_ptr` and `last_or_default_ptr` return either the first argument, if the given list is empty, or the pointer with the address of the first/last element of the given list. These are used to reduce the amount of cases that need to be considered.

With the defined list segment assertion, circular doubly linked lists can be defined:

```
fun hd_option :: "'a list ⇒ 'a option" where
  "hd_option [] = None"
| "hd_option (x#xs) = Some x"

abbreviation cdll_ptr :: "('conc::heap, 'abs) dll_abs list ⇒ 'conc cdll" where
"cdll_ptr l ≡ hd_option (map dll_abs_addr l)"

definition cdll_assn' :: "('abs ⇒ 'conc::heap) ⇒
                          ('conc, 'abs) dll_abs list ⇒ assn" where
"cdll_assn' F l ≡ dll_seg F l (ptr_to_node (last l)) (ptr_to_node (hd l))"

fun cdll_assn :: "('abs ⇒ 'conc::heap) ⇒ ('conc, 'abs) dll_abs list ⇒
                  'conc cdll  ⇒ assn" where
"cdll_assn F l ptr = ↑(ptr = cdll_ptr l) * cdll_assn' F l"
```

A useful lemma, when explicitly storing the addresses in the abstraction, states that the used addresses must be distinct in the list. For that it is first shown that the list segment actually occupies all addresses that occur in the abstract list given as argument. Then the main lemma, about the distinctness of the addresses inside of the abstract list, follows by induction on the list length. This lemma will not be needed for the correctness proofs of the list functions, but it is useful for the Fibonacci heap verification.

```
lemma dll_seg_star_sngr_false:
assumes "x ∈ (dll_abs_addr ` set l)"
shows "x ↦r y * dll_seg F l pre end_next = false"

corollary dll_seg_distinct:
      "¬ distinct (map dll_abs_addr l) ⇒ dll_seg F l pre end_next = false"
```

Verification

With this representation it is now possible to verify some basic list operations. These can also already be found in Stüwe's thesis [17], though the implementation may differ.

The first function to be proven creates an empty list, which simply returns the null-pointer:

```
definition cdll_empty :: "'a::heap cdll Heap" where
"cdll_empty ≡ return None"

lemma cdll_empty_rule: "<emp> cdll_empty <cdll_assn F []>"
```

The next function creates a list with a single element. The concrete element is supplied by the function call, but the abstract element that represents the concrete element must be provided by the user during the proof. It might be an interesting idea to require the used function, that maps an abstract representation to the concrete element, to be bijective, so that the abstract element does not need to be manually provided, but can directly be generated from the concrete element.

```
definition cdll_singleton :: "'a::heap ⇒ 'a cdll Heap" where
"cdll_singleton x = do {
   ptr ← ref (Dll_Node x None None);
   ptr := Dll_Node x (Some ptr) (Some ptr);
   return (Some ptr)
}"

lemma cdll_singleton_rule:
assumes "F x_abs = x_conc"
shows
  "<emp> cdll_singleton x_conc <λptr. cdll_assn F [Dll_Abs_Node (the ptr) x_abs] ptr>"
```

The above shown Hoare triple makes use of an important lemma, that is also relevant for all the other Hoare triples shown in thesis. It states the equivalence of necessary conditions for a Hoare triple and pure assertions in the precondition:

```
lemma norm_pre_pure_iff: "<P*↑b> f <Q> ⟷ (b ⟶ <P> f <Q>)"
```

The advantage of pulling pure assertions into the necessary conditions for the Hoare triple is mainly that it simplifies proofs as the statements can be directly used in the proofs, without first needing to show that one may assume that the precondition is not `false`. This does not interfere with the Hoare triple automation, as it also makes the same modification to preconditions.

The next function that is proven prepends a new element to the list. Its implementation can be seen in Listing 4.1 This differs from the implementation already found in Stüwe's work [17], as this version of prepend preserves the addresses of the already existing list nodes. The corresponding Hoare triple is defined as follows:

```
lemma cdll_prepend_rule:
assumes "F x_abs = x_conc"
shows
   "<cdll_assn F l ptr>
    cdll_prepend ptr x_conc
    <λptr. cdll_assn F ((Dll_Abs_Node (the ptr) x_abs)#l) ptr>"
```

Until now the proofs of the Hoare triples were quite simple. As mentioned, the most difficult part is that for reads and writes to addresses one needs to expose the points-to assertions of these addresses. The separating conjunction makes this more complicated as overlapping addresses are not allowed. This means that for prepend one needs to

```
fun cdll_prepend :: 'a::heap cdll ⇒ 'a ⇒ 'a cdll Heap where
  cdll_prepend None new_el = do {
     new_addr ← ref (Dll_Node new_el None None);
     new_addr :=  Dll_Node new_el (Some new_addr) (Some new_addr);
     return (Some new_addr)
  }
| cdll_prepend (Some first_addr) new_el = do {
     first_el ← !first_addr;
     let last_addr = (the (dll_prev first_el));
     new_addr ← ref (Dll_Node new_el (Some last_addr) (Some first_addr));

     first_addr := Dll_Node (dll_val first_el) (Some new_addr) (dll_next first_el);
     last_el ← !last_addr;
     last_addr := Dll_Node (dll_val last_el) (dll_prev last_el) (Some new_addr);
     return (Some new_addr)
  }
```

Listing 4.1: The implementation the prepend function

differentiate whether the list parameter contains only one or more than one element. In the former case, the first and last node of the list, which are changed during prepend, are the same. In the end there are three cases that need to be considered for the verification as the list can also be empty. For the implementation of append, which concatenates two lists and which is presented in Listing 4.2, this is slightly more complicated. Here the first and last element of both lists are changed (if they are not empty), which results in $3 \cdot 3 = 9$ cases. The actual proofs of each case are again relatively simple. For all of these proofs (for prepend and append) only the mentioned split rules `dll_seg_Cons` and `dll_seg_append` for the list segment are needed. The Hoare triple can be defined as follows:

```
lemma cdll_append_rule:
   "<cdll_assn F xs ptr1 * cdll_assn F ys ptr2>
    cdll_append_cdll ptr1 ptr2
    <cdll_assn F (xs@ys)>"
```

The last function to be proven, unlinks a node from a list. The removed node is subsequently made into a singleton list. The implementation of this function is shown in Listing 4.3. As the node to be removed is only identified by its address, the explicit addresses in the invariant are required here, otherwise the proof of the Hoare triple would not be possible. This is the only list function not already found in Stüwe's thesis [17].

For the proofs it is again necessary to differentiate the cases where two changed nodes are actually the same. Here the changed nodes are the previous and next node of the node that should be removed. An important thing to note for the proofs is that the element to be removed might be the first element. In that case it is necessary to change the pointer to the first element of the list, which is returned by the function. Apart from the additional cases due to the potential list pointer change, the proofs for unlink are not harder than the ones for prepend and append. The proven Hoare triple is defined as follows:

```
fun cdll_append_cdll :: "'a::heap cdll ⇒ 'a cdll ⇒ 'a cdll Heap" where
  "cdll_append_cdll None ptr = return ptr"
| "cdll_append_cdll ptr None = return ptr"
| "cdll_append_cdll (Some addr1) (Some addr2) = do {
      first_el1 ← !addr1;
      first_el2 ← !addr2;
      let last_addr1 = (the (dll_prev first_el1));
      let last_addr2 = (the (dll_prev first_el2));

      addr1 := Dll_Node (dll_val first_el1) (Some last_addr2) (dll_next first_el1);
      addr2 := Dll_Node (dll_val first_el2) (Some last_addr1) (dll_next first_el2);
      last_el1 ← !last_addr1;
      last_el2 ← !last_addr2;

      last_addr1 := Dll_Node (dll_val last_el1) (dll_prev last_el1) (Some addr2);
      last_addr2 := Dll_Node (dll_val last_el2) (dll_prev last_el2) (Some addr1);
      return (Some addr1)
}"
```

Listing 4.2: The implementation of the append function, that concatenates two lists

```
lemma cdll_unlink_rule:
assumes "Dll_Abs_Node addr v ∈ set l"
shows
   "<cdll_assn R l list_ptr >
    cdll_unlink list_ptr addr
    <λptr. cdll_assn R [Dll_Abs_Node addr v] (Some addr)
           * cdll_assn R (remove1 (Dll_Abs_Node addr v) l) ptr>"
```

Disadvantages of This List Representation

The used abstract list representation is actually somewhat problematic. For data structures like the Fibonacci heap, that have nodes that also function as list nodes, one normally implements a monolithic node, which contains all pointers and which also implements the list functionality all by itself. This is not a problem when the functions are just implemented and not verified, since the implementation of the list functionality is not very difficult. For the verification it would mean that the list functionality would need to be reverified for every data structure, whose nodes are also list nodes.

One of the reasons for not reusing list node implementations for the implementation of other data structures, that require some kind of list functionality, is that in object oriented programming the list implementations generally do not expose the list nodes through their public interface. The node implementations are normally hidden as private implementation details. While this means that the used approach, of reusing a list node implementation, does not model the reality of many programming languages, this no direct argument against using this approach in Imperative HOL.

There is a bigger problem though, that is also very relevant for Imperative HOL. It

```
definition cdll_unlink :: "'a::heap cdll ⇒ 'a dll ref ⇒ 'a cdll Heap" where
"cdll_unlink list_ptr del_addr = do {
   let list_addr = the list_ptr;
   del_el ← !del_addr;
   if dll_next del_el = (Some del_addr) then
      return None
   else do {
      let next_addr = the (dll_next del_el);
      let prev_addr = the (dll_prev del_el);

      next_el ← !next_addr;
      next_addr := Dll_Node (dll_val next_el) (dll_prev del_el) (dll_next next_el);
      prev_el ← !prev_addr;
      prev_addr := Dll_Node (dll_val prev_el) (dll_prev prev_el) (dll_next del_el);
      del_addr := Dll_Node (dll_val del_el) (Some del_addr) (Some del_addr);
      if list_addr = del_addr then
         return (Some next_addr)
      else
         return list_ptr
   }
}"
```

Listing 4.3: The implementation of the unlink function, which removes the given element from the list and turns it into a singleton list.

occurs, when an overlaid data structure is considered, where a value is stored directly in at least two lists at once. In the presented approach, this would be implemented with an outer list node containing an inner list node that contains the value. This does not work with the used list abstraction, as here each list node must have its own address identity (which appears in the abstract representation of the node). In Imperative HOL only the whole object stored on the heap has an address identity, which here corresponds to the outer list node.

A concept from low level programming, that could alleviate both problems, are so called intrusive lists [2]. Here the list is not responsible for allocating a list node, but the user of the list needs to provide the complete node with the required list pointers. If the intrusive list implementation allows the user to provide the getters and setters for the list pointers, then one could implement a monolithic node, that is in two lists at the same time, by using two different intrusive lists to implement the list functionality. This does not seem to be doable in Isabelle/HOL. To verify that an intrusive list implementation works correctly, one would need to require, that the setter of the list pointers provided by the user only changes the list pointer and nothing else. This concept does not seem to be expressible in Isabelle/HOL, but we do not know enough about Isabelle/HOL to be able to state this with certainty.

4.2.2 Fibonacci Heap

This section starts with an overview over the definition of the Fibonacci heap assertion, and then the problems of this approach are presented, which are the reason why decrease-key was not verified with this approach.

Separation Logic Assertion

With the definition of lists it is now possible to define the Fibonacci heap itself. As in Stüwe's approach (Section 4.1), a Fibonacci heap node is a list node that contains a Fibonacci tree node. The latter looks as follows:

```
datatype 'a fib_tree = Fib_Node
(fib_rank: nat) (fib_marked: bool) (fib_val: 'a)
(fib_children: "'a fib_tree cdll") (fib_parent: "'a fib_tree dll ref option")
```

The big difference to the previous approach is the introduction of the parent pointer. This is crucial for the implementation of decrease-key, as its subroutine cascading-cut traverses the heap along these pointers.

The abstraction of a Fibonacci tree is also a rose tree;

```
datatype 'a fib_tree' = Fib_Node'
(fib'_rank: nat) (fib'_marked: bool) (fib'_val: 'a)
(fib'_children: "('a fib_tree, 'a fib_tree') dll_abs list")
```

The pointer to the parent node is not stored in the abstraction, as it can be generated from the recursive structure. In the recursive function that generates the assertion from the abstract heap representation, which will be shown below, each parent node is responsible for passing its own address to its children. The abstraction also does not contain its own address, as this handled by the list node abstraction. A full Fibonacci heap node has the following type:

```
'a::heap fib_tree dll
```

As mentioned, this heap implementation does not have an attribute that stores the number of elements, as it is not necessary for the implementation of the heap functions. With that in mind, the Fibonacci heap itself has the type:

```
'a::heap fib_tree cdll
```

The abstract representation of the heap has the following type:

```
('a::heap fib_tree, 'a fib_tree') dll_abs list
```

Since the list assertion is only responsible for the values directly stored in the list, as mentioned in the previous subsection, the invariant used by Stüwe [17] cannot be reused.

First an assertion generating function for lists of Fibonacci trees is defined. It requires the function that given the pointer to the parent node, converts an abstract node into a concrete one, that is used by the list assertion:

```
fun fib_tree_node :: "'a::heap fib_tree dll ref option ⇒ 'a fib_tree' ⇒ 'a fib_tree"
where
"fib_tree_node prnt (Fib_Node' r m v chldrn) = Fib_Node r m v (cdll_ptr chldrn) prnt"

function fib_list_assn :: "'a::heap fib_tree dll ref option ⇒
                           ('a fib_tree, 'a fib_tree') dll_abs list ⇒ assn" where
"fib_list_inv prnt l = cdll_assn' (fib_tree_node prnt) l
                       * prod_mset ((λt. fib_list_inv (node_to_ptr t)
                                   (fib'_children (dll_abs_val t))) `# (mset l))"
```

In the function `fib_list_assn` the left side of the separating conjunction is responsible for generating the assertion for the current list, with the given pointer to the parent node. The right side recursively generates the assertions for all children. The function `prod_mset` is a separating product over all elements in the given multiset. In the shown definition this multiset contains all Fibonacci tree list assertions for all of the child lists of the nodes in the current list. Using a multiset instead of a normal set ensures, that if there are two elements in the current list that are the completely equal, then the resulting assertion would be equivalent to `false`.

As the root list of the Fibonacci heap is a list of trees that do not have a parent, which is represented by the nullpointer (equivalent to `None` in Imperative HOL), the assertion for the whole heap is the following:

```
definition fib_heap_inv :: "('a::heap fib_tree, 'a fib_tree') dll_abs list ⇒
                            'a fib_tree dll ref option ⇒ assn" where
"fib_heap_inv l ptr = ↑(cdll_ptr l = ptr) * fib_list_inv None l"
```

For this approach the invariants for the abstract heap representation were not defined, as this approach was abandoned before they became relevant. The reasons for that are explained in the next section.

Problems for the Decrease-Key Verification

When one tries to use this approach for the verification of decrease-key, a major problem becomes visible. As part of verifying decrease-key, it is necessary to verify a single cut operation. Here a node, indicated by its address, needs to be removed from the child list of its parent (an element in the root list cannot be cut). In the functional representation this translates to removing the cut element from the child list stored in the parent node (and extracting the content of that element, which contains the whole subtree that is rooted in it). So before being able to do the cut on the abstract data structure, it is necessary to retrieve the parent address at some point. Since the abstract nodes do not contain the parent pointer and because the node to be cut and its parent are buried somewhere in the heap structure, it is necessary to search the whole tree for the parent node. To at least somewhat direct the search for the nodes, the presented implementation of the search checks whether the child address is actually contained in a sub tree, before going down further into it. This is done with the help of the function `fib_list_addr_set` that returns the set of addresses contained in a list of Fibonacci trees. The function `get_first` uses this function to direct the search. It is called that way, because there is

the possibility that there is more than one abstract node with same address. In that case the generated assertion for that abstract heap would be `false`, but the function retrieving the parent cannot directly make use of this information.

```
function fib_list_addr_set :: "('a::heap fib_tree, 'a fib_tree') dll_abs list ⇒
                               'a fib_tree dll ref set" where
"fib_list_addr_set l = (dll_abs_addr ` set l)
                ∪ (⋃ x ∈ set l. fib_list_addr_set (fib'_children (dll_abs_val x)))"

fun get_first where
  "get_first a b [] = undefined"
| "get_first addr F (x#xs) =
                (if addr ∈ fib_list_addr_set (fib'_children (dll_abs_val x))
                 then F x else get_first addr F xs)"

function get_parent_addr ::
      "'a::heap fib_tree dll ref ⇒ 'a fib_tree dll ref option ⇒
       ('a fib_tree, 'a fib_tree') dll_abs list ⇒ 'a fib_tree dll ref option" where
"get_parent_addr addr prnt l = (let rec_call =
   (λt. get_parent_addr addr (node_to_ptr t) (fib'_children (dll_abs_val t))) in
      (if addr ∈ dll_abs_addr ` (set l) then prnt else get_first addr rec_call l))"
```

Theoretically this function would not be necessary, as the abstract nodes could simply store the address of the parent node, but then the abstract representation would contain redundant information (as the parent relation is already defined by the recursive structure of the abstract heap). Additionally, the problem that relatively simple queries/operations on the heap have to be implemented by searching the whole heap for the node that should be returned/changed, will also occur for other functions that are necessary for the verification. This can be seen in the next few paragraphs.

As mentioned in Section 3.3 the Hoare triple proof automation requires exposed points-to assertions to be able to deal with reads and writes to these addresses. For that to happen, it is necessary to split the heap invariant. Such a split could look like this:

```
"fib_list_inv None l = fib_list_inv_hole hole_addr None l
                       * dll_seg_single [...] * fib_list_inv [...]"
```

The first part describes the heap with exception for the subtree rooted in the node to be changed. The second part describes only the node to be changed and the third part describes its children nodes and the subtrees rooted in them. This split is illustrated in Figure 4.1 (with corresponding colors). There are two functions required by this split. The first one is shown: it is the function generating the assertion without the subtree that is rooted in the node that is split from the heap assertion. The second function retrieves this subtree, so that the two other parts of the split heap assertion can be defined. Note that only fetching the abstract subtree is actually not enough, because the assertion for the root node of that subtree requires knowledge about the previous, next and parent pointers of the node, which are not directly stored in the abstract nodes, but they are generated from the surrounding structure (these are the orange pointers in Figure 4.1). So in addition to the abstract subtree, the pointer interface with the rest of the heap of the root node of the subtree has to be retrieved. This is achieved, by both retrieving

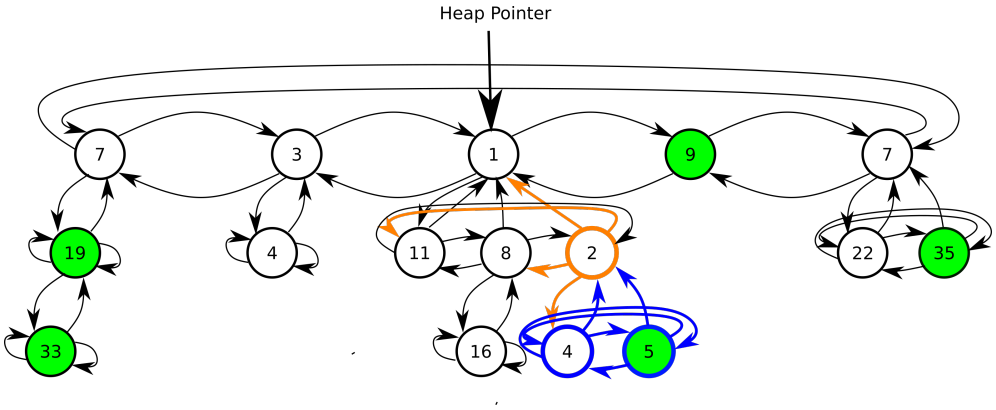

Figure 4.1: A visualization of the split heap, where "2" is the node for which the points-to assertion should be exposed. The different colors of the nodes and the pointers indicate which part of the split assertion describes these nodes/pointers. The parts are the heap without the node that should be split, the node itself and its children (together with their subtrees).

the abstract node and also the concrete node as it is stored in the heap, which contains all of the needed pointers. The first mentioned function that creates the invariant for the heap without the split subtree that is rooted in the node with address `hole_addr` is implemented as follows:

```
fun dll_seg_hole where
  "dll_seg_hole hole_addr _ [] _ _ = emp"
| "dll_seg_hole hole_addr F [x] pre end_next =
       (if dll_abs_addr x = hole_addr then emp else dll_seg_single F x pre end_next)"
| "dll_seg_hole hole_addr F (x#y#xs) pre end_next =
       (if dll_abs_addr x = hole_addr then dll_seg F (y#xs) (node_to_ptr x) end_next
        else dll_seg_single F x pre (node_to_ptr y)
             * dll_seg_hole hole_addr F (y#xs) (node_to_ptr x) end_next)"

abbreviation cdll_assn'_hole where
"cdll_assn'_hole addr F l ≡
                  dll_seg_hole addr F l (node_to_ptr (last l)) (node_to_ptr (hd l))"

function fib_list_inv_hole
   :: "'a::heap fib_tree dll ref ⇒ 'a fib_tree dll ref option ⇒
       ('a fib_tree, 'a fib_tree') dll_abs list ⇒ assn" where
"fib_list_inv_hole hole prnt l =
  cdll_assn'_hole hole (fib_tree_node prnt) l *
  (Π x ∈ {x |x. x ∈ set l ∧ dll_abs_addr x ≠ hole}.
          fib_list_inv_hole hole (node_to_ptr x) (fib'_children (dll_abs_val x))))"
```

The recursive structure is relatively similar to the get parent function, but it now needs to traverse the whole heap (except for the subtree rooted node indicated by the given address). The function returning the abstract subtree (and the interface of the root node of said tree) also has a very similar recursive structure, which is why its not explicitly

shown here.

There is still another auxiliary function that is missing. With this split it is now possible for the proof automation to transform the precondition of cut defined with the presented heap assertion (which is not explicitly stated in this thesis) according to the program. The key part that is missing is the definition of the postcondition. For its definition, one needs to remove the cut child from the children list of its parent (as mentioned above). The function that replaces the child list of the parent, with the modified one, again has to search the whole tree for the parent address, to able to do the modification. This means its structure is also very similar to the other mentioned functions, which is why it is also not shown.

So even before a single lemma is proven, this approach is already quite verbose. For proving lemmas there are two additional problems that become visible. Firstly, the abstract structure itself does not exclude that there are multiple abstract nodes with the same address (and possibly different contents) stored in the abstract heap. This will make the proofs more complex. For example, it has to be proven that when one obtains an address of a node with `get_parent_addr child_addr x l` (which guarantees that there is a node with the returned address that contains a node with address `child_addr` in its direct child list), that this also holds for the node that is returned by the function that returns the abstract node and its interface when called with the parent address. If there are two nodes with the same address, the latter function might return the other node, that does not necessarily contain the child in its child list. Secondly, many of the important lemmas which use the above mentioned functions need to be proven by induction over the recursive structure of these functions. This turned out to be much more complicated than anticipated (partly due to the just mentioned problem).

While there are probably some points where this approach can be improved, (for example, the function that retrieves the parent address might not be required), it looks like there are some fundamental issues with this approach that are not amendable, which is why it was not further pursued. Probably the main reason why this approach does not work very well is that the traversal of cascading cut (from an arbitrary node up to the root) does not follow the recursive structure of the heap definition (from the root down to all nodes). Bringing the heap structure into a form that follows the cascading-cut traversal for its verification does not seem to work, as defining a heap with an arbitrary node as the "root" seems to be very complicated and unnatural. If one wanted to define a recursive assertion generating function that starts at an arbitrary node and works itself along the parent pointers, generating the assertion for the list of Fibonacci trees in which the parent of the starting node is located is quite problematic. The reason for this is that the subtree of the parent contains the root, which was already described by the assertion generating function, while the subtrees of siblings of the parents do not contain any nodes already described. This also holds for the grand-parent and so on as well.

4.2.3 Disadvantages of a Recursive Refinement Relation

The approach presented in the previous section did not use a potentially recursive refinement relation, as was done in Stüwe's thesis [17] (see Section 4.1). The main goal of the refinement relation is that it checks, whether the abstraction given as the first parameter correctly represents the concrete node given as the second parameter, and it returns an assertion. Since it returns an assertion it can also be used to verify not only the contents directly stored in the list node, but also the whole object that is represented by the contents of the abstract node. For the example of the heap this means, that it can also be used to generate the assertion for the whole subtree that is rooted in a node, since the abstract version of each subtree is stored in the abstract version of its root node.

The first problem with this approach is a conceptual one, as it violates a fundamental programming principle: the separation of concerns. The used refinement relation is directly responsible for checking that the abstract node correctly abstracts the value stored in the concrete list node stored on the heap and then it also potentially generates the assertion for a whole data structure (in the case of the heap the subtree rooted in that node).

So the first step is to break up this relation into two functions. The first one generates the concrete node that is stored on the heap from the given abstract node. The second one generates an assertion from the given abstract value, which can be used to generate the assertion for the subtree in the heap example. Using functions instead of a relation eliminates an existential quantifier, because when using the relation, it was necessary, to state that there exists a concrete node that is represented by the abstract one, while the first function now simply generates the concrete node from the abstract one. Since a function is used to generate the concrete node stored on the heap, it is required that each concrete node is fully described by the abstract representation. In cases where the invariant does not need to describe the whole state of a data structure (for example the user of the Fibonacci heap functions does not need to know that exact state the heap is in), we believe it might be a good idea for the proofs of the Hoare triples to assume that one knows the exact state of a data structure and then existentially quantify the part of the state the user does not need to track. This was done for the Fibonacci heap verification presented in the next section.

Even with this separation of concerns, there is still a big problem with this approach and it becomes visible when applied to the Fibonacci heap. Here the value of each node depends on the address of its parent node (as the value of the parent pointer depends on that address). In the recursive definition of the heap assertion in the previous section, each node is responsible for passing its address to its children in the recursive call. If the recursive refinement function is used, which generates the assertion for the children, it is responsible for forwarding that address. This does not seem so problematic, as this can simply be implemented by defining the refinement function not as a relation between an abstract element and a concrete element, but between an abstract list node (which is a pair of the address and the abstract element) and the concrete element. The list segment assertions are then defined as follows:

```
abbreviation dll_seg_single where
"dll_seg_single R F x pre next ≡
                  R x * (dll_abs_addr x) ↦r (dll_Cell (F (dll_abs_val x)) pre next)"

fun dll_seg :: "(('conc::heap, 'abs) dll_abs ⇒ assn) ⇒ ('abs ⇒ 'conc) ⇒
                ('conc, 'abs) dll_abs list ⇒ 'conc dll ref option ⇒
                'conc dll ref option ⇒ assn" where
  "dll_seg R F [] pre end_next = emp"
| "dll_seg R F [x] pre end_next = dll_seg_single R F x pre end_next"
| "dll_seg R F (x#y#xs) pre end_next = dll_seg_single R F x pre (node_to_ptr y)
                                       * dll_seg R F (y#xs) (node_to_ptr x) end_next"

abbreviation cdll_assn' where
"cdll_assn' R F l ≡ dll_seg R F l (node_to_ptr (last l)) (node_to_ptr (hd l))"
```

The problem with this approach appears when a new list element is created. Similar to the Hoare triples defined in Section 4.2.1 for the list singleton and prepend functions, the precondition needs to state that the refinement relation holds for the newly added element and its abstraction. The abstraction has to be provided manually by the user of these Hoare triples. As was just mentioned, this expression depends on the address of the node to be created. This address is unknown before the element is allocated. Therefore it has to be shown that the result of the function that generates the recursive assertion for the node is independent of that address. So a changed Hoare triples for the `cdll_prepend` function could look like this:

```
lemma cdll_prepend_rule:
assumes
   "x_conc = F x_abs"
   "∀ addr. R (dll_abs_Cell x_abs addr) = asn"
shows
   "<cdll_assn R F l ptr * asn>
    cdll_prepend x_conc ptr
    <λptr. cdll_assn R F ((dll_abs_Cell x_abs (the ptr))#l) ptr>"
```

This Hoare triple requires that the refinment relation generates the same assertion `asn` for all possible adresses of the newly allocated node. In the case of the Fibonacci heap, the fixed assertion `asn` would actually be equivalent to `emp`, as a new heap node does not have any children.

This construction in the Hoare triple is somewhat unintuitive, so it was decided to scrap the recursive aspect completely. That is why in the presented approach, a list assertion is only responsible for the actual contents of the list and nothing else.

4.2.4 Summary

While the augmentation of Stüwe's approach was easy to implement, using it for verification did not work out as expected. The main problem was that decrease-key can start at an arbitrary node in the heap and from there continues along the parent pointers, which does not match the recursive definition of the abstract heap structure. The list function `cdll_unlink` also changes an arbitrary element, only indicated by its address, in the list

(it is removed), but the list assertion is much easier to split than the heap assertion, which is why it is much easier to verify. This start at an arbitrary node is something that does not occur in many complex data structures, like the Fibonacci heap, which is why there are also seemingly hardly any (or even no) other papers, that make use of separation logic to verify such data structures.

4.3 Map-Based Approach

As mentioned in the previous section, the proof of the handle based operations with a simple functional heap representation poses some problems. So the question is, whether there is a better approach to define the Fibonacci heap assertion. After contemplating different ideas (one of which, that was only shortly considered, is discussed in Section 4.4), the final choice was to use as little abstraction as possible by choosing the most basic representation that works for any imperative data structure: a map of addresses to the objects that make up said data structure. This representation has the big advantage that it most closely represents the actual reality. For the Fibonacci heap that means it is very simple to traverse the heap in any arbitrary direction, because there is now no predetermined structure that favors a single direction, as is the case for the functional representation. The big disadvantage is that the structural and logical invariants now have to be defined for an inherently unstructured representation.

As the heap representation used in Imperative HOL (presented in Section 3.2) is itself a map from addresses to objects, there is the possibility of directly applying the invariants to the heap itself. This was not done as the separation logic framework, does not support the direct access of the heap in the pre- and postconditions of the Hoare triples. Not using the separation logic framework was also not an option, as the program verification framework of Imperative HOL itself is relatively basic and probably not suited for complex data structures. In addition, if one wants to use abstract representations of the objects for the verification, which was done in the approach presented in this section, directly working on the heap does not work.

There are two main ways to represent the abstract map in Isabelle/HOL: one is to use the existing map datatype, which is maps key to options of values, and the other is to use a pair of a key set, denoting the domain of the map, and a function from keys to values. The main advantage of the second approach is that it directly stores the values and not value options, which means querying the values is slightly easier. The main disadvantage of the second approach is that there are infinitely many ways to denote a certain state of the map, as only the values inside of the domain are relevant for the logical state of the map. The values outside of the domain are undefined. For the map datatype all entries outside of the domain are set to `None`. In the end the second approach was chosen due to the advantage mentioned. Conceptually using the dedicated map datatype is definitely the better choice, but is not clear whether it will actually lead to nicer proofs, which is probably something that should be explored.

Before beginning with the verification of the list and the heap with the new approach,

the concept of footprints for separation logic assertions is formalized (Section 4.3.1). It turned out to be quite helpful for the proofs. A footprint is an address set for which an assertion can hold on any heap. After that it is shown how the already presented list assertion has to be changed, so that it works with the new approach (Section 4.3.2). This assertion is then used to define the Fibonacci heap assertion (Section 4.3.3). The basic assertion needs to be extended with Boolean predicates on the map which ensure that the abstract map represents a valid Fibonacci heap. These Boolean invariants include a part that deals with the order and rank of elements and a part that enforces the structure, which in the previous approach was given by the functional rose tree structure. Then the implementation of decrease-key and its used subroutines is shown (Section 4.3.4), before they are verified. First comes the proof of cut (Section 4.3.5), then the proof of cascading-cut (Section 4.3.6) and the finally the proof of decrease-key itself (Section 4.3.7). This is followed by the implementation and verification of four additional heap functions: make-heap, heap-singleton (which creates a new heap with a single element), get-min and heap-union (Section 4.3.8). In the end the main takeaways from this approach are summarized (Section 4.3.9).

4.3.1 Assertion Footprint

Pretty early in the development of the verification process, it became clear that it might be useful to look at the set of addresses a data structure currently occupies. This set will from now on be called *footprint*[1]. For a certain data structure at every point in time, this footprint is fixed. Since in separation logic data structures are described by assertions, this concept applies here as well. Note that in contrast to data structures, an assertion can have multiple footprints: the assertion $\mathtt{x} \mapsto_r \mathtt{a} \vee_A \mathtt{y} \mapsto_r \mathtt{b}$, where $x \neq y$ has the footprints x and y.

In separation logic there already exists the concept of precision [14] which is not to be confused with the presented concept of a footprint. An assertion A is precise, iff for any heap h there is at most one valid subheap (h, as) for which $A \models (h, as)$ holds. For different heaps the address set as may differ. An example for a precise assertion is the linked list assertion without the addresses in the abstract representation (presented in Section 4.1.1). If an assertion has a (strong or weak) fixed footprint (which will be defined in the next few paragraphs), it is also precise. The main difference between an assertion having a weak fixed footprint and being precise is that the former only allows one single address set (the footprint) for all heaps that satisfy the assertion, while latter allows one address set per heap that satisfies the assertion. This relationship was not formally proven Isabelle/HOL as the preciseness concept that is implemented in the used separation logic framework [11] differs from definition described here, even though both versions should describe the same concept. The framework version requires the precise assertion to take at least two arguments (that are not the partial heap).

[1] In the orignal separation logic paper [16] this term was used to describe the set of addresses used by a program.

The reason for introducing the footprint is that in the presented approach the abstract list and Fibonacci heap representations explicitly store the addresses they "occupy" in their abstract representation. The footprint is used to show that the addresses occurring in the abstract representations of two structures connected by the separating conjunction may not overlap. This becomes very important for the map representation, where this fact is essential to show that certain updates of the abstract map will not have unintended consequences. In other words this footprint is used to transfer the knowledge of the separation of the addresses, that is encoded in the separating conjunctions, out of the assertion part of the invariant into the Boolean predicates, that ensure that the map actually represents a valid heap (see Section 4.3.3 for more details on the heap assertion).

One way to define the footprint of an assertion is to state that an address set as is a valid footprint of an assertion A, if there exists a heap h so that for the partial heap (h, as) "$(h, as) \models A$" holds. This leads to the following definitions:

```
definition possible_footprint :: "assn ⇒ addr set ⇒ bool" where
"possible_footprint A as ≡ ∃ h. (h, as) ⊨ A"

definition possible_footprints :: "assn ⇒ addr set set" where
"possible_footprints A ≡ {x. possible_dom A x}"
```

Since for the following analysis only the separation logic operators $\mapsto$ and $*$ will be used (and $\exists_A$, but this operator can always be eliminated for the proofs), the used assertions will only have a single footprint. This is expressed with the following definition:

```
definition strong_fixed_footprint :: "assn ⇒ addr set ⇒ bool" where
"strong_fixed_footprint A as ≡ possible_doms A = {as}"
```

There is also a similar definition with a `_t` suffix, which indicates that the given set is a typed `'a ref set`. The reason for having two definitions is that the untyped addresses are used for the definition of partial heaps, while the points-to assertion always uses the typed `ref` type.

It is now possible to state the fixed footprints of `emp` and the points-to-assertion, which are the only two basic separation logic atoms that have a fixed footprint:

```
lemma fixed_footprint_emp: "strong_fixed_footprint_t emp {}"

lemma fixed_footprint_sngr: "strong_fixed_footprint_t (addr ↦ᵣ v) {addr}"
```

This definition has a slight quirk that makes some proofs slightly more verbose than necessary: an assertion that is false cannot have a fixed footprint. This is shown in the following lemma:

```
lemma strong_fixed_footprint_false: "¬ strong_fixed_footprint false fp"
```

The reason why this behavior might not be desired can be seen in the most important lemma that will be proven with footprints. It states that the separating conjunction of two assertions is `false` if the fixed footprints of the assertions overlap. With the additional assumption that the conjunction is not false to begin with, it is possibly to conclude that the footprints do not overlap. Due to this additional assumption (which will come from the fact that one may assume that the precondition of a Hoare triple is not `false`), it

does not matter what the footprint of the two assertions is, if one of them is `false`. In that case the separating conjunction of the two would also be `false`. To express that this case is not relevant, it makes sense to allow any arbitrary footprint for an assertion that is `false`. This weakened version (of which there is also a typed version, indicated by the same `_t` suffix) can be defined as follows:

```
definition weak_fixed_footprint :: "assn ⇒ addr set ⇒ bool" where
"weak_fixed_footprint A as ≡ ∀ h as'. (h, as') ⊨ A → as' = as"

lemma weak_fixed_footprint_false: "weak_fixed_footprint false fp"

lemma strong_fixed_footprint_weak_fixed_footprint:
"strong_fixed_footprint A as ⟷ (weak_fixed_footprint A as ∧ A ≠ false)"
```

Both definitions are used during this work, because for some proofs (one of which is the lemma shown next) it is actually easier to prove the version for the strong footprint and then prove the version for the weak footprint by using the above defined equivalence lemma. With these definitions, the already mentioned key lemma for the separating conjunction is proven:

```
lemma strong_fixed_footprint_star:
assumes "strong_fixed_footprint A as1" "strong_fixed_footprint B as2"
shows "strong_fixed_footprint (A * B) (as1 ∪ as2) ⟷ as1 ∩ as2 = {}"

corollary weak_fixed_footprint_star:
assumes "weak_fixed_footprint A as1" "weak_fixed_footprint B as2"
shows "weak_fixed_footprint (A * B) (as1 ∪ as2)"

corollary weak_fixed_footprint_star_false:
assumes "weak_fixed_footprint_t A as1" "weak_fixed_footprint_t B as2"
        "as1 ∩ as2 ≠ {}"
shows "A * B = false"
```

The key part of the Fibonacci heap assertion will be the separating product, so the footprint of such a product is defined here already. Note that for the product to be well defined the set has to be finite.

```
lemma fixed_footprint_prod_sngr:
"finite as ⟹ strong_fixed_footprint_t (Π a ∈ as. a ↦r F a) as"
```

4.3.2 Circular Linked List

When an abstract map is used to store the abstract objects (in the case of the linked list the abstraction of the value stored in the nodes), it is now no longer necessary to store the abstract nodes in the list itself. The addresses are sufficient, as they can be used to query the abstract map for the abstract values. So from a list of addresses it is possible to generate the abstract list used in the list assertion shown in the previous section with the help of the following expression:

```
map (λ addr. Dll_Abs_Node addr (nm addr)) addr_list
```

The problem with using this approach is that, the transformation makes the already quite complicated proofs, that come up during the verification of the Fibonacci heap, even more complex. For that reason it was decided to amend the previously shown definition of the list assertions to directly use the abstract map. This means that the assertion for the doubly linked list segment now works as follows (with the second parameter being the abstract node map):

```
abbreviation dll_seg_map_single where
"dll_seg_map_single F nm x pre next ≡ x ↦r (Dll_Node (F (nm x)) pre next)"

fun dll_seg_map ::  "('a ⇒ 'b) ⇒ ('b dll ref ⇒ 'a) ⇒ 'b dll ref list ⇒
                     'b dll ref option ⇒ 'b dll ref option ⇒ assn" where
  "dll_seg_map _ _ [] _ _ = emp"
| "dll_seg_map F nm [x] pre end_next = dll_seg_map_single F nm x pre end_next"
| "dll_seg_map F nm (x#y#xs) pre end_next = dll_seg_map_single F nm x pre (Some y)
                                          * dll_seg_map F nm (y#xs) (Some x) end_next"
```

For this definition it can be shown that is is equivalent to the previous version of the segment assertion (Section 4.2.1), when the above transformation function is used. While this would allow to directly reuse the list lemmas proven for the previous list definition, equivalent lemmas were defined for the new version, so that the proofs that make use of them do not have to deal with the transformation.

```
lemma dll_seg_map_dll_seg:
   "dll_seg_map F nm l pre end_next =
    dll_seg F (map (λ addr. Dll_Abs_Node addr (nm addr)) l) pre end_next"
```

The lemmas for splitting the list segments and list were defined as for the list presented in the previous section, so they are not repeated here. The footprint of the list is defined as follows:

```
lemma dll_seg_map_footprint:
   "weak_fixed_footprint_t (dll_seg_map F nm l pre end_next) (set l)"
```

It should be possible to prove the strong footprint, as the segment assertion is just a separating product of points-to assertions, but only the weak footprint is needed for the proofs.

The newly introduced map makes it necessary to prove two new lemmas concerning the list, which state that the list is not changed when the abstract map is updated at an address which is not part of the footprint of the list:

```
lemma dll_seg_map_eq:
assumes "⋀ x. x ∈ set l ⟹ nm1 x = nm2 x"
shows "dll_seg_map F nm1 l prev end_next = dll_seg_map F nm2 l prev end_next"

lemma dll_seg_map_upd_noop:
assumes "x ∉ set l"
shows
   "dll_seg_map F (fun_upd m x y) l prev end_next = dll_seg_map F m l prev end_next"
```

If the approach with the abstract map was used for more types, it would probably make sense to define an analogous concept of the assertion footprint for the map to unify these

proofs for different assertions. For the map, the footprint would be the smallest set of addresses for which an arbitrary change of the map at an address outside of this set does not change the assertion. Both the map footprint and the assertion footprint should always be the same.

With these lemmas it is now possible to prove all of the Hoare triples defined for the functions defined in the previous section again. The first two functions are the ones that generate a new list:

```
lemma cdll_map_empty_rule: "<emp> cdll_empty <cdll_map_assn F (λ x. undefined) []>"

lemma cdll_map_singleton_rule:
assumes "F x_abs = x_conc"
shows
   "<emp>
    cdll_singleton x_conc
    <λptr. cdll_map_assn F (fun_upd (λx. undefined) (the ptr) x_abs) [the ptr] ptr>"
```

Here the problem with the used map representation becomes visible. In both cases `λx. undefined` could have been replaced by any other function, for these Hoare triples to be correct. The reason why an explicit function was given, instead of using a free variable for the function is that it is easier for the automation to deal with a concrete value. There also should not be a reason for using any other function.

The Hoare triples for the other list functions are defined as follows:

```
lemma cdll_map_prepend_rule:
assumes "F x_abs = x_conc"
shows
   "<cdll_map_assn F nm l ptr>
    cdll_prepend ptr x_conc
    <λptr. cdll_map_assn F (fun_upd nm (the ptr) x_abs) ((the ptr)#l) ptr>"

lemma cdll_map_append_rule:
   "<cdll_map_assn F nm1 xs list_ptr1 * cdll_map_assn F nm2 ys list_ptr2>
    cdll_append_cdll list_ptr1 list_ptr2
    <cdll_map_assn F (override_on nm1 nm2 (set ys)) (xs@ys)>"

lemma cdll_unlink_rule:
assumes "addr ∈ set l"
shows
   "<cdll_map_assn R nm l list_ptr>
    cdll_unlink list_ptr addr
    <λptr. cdll_map_assn R nm [addr] (Some addr)
           * cdll_map_assn R nm (remove1 addr l) ptr>"
```

The `override_on` function returns a function that returns the values of the second given function if an input appears in the given set (the last parameter) and the values of the first function for all other inputs. To show that `override_on` does not overwrite any values stored by the first list, the footprint is used to show that `xs ∩ ys = {}`.

A key feature of circular linked lists, which was not shown until now, but which is important for the Fibonacci heap, is that any node can be chosen as the starting node of

the list. This is important for the Fibonacci heap verification, as here the heap pointer, that points to the minimum of the list, might be set to another node in the root list, either because decrease-key or insert was called. The corresponding lemmas look are defined as follows:

```
lemma cdll_assn'_rotate_single: "cdll_assn' F (x#xs) = cdll_assn' F (xs @ [x])"

lemma cdll_assn'_rotate: "cdll_assn' F (xs @ ys) = cdll_assn' F (ys @ xs)"
```

The second lemma can be proven by induction over the length of the first list using the first lemma.

4.3.3 Fibonacci Heap

The Fibonacci Heap invariant that will be used as pre- and postcondition for the Hoare triples of the heap functions is split into three parts, which will be presented in this subsection in the following order:

1. **An assertion part** which converts the nodes stored in the map to a separation logic assertion which ensures that the heap memory correctly models the state of the map.
2. **A structural invariant** that ensures that the abstract map models a heap (structurally).
3. **A logical invariant** that ensures that the rank and order invariants hold in the abstract map.

The first two parts were previously covered by the function converting the abstract rose tree to an assertion. The last part is equivalent to the Boolean predicates on the rose tree as defined by Stüwe (see Section 4.1.2).

To differentiate the heap invariant which encompasses all three parts (and which is a separation logic assertion) from the assertion part, the former will be called *heap invariant* while the latter will be referred to as *heap assertion (part)*.

Assertion

In this approach the abstract representation of a heap node is almost the same as in the previous approach:

```
datatype 'a abs_fib_tree = Abs_Fib_Node
(abs_fib_rank: nat) (abs_fib_mark: bool) (abs_fib_val: 'a)
(abs_fib_children: "'a fib_tree dll ref list")
(abs_fib_parent: "'a fib_tree dll ref option")
```

The big difference is that the recursive nature of the definition is now broken, since the child list does not contain full elements, but only addresses to elements. It is the responsibility of the abstract map to provide the abstract nodes at these addresses and the lost structure now needs to be defined in additional structural invariants on the heap.

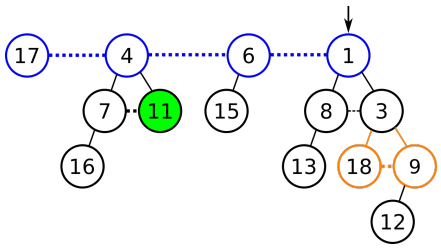

Figure 4.2: In this simplified representation of a Fibonacci heap, the nodes described by the assertion generated by `child_list_assn` for the address of node 3 is marked orange. The assertion generated by the part responsible for the root list is marked blue.

As member accesses to the abstract list node will quite often be used in combination with a query to the abstract node map, the notation `abs_fib_prnt nm addr` will be used to indicate the expression `abs_fib_parent (nm addr)` (the same holds for the other members analogously).

As the members of the abstract object are normally queried by first retrieving the object from the abstract map, the following definitions will use a shortened notation where the vowels are left out. As an example, the notation `abs_fib_chldrn nm addr` will be equivalent to `abs_fib_children (nm addr)`.

The imperative implementation of the heap node is the same as in the previous approach:

```
datatype 'a fib_tree = Fib_Node
(fib_rank: nat) (fib_mark: bool) (fib_val: 'a)
(fib_children: "'a fib_tree dll ref option")
(fib_parent: "'a fib_tree dll ref option")
```

The basic heap assertion is now defined as a composition of the assertions for the child lists of every node together with the assertion for the root list. the assertion generating function has four parameters: the abstract map, which is represented by an address set and the abstract node map, the representation of the root list, which is the list of the addresses of the nodes in the root list, and the heap pointer. A visualization of the following definition of this assertion can be seen in Figure 4.2.

```
abbreviation fib_node_from_abs where
"fib_node_from_abs abs_node ≡
   Fib_Node (abs_fib_rank abs_node) (abs_fib_mark abs_node) (abs_fib_val abs_node)
            (hd_option (abs_fib_children abs_node)) (abs_fib_parent abs_node)"

abbreviation child_list_assn where
"child_assn nm addr ≡ cdll_map_assn' fib_node_from_abs nm (abs_fib_chldrn nm addr)"

definition child_lists_assn where
"child_lists_assn aset nm rl ≡ (Π addr ∈ aset. child_assn nm addr)"
```

```
fun fib_heap_assn
   :: "'a::heap fib_tree dll ref set ⇒ ('a fib_tree dll ref ⇒ 'a abs_fib_tree) ⇒
       'a fib_tree dll ref list ⇒ 'a fib_tree dll ref option ⇒ assn" where
"fib_heap_assn aset nm rl ptr =
   cdll_map_assn fib_node_from_abs node_map root_list ptr
   * child_lists_assn aset node_map root_list"
```

Structural Invariant

Due to the lack of structure in the abstract representation, the assertion shown in the previous section only states that the heap consists of several lists, one root list and several child lists. Nothing is said about the relationship of these lists. To introduce this relationship, the concept of reachability has to be defined for the abstract map.

This concept can be defined in Isabelle/HOL with the help of an inductive set. Here one starts with a root set (which in the case of the Fibonacci heap contains the addresses in the root list) to which other addresses are added according to the following inductive construction: an element is in the inductive set if it is contained in the address set associated with an address that is already in the inductive set. This association is represented by a map that maps each address to an address set (which, in the case of a Fibonacci heap, will map each address to the set of the address of its children). This is expressed in Isabelle/HOL in the following way:

```
inductive_set map_rec_set :: "('a ⇒ 'a set) ⇒ 'a set ⇒ 'a set"
for set_map :: "('a ⇒ 'a set)"
and root_set :: "'a set"
where
  "x ∈ root_set ⟹ x ∈ map_rec_set set_map root_set"
| "p ∈ map_rec_set set_map root_set ⟹
   c ∈ set_map p ⟹ c ∈ map_rec_set set_map root_set"
```

While Isabelle/HOL can generate induction schemes for this definition, which are quite useful, this is probably not how a human thinks about this set. A more intuitive definition of the recursive set looks like this:

```
lemma map_rec_set_inductive_union:
   "map_rec_set set_map rs = rs ∪ (⋃ x ∈ rs. map_rec_set set_map (set_map x))"
```

Another alternative way of describing this set is that it contains the element reachable from the root set within the reflexive transitive closure over the relation between an address and the addresses in the set associated with it.

With this set one can define the set of all reachable elements from the root list using the presented map function (root list → root set, child lists → sets stored in the map):

```
abbreviation node_map_to_child_set_map
   :: "('a ⇒ 'b abs_fib_tree) ⇒ 'a ⇒ 'b fib_tree dll ref set" where
"node_map_to_child_set_map nm ≡ set ∘ abs_fib_children ∘ nm"

abbreviation child_set :: "('a fib_tree dll ref ⇒ 'a abs_fib_tree) ⇒
                           'a fib_tree dll ref list ⇒ 'a fib_tree dll ref set" where
"child_set nm rl ≡ map_rec_set (node_map_to_child_set_map nm) (set rl)"
```

Some intuitive statements that hold for the heap, can be defined for this set. The first important observation is that if an address is in the recursive set, than the recursive set going out from only that address is completely in the recursive set. This captures the concept, that if a node is reachable in the heap, then the subtree rooted in that node is also reachable.

```
lemma map_rec_set_subset':
   "x ∈ map_rec_set set_map rs ⟹ map_rec_set set_map {x} ⊆ map_rec_set set_map rs"
```

As this set will be used in the heap to represent all addresses in the footprint of the heap, it needs to be shown how different operations may affect the heap. For decrease-key one has to (potentially) repeatedly remove elements (represented here by addresses) from different child lists and readd them to the root. The effects of these operations can be shown with the following two lemmas:

```
lemma map_rec_set_union:
   "map_rec_set set_map (rs ∪ rs') = map_rec_set set_map rs
                                     ∪ map_rec_set set_map rs'"

lemma map_rec_set_remove_single:
assumes "remove_addr ∈ map_rec_set set_map rs"
shows "map_rec_set set_map rs =
          map_rec_set (in_place_upd set_map (λ x. x - {remove_addr}) addr) rs
          ∪ map_rec_set set_map {remove_addr}"
```

The `in_place_upd` function returns the given function representing a map, where the given transformation function is applied to the element at the given key.

For heap-union it is necessary to merge two different heaps, each of which brings its own map. This can be expressed with the following lemma:

```
lemma map_rec_set_combine:
assumes
   "map_rec_set sm1 rs1 ∩ map_rec_set sm2 rs2 = {}"
shows
   "map_rec_set (override_on sm1 sm2 (map_rec_set sm2 rs2)) (rs1 ∪ rs2) =
    map_rec_set sm1 rs1 ∪ map_rec_set sm2 rs2"
```

With this concept of reachability, it is already possible to define the structural heap invariant. The first part is the reachability requirement, which is expressed with the just introduced recursive set and which states that every node in the heap (indicated by its address) has to be reachable from the root list:

```
abbreviation aset_eq_child_set where
"aset_eq_child_set aset nm rl ≡ aset = child_set nm rl"
```

Then the requirements on the parent pointer for each node are expressed: Nodes in the root list must not have a parent, which is expressed by the parent pointer being the nullpointer `None` in the case of Imperative HOL). For all other nodes the parent pointer has to point to the node, which contains the former in its child list:

```
abbreviation rl_parent_none where
"rl_parent_none nm rl ≡ ∀ x ∈ set rl. abs_fib_prnt nm x = None"
```

```
abbreviation prnt_of_children where
"prnt_of_children aset nm ≡
   ∀ p ∈ aset. ∀ c ∈ set (abs_fib_chldrn nm p). abs_fib_prnt nm c = (Some p)"
```

This last fact is strong enough to prove that an element may not appear in two different child lists of nodes in the heap, as a node may not have two different parents. This can also be shown by assuming that the heap assertion is not `false`, but using the parent requirement is a bit simpler.

The last part of the invariant ensures that the set of addresses occupied by the heap is finite. This is required to show that the separating product in the heap assertion is well defined.

```
abbreviation finite_aset :: "'a fib_tree dll ref set ⇒ bool" where
"finite_aset aset ≡ finite aset"
```

The combination of these then makes up the structural invariant of the heap. Combined with the heap assertion it should now make up for the information lost by using a map instead of a data structure to represent the heap.

```
definition fib_heap_struc_inv
   :: "'a fib_tree dll ref set ⇒ ('a fib_tree dll ref ⇒ 'a abs_fib_tree) ⇒
       'a fib_tree dll ref list ⇒ bool" where
"fib_heap_struc_inv aset nm rl ⟷
   aset_eq_child_set aset nm rl ∧
   rl_parent_none nm rl ∧
   prnt_of_children aset nm ∧
   finite_aset aset"
```

Two quite useful lemmas connected to the structural invariant express the inverse of the parent part of the invariant. The first lemma states, that if the parent of a node is `None`, that node has to be in the root list. The second lemma states, that if the parent pointer of a node points to another node, the former has to be in the child list of the latter.

```
lemma fib_heap_parent_none:
assumes
   "fib_heap_struc_inv aset nm rl"
   "x ∈ aset"
shows
   "abs_fib_prnt nm x = None ⟷ x ∈ set rl"

lemma fib_heap_parent_some:
assumes
   "fib_heap_struc_inv aset nm rl"
   "x ∈ aset"
   "abs_fib_prnt nm x = (Some y)"
shows
   "y ∈ aset" "x ∈ set (abs_fib_chldrn nm y)"
```

There is a key attribute missing, that one would expect from the heap being a list of trees: each node should only be reachable by a single path. For this analysis, it makes sense to view each list as a single node. The used path definition stores all nodes, for

which the child pointer is followed to reach the last node (so in the case of the example heap shown in Figure 4.2, the path from the root list to the node with value "18" would be $1 \rightarrow 3 \rightarrow 18$). Currently, it is only required that an element is reachable from the root, but there is no restriction on the number of of paths to it. As it turns out, the requirements on the parents of nodes are already strong enough to prove this fact. The intuition is that each node in a child list cannot appear in any other child list (as it may have only one parent), so it can also only be reached by a single path. The same is true for the nodes in the root list. As they are not in any child list, they can also not be reached by a path that contains more than just the node itself.

Explicitly stating the uniqueness of the paths is actually not necessary for the verification of decrease-key, but what needs to be shown is the absence of circles. This implies that a root of a subtree cannot appear again in the subtree (which means that it cannot be its own child or grand child) and it also means that if one follows the parent pointers from a node to the root of its tree, the node cannot reappear. The used proof is somewhat long, but we did not spent too much time on finding optimizations for the used approach. Firstly, the concept of paths has to be introduced. For that, the same abstraction as for the reachability is used (`root_set`, `set_map`):

```
fun map_rec_set_path :: "('a ⇒ 'a set) ⇒ 'a list ⇒ bool"  where
  "map_rec_set_path sm [] ⟷ False"
| "map_rec_set_path sm [x] ⟷ True"
| "map_rec_set_path sm (x#y#l) ⟷ y ∈ sm x ∧ map_rec_set_path sm (y#l)"

abbreviation rooted_map_rec_set_path where
"rooted_map_rec_set_path sm rs l ≡ map_rec_set_path sm l ∧ hd l ∈ rs"
```

These paths function as one would expect: Each (non-empty) prefix and suffix of a path (for the rooted path only the former) are valid paths and the paths can also be split and merged. With the path definition it is now possible to define an alternative definition for the reachability of a node in the tree (there has to be a rooted child path to it):

```
lemma map_rec_set_ex_map_rec_set_path:
   "x ∈ map_rec_set sm rs ⟷ (∃ p. rooted_map_rec_set_path sm rs (p@[x]))"
```

To be able to guarantee that there are no circles, the concept of circles needs to be defined first. Here they are just paths where the head of the path is in the address set (which for the heap are the addresses in the abstract child list) of the last element of the path:

```
abbreviation map_rec_set_circle where
"map_rec_set_circle sm l ≡ map_rec_set_path sm l ∧ hd l ∈ sm (last l)"
```

This definition is used instead of a path which starts and ends in the same node as in the that case one needs to also verify that the path length is at least two.

Back to the heap itself: As mentioned, the reason for the uniqueness of paths is the uniqueness of parents, so it makes sense to define parent paths as well. It can be shown that in a heap a parent path is a reverse child path:

```
fun parent_path  :: "('a fib_tree dll ref ⇒ 'a abs_fib_tree) ⇒
                     'a fib_tree dll ref list ⇒ bool" where
  "parent_path nm [] ⟷ False"
| "parent_path nm [x] ⟷ True"
| "parent_path nm (x#y#ys) ⟷ abs_fib_prnt nm x = (Some y) ∧ parent_path nm (y#ys)"

lemma child_path_imp_parent_path:
assumes
   "fib_heap_struc_inv aset nm rl"
   "set l ⊆ aset"
shows "child_path nm l ⇒ parent_path nm (rev l)"
```

Now several lemmas are shown that are needed to prove the absence of circles. The idea is to show that there is no way of reaching the root from a node that is inside of a circle. In a first step it is shown that all parents of nodes in a circle also have to appear in the circle. Then it is shown that if all parents of nodes in a certain set are in the same set, each path starting in this set has to stay in this set. The third and last lemma shows that an element that is in a circle may not be in the root list, as its parent cannot be `None`. With these lemmas it can be shown that an element cannot be part of a rooted path and a circle at the same time (as any path must always stay in the circle). All of these lemmas only hold if all of the elements in the paths/circles are completely part of the heap (in its address set), so it also needs to be shown that any path that starts at a node in the heap stays in the heap. This leads to the final lemma where it is shown that an element in a circle cannot be in the heap:

```
lemma circle_not_in_aset:
assumes
   "fib_heap_struc_inv aset nm rl"
   "child_circle nm l"
   "x ∈ set l"
   "x ∈ aset"
shows "False"
```

With these lemmas and the structural invariant itself, the structure of the heap is now described in a way that suffices for the decrease-key verification.

Logical Invariant

The next step is to define the last part of the heap invariant, that ensures the correctness and the running time of the heap functions (if the heap, for which these functions are called, is also structurally valid). The following expressions that concern the order and the rank of the nodes are just adapted version of the ones presented in Section 4.1.2, which were directly taken from Stüwe's thesis [17].

Firstly, the parts of the invariant concerning the order of elements in the heap are defined. For each node its children have to be bigger than the parent and the element pointed to by the heap pointer has to be the smallest element in the root list.

```
abbreviation root_list_order where
"root_list_order nm rl ≡ ∀ x ∈ set rl. abs_fib_vl nm (hd rl) ≤ abs_fib_vl nm x"
```

```
abbreviation tree_order where
"tree_order aset nm ≡ ∀ p ∈ aset. ∀ c ∈ set (abs_fib_chldrn nm p).
                                    abs_fib_vl nm p ≤ abs_fib_vl nm c"
```

There are two parts of the invariant which deal with the rank of each node. Firstly the rank has to be equal to the length of the abstract child list:

```
abbreviation rank_invariant where
"rank_invariant aset nm ≡
   ∀ p ∈ aset. length (abs_fib_chldrn nm p) = abs_fib_rnk nm p"
```

Secondly the rank of a node in the children list is lower bounded by position in the children list and if it is marked it can have one child less:

```
definition child_rank_invariant where
"child_rank_invariant aset nm ≡
   ∀ p ∈ aset. ∀ i ∈ {0..<length (abs_fib_chldren nm p)}.
                        abs_fib_rnk nm (abs_fib_chldrn nm p ! i)
                        ≥ i - of_bool (abs_fib_mrk nm (abs_fib_chldrn nm p ! i))"
```

The notation `list ! i` accesses the element at index `i` (starting at 0) in the list.

There is also a last part of the invariant which is not directly needed for the proof of the correctness, but which will be useful to define the heap invariant that is exposed by the Hoare triples for the heap functions. In addition to the heap pointer, the caller of a heap function needs to know two things about the heap: the values that are stored in it, and the addresses at which each value is stored (to be able to call decrease-key for the right elements). This knowledge can be represented with a map that maps addresses to values (and nothing else). The domain of this map is already given as it is equal to the domain for the node map, so only the function that maps addresses to the values is needed. The validation that this map represents the contents of the heap is done with the following expression:

```
abbreviation matching_values where
"matching_values nm aset val_map ≡ ∀ x ∈ aset. abs_fib_vl nm x = val_map x"
```

These five expressions are joined (with a logical conjunction) in the logical invariant of the heap:

```
definition fib_heap_logic_inv where
"fib_heap_logic_inv aset nm rl val_map ⟷
   root_list_order nm rl ∧
   tree_order aset nm ∧
   rank_invariant aset nm ∧
   child_rank_invariant aset nm ∧
   matching_values nm aset val_map"
```

Full Heap Invariant

The three presented parts, the assertion part, the structural invariant, and the logical invariant make up the complete separation logic heap invariant. Keeping in mind that

the structural and logical invariant are just boolean predicates on the abstract node map (and some other parameters), the full heap invariant can be defined as follows:

```
fun fib_heap_inv
   :: "'a::{linorder, heap} fib_tree dll ref set ⇒
       ('a fib_tree dll ref ⇒ 'a abs_fib_tree) ⇒
       'a fib_tree dll ref list ⇒ ('a fib_tree dll ref ⇒ 'a) ⇒
       'a fib_tree dll ref option ⇒ assn" where
"fib_heap_inv aset nm rl val_map ptr =
   ↑(fib_heap_struc_inv aset nm rl ∧ fib_heap_logic_inv aset nm rl val_map)
   * fib_heap_assn aset nm rl ptr"
```

This invariant is a separation logic assertion, but in the following sections, to be able to differentiate between the assertion part of the invariant and the full invariant, the latter is only referred to as (heap) invariant, while the former may be referred to as heap assertion. The boolean parts are only referred to by structural or logical invariant respectively.

As mentioned the Hoare triples should not expose the internal state of the heap, as it is irrelevant for the correct usage of the heap functions. This is achieved by existentially quantifying the state, as was done in the following definition:

```
abbreviation fib_heap_user_inv
   :: "'a::{linorder, heap} fib_tree dll ref set ⇒ ('a fib_tree dll ref ⇒ 'a) ⇒
       'a fib_tree dll ref option ⇒ assn" where
"fib_heap_user_inv aset val_map ptr ≡ ∃A nm rl. fib_heap_inv aset nm rl val_map ptr"
```

All proofs of Hoare triples that appear in this thesis are done with the exact invariant if possible. The Hoare triples with existentially quantified variables directly follow from the Hoare triples with no existentially quantified variables. This is just the separation logic Hoare triple version of a basic statement that holds for predicate logic:

```
(∀ x. P x ⟶ Q (F x)) ⟹ ((∃ x. P x) ⟶ (∃ x. Q x))
```

The separation logic version:

```
(∀ x. <P x> c <λ ret. Q (F x) ret> ⟹ <∃A x. P x> c <λ ret. ∃A x. Q x ret>
```

With this the definition of the heap invariant is complete. This makes it possible to verify the heap functions.

4.3.4 Decrease-Key Implementation

Before starting with the verification, the implementation of decrease-key and its two auxiliary functions, cut and cascading-cut, are shown. The reasons for splitting cut and decrease-key are explained in their respective verification subsections. Descriptions of these functions can be found in Section 2.4.

```
definition imp_remove_child where
"imp_remove_child heap_ptr child_addr prnt_addr = do {
   prnt_el ← !prnt_addr;
   new_children ← cdll_unlink (fib_get_children prnt_el) child_addr;
   let prnt_el' = fib_set_children prnt_el new_children;
   prnt_addr := fib_set_rank prnt_el' ((fib_get_rank prnt_el) - 1)
}"

definition imp_set_child_as_root where
"imp_set_child_as_root heap_ptr child_addr = do {
   cdll_append_cdll heap_ptr (Some child_addr);
   child_el ← !child_addr;
   child_addr := fib_set_mark (fib_set_parent child_el None) False
}"

definition fib_cut
   :: "'a::heap fib_heap_node ref option ⇒ 'a fib_heap_node ref ⇒
       'a fib_heap_node ref ⇒ unit Heap" where
"fib_cut heap_ptr child_addr prnt_addr = do {
   imp_remove_child heap_ptr child_addr prnt_addr;
   imp_set_child_as_root heap_ptr child_addr
}"

partial_function (heap) fib_cascading_cut
   :: "'a::heap fib_heap_node ref option ⇒
       'a fib_heap_node ref ⇒ unit Heap" where
"fib_cascading_cut heap_ptr curr_addr = do {
   curr_el ← !curr_addr;
   let prnt_ptr = fib_get_parent curr_el;
   if prnt_ptr ≠ None then do{
      if ¬ fib_get_mark curr_el then do {
         curr_addr := fib_set_mark curr_el True
      } else do {
         fib_cut heap_ptr curr_addr (the prnt_ptr);
         fib_cascading_cut heap_ptr (the prnt_ptr)
      }
   } else
      return ()
}"
```

Listing 4.4: The implementation of cut and cascading-cut

```
definition fib_decrease_key_cut where
"fib_decrease_key_cut heap_ptr dec_addr new_key = do {
   dec_el ← !dec_addr;
   let prnt_ptr = fib_get_parent dec_el;
   if prnt_ptr ≠ None then do {
      let prnt_addr = the prnt_ptr;
      prnt_el ← !prnt_addr;
      if new_key < fib_get_val prnt_el then do {
         fib_cut heap_ptr dec_addr prnt_addr;
         fib_cascading_cut heap_ptr prnt_addr
      } else
         return ()
   } else
         return ()
}"

definition fib_decrease_key_decrease where
"fib_decrease_key_decrease heap_ptr dec_addr new_key = do {
   dec_el ← !dec_addr;
   dec_addr := fib_set_val dec_el new_key;
   min_el ← !(the heap_ptr);
   if new_key < fib_get_val min_el then
      return (Some dec_addr)
   else
      return heap_ptr
}"

definition fib_decrease_key
   :: "'a::{heap, linorder} fib_heap_node ref option ⇒ 'a fib_heap_node ref ⇒
       'a ⇒ 'a fib_heap_node ref option Heap" where
"fib_decrease_key heap_ptr dec_addr new_key = do {
   fib_decrease_key_cut heap_ptr dec_addr new_key;
   fib_decrease_key_decrease heap_ptr dec_addr new_key
}"
```

Listing 4.5: The implementation of decrease-key

4.3.5 Verification of Cut

As can be seen in the previous subsection, the cut operation was split into two parts: one part that removes the element to be cut from the child list of its parent (note a node already in the root list can never be cut) and one part that readds that node to the root list. The reason for splitting the cut function is the used separation logic framework. As described in Section 3.3, the rules for the lookup and update of a memory location indicated by a certain address are only defined for assertions of the following form `addr` $\mapsto_r$ `val`, where additional assertions might be pre- or appended with the separating conjunction (due to the frame rule). The heap assertion itself is a black box, where these terms are hidden somewhere in the list assertions defining the root and child lists. So for the lookup and update, these invariants have to be split. For the verification of `imp_remove_child`, the heap invariant needs to be split in such a way that the `prnt_addr` points-to assertion

is exposed, to be able to read and modify it. If the parent is in the root list, the list assertion for the root list is split. The function `imp_set_child_as_root` appends the cut child element to the root list with the list append function `cdll_append_cdll` for which a Hoare triple was previously defined (Section 4.3.2). This Hoare triple requires an intact list invariant. If cut were to be proven as one function, the root list invariant first would need to be split to access the parent, and then the invariant would need to be restored to be to use the Hoare triple of `cdll_append_cdll`. Restoring the invariant once split is actually somewhat difficult with the used framework, so it was deemed to be easier to split the function and manually define a Hoare triple using the full heap invariants as pre- and postconditions.

If one needs to only lookup the value stored at an address, it is actually possible to define manual Hoare triples for just the lookup for each defined invariant, to avoid the need to split. For the heap and list this looks as follows:

```
lemma cdll_map_assn'_lookup_rule:
assumes "x ∈ set l"
shows
   "<cdll_map_assn' F nm l>
    !x
    <λ c. ↑(dll_val c = F (nm x)) * cdll_map_assn' F nm l>"

lemma fib_heap_assn_lookup_rule:
assumes "x ∈ aset" "fib_heap_struc_inv aset nm rl"
shows
   "<fib_heap_assn aset nm rl heap_ptr>
    !x
    <λ c. ↑(dll_val c = fib_node_from_abs (nm x))
          * fib_heap_assn aset nm rl heap_ptr>"
```

Such lemmas cannot be defined for writing to a memory cell, because the write may destroy the invariant. That means there is not necessarily a nice way to define the postcondition.

Another more general, important idea for the verification is expressed in the following lemma, which states that the boolean (the pure assertions) and assertion part of a Hoare triple can be proven separately:

```
lemma mixed_assn_Hoare_triple:
assumes "(P ≠ false ⟹ Q ⟹ Q')"
        "(Q ⟹ <P> c <λ x. P' x>)"
shows   "<↑(Q) * P> c <λ x. ↑(Q') * P' x>"
```

The assumption that the assertion is not `false` is needed for the proof of the boolean part, because it is used to show that the lists that appear in the abstract map (and also the root list) must not contain any address twice.

This lemma is not very profound. In fact it is a direct consequence of already existing lemmas in the framework, but it is very useful, as it separates concerns. For the cut it also means that it is not necessary to find Boolean invariants for the two parts of cut. It is enough to state a Boolean pre- and postcondition for the whole cut. So as a first step, the assertion part of the invariant was proven for the two parts of cut (and with that for

the whole cut operation). Note that result of cut is also structurally a valid heap. Only the logical invariant may be violated after a cut (this will be discussed in more detail later in this subsection).

For the first part of cut, that removes the node indicated by the given address from the heap, one needs to differentiate whether the parent of the node to be cut is in the root list or in a child list, as different parts of the heap invariant need to be split. Otherwise the proofs are very similar. The resulting Hoare triple for the assertion part looks as follows:

```
lemma imp_remove_child_rule:
assumes
   "fib_heap_struc_inv aset nm rl"
   "chld ∈ aset"
   "abs_fib_prnt nm chld = (Some prnt)"
shows
   "<fib_heap_assn aset nm rl heap_ptr>
    imp_remove_child heap_ptr chld prnt
    <λ _. fib_heap_assn aset (remove_chld nm prnt chld) rl heap_ptr
        * cdll_map_assn' fib_node_from_abs (remove_chld nm prnt chld) [chld]>"
```

The `remove_chld` function is the abstract representation of the operation. It updates the given node map by removing the second given address (the third parameter) from the child list stored at the node indicated by the first given address (the parent) and it also updates the rank of the parent. There is an interesting thing to note about the postcondition: Even though the node with address `child_addr` was removed from the heap, the assertions for the child lists in the subtree rooted in that node are still generated by the heap assertion that contains the `child_addr` in its address set. The reason why this works is that the assertion generating function generates only the assertion for the child list of each node and not for the node itself.

A key part of the proof, that also has to be done for all other proofs, is to show that the child address occurs at only one location in the abstract heap representation. With the lemma for the distinctness of addresses occurring in abstract list representation, which was already defined for the list assertion in Section 4.2.1, and the lemma stating the uniqueness of parents, it is shown that the updated node may only occur at one place in the heap (either in the root list or in a child list). This means that the update on the abstract map only changes the Fibonacci heap in one place. Instead of using the uniqueness of parents to prove that a certain node can be only in a single list, one could also proof this by assuming that the heap assertion is not `false` (note that this fact is already used to show that the lists are distinct). The former approach just seemed easier. Proving the uniqueness of the address is necessary for all heap operations that change a node (both for the assertion and the boolean parts of the invariant). The proofs are always the same.

The next step is the proof of the Hoare triple for the assertion part for the second part of cut `imp_set_child_as_root`, that readds the just removed element to the root list. Here only a single case needs to be considered for the proof. The resulting Hoare triple looks as follows:

```
lemma imp_set_child_as_root_rule:
assumes
   "fib_heap_struc_inv aset nm rl"
   "chld ∈ aset"
   "abs_fib_prnt nm chld = (Some prnt)"
shows
   "<fib_heap_assn aset (remove_chld nm prnt chld) rl heap_ptr
     * cdll_map_assn' fib_node_from_abs (remove_chld nm prnt chld) [chld]>
    imp_set_child_as_root heap_ptr chld
    <λ _. fib_heap_assn aset (abs_fib_cut nm chld prnt) (rl@[chld]) heap_ptr>"
```

Here `abs_fib_cut` combines the effect of `remove_chld` with an update of the mark and parent of the cut element (which are set to `False` and `None`) respectively. This Hoare triple proof makes use of the footprint. It is used to show that the address `child_addr` of the node that was cut, which makes up the footprint of the singleton list, may not appear in the footprint of the rest of the heap. That means that if the cut node is updated, when it is added to the root list, the rest of the heap does not change.

The Hoare triple for the heap assertion part for cut follows directly from the previous two Hoare triples:

```
lemma fib_cut_rule:
assumes
   "fib_heap_struc_inv aset nm rl"
   "chld ∈ aset"
   "abs_fib_prnt nm chld = (Some prnt)"
shows
   "<fib_heap_assn aset nm rl heap_ptr>
    fib_cut heap_ptr chld prnt
    <λ _. fib_heap_assn aset (abs_fib_cut nm chld prnt) (rl@[chld]) heap_ptr>"
```

So now the boolean parts of the invariant need to be proven. The first step here is to prove that the structural invariant still holds for the abstract map after a cut. This is shown in the following lemma:

```
lemma fib_cut_rule_bool_struc:
assumes "chld ∈ aset"
   "abs_fib_prnt nm chld = (Some prnt)"
   "fib_heap_struc_inv aset nm rl"
   "⋀ x. x ∈ aset ⟹ distinct (abs_fib_chldrn nm x)"
shows "fib_heap_struc_inv aset (abs_fib_cut nm chld prnt) (rl@[chld])"
```

The most difficult part of the invariant is the proof for the preservation of the reachability. For that the `map_rec_set` lemmas (see Section 4.3.3) are needed. The distinctness of the children lists is needed to prove the fact that parent of the nodes in the changed heap is the same one as in the unchanged heap (except for the cut element of course). This distinctness can be proven with the fact that the heap assertion can be assumed to be not false, as mentioned previously in this subsection.

So now the logical invariant has to be proven. It can be violated by the cut, so to determine the intermediate invariants that cut can produce, one needs to look at the different calling contexts of cut. In the used implementation, cut occurs at two

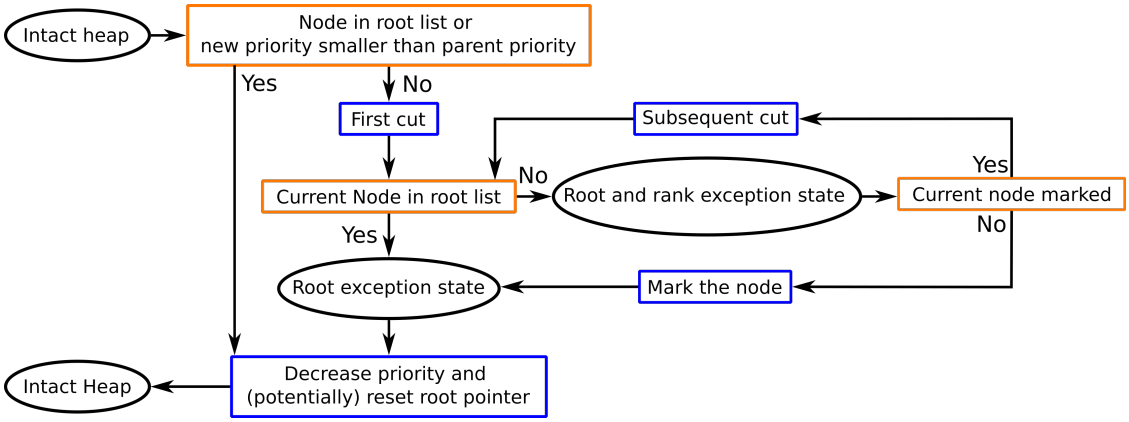

Figure 4.3: A simplified version of the control flow through decrease-key with the different heap states (≙ invariants) that are relevant for the calls to cut. Note that the current node, which is the node that may violate the child rank invariant, changes directly after every cut to its parent. Directly after every cut there is also a call to cascading-cut, so this scheme can also be used to determine the possible pre- and postconditions of cascading-cut. The states are black, branches orange and executed code blue.

different places: The first cut (that is found in decrease key) and subsequent cuts (found in cascading cut). A control flow graph which illustrates the different calling contexts can be seen in Figure 4.3. The names of the states in the figure will be explained in the next few paragraphs. In this graph it can already be seen, why the decrease of the element is done after the last cut. If one would decrease the element before the first cut, it might already violate the heap invariant because if the decreased node has a parent, its value could be now smaller than the one of its parent. This would mean that there is an additional state the heap could be in before the first cut.

The first cut is special for two reasons: Firstly it cuts a node, whose value will be decreased, from its current position and puts it in the root list. This might mean that there is a new minimum in the root list after the priority is decreased, which leads to a change in the heap pointer. So after the first cut it needs to be tracked, that the to be decreased element is in the root list. This can be very easily expressed with the following expression:

```
dec_addr ∈ set root_list
```

Secondly the precondition of the first cut is the intact heap invariant, while for all subsequent cuts the heap is in an intermediate state, which is has to be converted back into the intact state by cascading-cut. This intermediate state arises from the fact that the parent of the previously cut node now violates the the child rank invariant (which was defined in Section 2.3), since its rank (the length of its child list) might now be one less than allowed. This can be expressed with the following expression (the change compared to the normal child rank invariant is bold):

```
definition child_rank_invariant_exc where
"child_rank_invariant_exc aset nm exc ≡
   ∀ p ∈ aset. ∀ i ∈ {0..<length (abs_fib_chldrn nm p)}.
      abs_fib_rnk nm (abs_fib_chldrn nm p ! i)
      ≥ i - of_bool (abs_fib_mrk nm (abs_fib_chldrn nm p ! i))
           - of_bool (exc = (abs_fib_chldrn nm p ! i))"
```

So now there are three heap invariants: the intact version, the version where the heap invariant is intact, but where it is known that the element to be decreased is in the root list, and the version where the child rank invariant might be violated for the parent (and where it is also known that the element to be decreased is in the root list). The second mentioned invariant will be called *root (exception) invariant*, since while there is no exception to the invariant yet, the decrease of the element might create one, as the heap pointer might not point to the node with minimum priority anymore. The third mentioned invariant will be called *rank (exception) invariant*. These names correspond to the states shown in Figure 4.3. Since as mentioned the structural components (the assertion part and the structural invariant) are the same for all of the invariants, only the logical part of the two latter invariants is shown here (the changes compared to the intact logical invariant are bold):

```
definition fib_heap_root_exc_logic_inv where
"fib_heap_root_exc_logic_inv aset nm rl val_map root\_exc ⟷
   fib_heap_logic_inv aset nm rl val_map ∧
   root_exc ∈ set rl"

definition fib_heap_rank_exc_logic_inv where
"fib_heap_rank_exc_logic_inv aset nm rl val_map root_exc rank_exc ⟷
   root_list_order nm rl ∧
   tree_order aset nm ∧
   rank_invariant aset nm ∧
   child_rank_invariant_exc aset nm rank_exc ∧
   matching_values nm aset val_map ∧
   root_exc ∈ set rl
```

As described above (and shown in Figure 4.3) there are four possible Hoare triples for cut: For the first cut the precondition is always the intact heap, but there are two possible post conditions. If the parent is already in the root list, the cut generates a heap in the root exception state. Otherwise the postcondition is the rank exception invariant, with the parent of the cut node being the possible exception. These postconditions can also occur for the subsequent cuts, but here the precondition is always the rank exception invariant. As the lemmas for the cases are very similar, only the lemma for the last case is shown. Here the Hoare triple for the whole heap invariant is used:

```
lemma fib_cut_rule_rank_rank:
assumes
   "chld ∈ aset"
   "abs_fib_prnt nm chld = (Some prnt)"
   "abs_fib_prnt nm prnt = (Some gp)"
   "fib_heap_struc_inv aset nm rl"
   "fib_heap_rank_exc_logic_inv aset nm rl val_map root_exc chld"
shows
   "<fib_heap_assn aset nm rl heap_ptr>
    fib_cut heap_ptr chld prnt
    <λ _. fib_heap_inv_rank_exc aset (abs_fib_cut nm chld prnt) (rl@[chld]) val_map
                                heap_ptr root_exc prnt>"
```

The assertion and structural parts were already proven. Only the proof for the logical invariant is missing. The most difficult part is the proof for the violated child rank invariant, since there are three cases to consider: For the child list from which the cut element was removed it needs to be shown that removing an element from a child list does not violate the child rank invariant for the other nodes in that list, as the their list position may decrease while their rank stays the same. In the child list in which the parent of the cut element is located the parent might now violate the child rank invariant. All other elements are not affected.

With the four mentioned Hoare triples it is possible to verify cascading-cut.

4.3.6 Verification of Cascading-Cut

As can be seen above in Listing 4.4, cascading-cut is a simple, tail recursive function, that more or less only calls cut. For that reason, it seems that it should be quite easy to prove the correctness of cascading-cut as the Hoare triples for cut that are needed in the induction proof are already proven. The problem lies in the fact that it needs to be proven that an induction is actually allowed. The first, very simple idea for an induction scheme would be the following:

Base case *The parent is none (the current node is in the root list)*: cascading-cut does nothing

Induction Step *The parent is not none (but closer to the root list)*: By knowing what cascading-cut does for the parent (the induction hypothesis), one can locally argue what cascading-cut does for the current node

This scheme is does not work due to mismatching preconditions. In the above mentioned base case, cascading-cut is just an elaborate noop. Here the first cut, that is executed before cascading-cut is called, already was enough to return the heap in the root exception state (see Figure 4.3). As this means that the parent of the cut node is in the root list, cascading cut does nothing. In any other case (where the current node is not in the root list) the first cut returns a heap in the rank exception state. Here cascading cut has to transform a heap in the rank exception state into a heap in the root exception state. That means that there is a different precondition (root exception vs rank

exception). So two different Hoare triples need to be verified: one for the trivial base case, where cascading-cut is just a noop, and one for the case where the heap is transformed from a rank exception state into a root exception state. The induction scheme for the second mentioned Hoare triple looks as follows:

Base case 1 *The current node is not marked*: The current node is marked and with that the rank-exception is resolved, so the heap is now already in the root exception state.

Base case 2 *The parent is not none but the grand-parent is none*: Here a single cut is done before the noop cascading-cut is called. This can be easily verified with the rank → root cut Hoare triple.

Induction Step *Both the grand-parent and parent are not none*: First the rank → rank cut Hoare triple is used to show that the heap is still in a rank exception state after the cut (but now the parent is the rank exception) and then the induction hypothesis, which states that the cascading_cut call for the parent transforms the heap from a rank exception state into a root exception state, can be used.

Until now for every operation it was possible to explicitly state the state of the heap after each operation. This is not possible for the induction hypothesis mentioned above, as it is not known how many times cascading cut will be executed here, so the post condition must contain an existentially quantified abstract node map and root list.

There are now four cases to consider: the trivial noop base case, and the two induction base cases and the induction step for the rank → root case. Again the Hoare triples are not very difficult to state, so as an example only the Hoare triple for the induction step (where the induction Hypothesis is one of the assumptions) is shown:

```
lemma fib_cascading_cut_step_rule:
assumes
   "addr ∈ aset"
   "abs_fib_prnt nm addr = Some prnt"
   "abs_fib_prnt nm prnt = Some gp"
   "fib_heap_struc_inv aset nm rl"
   "fib_heap_rank_exc_logic_inv aset nm rl val_map root_exc addr"
   "abs_fib_mrk nm addr = True"
   "<fib_heap_assn aset (abs_fib_cut nm addr prnt) (rl @ [addr]) heap_ptr>
    fib_cascading_cut heap_ptr prnt
    <λx. ∃A nm rl. fib_heap_inv_root_exc aset nm rl val_map heap_ptr root_exc>"
shows
   "<fib_heap_assn aset nm rl heap_ptr>
    fib_cascading_cut heap_ptr addr
    <λ x. ∃A nm rl. fib_heap_inv_root_exc aset nm rl val_map heap_ptr root_exc>"
```

The proofs of the Hoare triples are not that hard as they mostly rely on the Hoare triples for cut. The biggest problem now is to prove the validity of the induction scheme. The chosen proof is somewhat Isabelle/HOL specific. Here the automatic generation of induction schemes for total functions is exploited. Generally, the function returning the

path that is generated by following the parent-pointer until a nullpointer is reached may not terminate, because there may be a circle. As the function is only called for elements in a heap, it is known that there will not be any circle, but it has to terminate for all possible inputs. To achieve this behavior, the set of possible parent addresses is tracked in a finite set (for which a special datatype exists in Isabelle/HOL). It is initiated to the address set of the heap. If the currently looked at address is in the set, it is prepended to the path and it is removed from the address set for the recursive call (if the current address has a parent). If the current address is not in the address set or it if does not have a parent, the function stops. This function is implemented as follows:

```
fun parent_pathf':: "('a fib_tree dll ref ⇒ 'a abs_fib_tree) ⇒
                     'a fib_tree dll ref fset ⇒ 'a fib_tree dll ref ⇒
                     'a fib_tree dll ref list" where
"parent_pathf' nm curr_aset curr_addr =
   (if curr_addr |∈| curr_aset
      then
         (if abs_fib_prnt nm curr_addr = None
          then [curr_addr]
          else curr_addr# parent_pathf' nm (curr_aset |-| {|curr_addr|})
                                           (the (abs_fib_prnt nm curr_addr)))
      else [])"
```

The operations encased in bars are just the set operations on finite sets. With an additional lemma, that shows that the cardinality of a finite set decreases if a contained element is removed, the unconditional termination of this function can be proven. As the used address set of the heap is not a finite set, another helper function `parent_pathf` is defined, which simply calls `parent_pathf'` with the address set converted to a finite set. If it is known that the given node map represents a heap, the functionality of `parent_pathf` can be expressed with two lemmas which greatly simplify the function:

```
lemma parent_path_base_case:
assumes
  "fib_heap_struc_inv aset nm rl"
  "curr_addr ∈ aset"
  "abs_fib_prnt nm curr_addr = None"
shows
   "parent_pathf nm aset curr_addr = [curr_addr]"


lemma parent_path_inductive_step:
assumes
  "fib_heap_struc_inv aset nm rl"
  "addr ∈ aset"
  "abs_fib_prnt nm addr ≠ None"
shows
  "parent_pathf nm aset addr = addr#parent_pathf nm aset (the (abs_fib_prnt nm addr))"
```

The second lemma is proven with the help of two additional lemmas. The first one states that the path returned by the function is always a valid `parent_path` (see Section 4.3.3). The second one states that removing addresses occurring in the path returned by

`parent_pathf` from the starting set does not change the result. These are proven with the induction scheme generated for `parent_pathf'`.

With the shown lemmas it is possible to prove the basic induction scheme, which can be proven with a (list) induction over the result of `parent-pathf node-map aset addr`:

```
lemma parent_induct:
assumes
  "fib_heap_struc_inv aset nm rl"
  "curr_addr ∈ aset"
  "⋀ curr_addr. abs_fib_prnt nm curr_addr = None ⟹ P curr_addr"
  "⋀ curr_addr. abs_fib_prnt nm curr_addr ≠ None ⟹
                P (the (abs_fib_prnt nm curr_addr)) ⟹ P curr_addr"
shows
  "P curr_addr"
```

This is still not enough to prove cascading-cut, as it refers to both the current address and its parent and more importantly because each step of cascading-cut modifies the heap. The induction is still sound, as each cut does not change the parent path from the parent node to the root.

The above induction scheme could now be generalized, so that it would work even when the heap is modified in each inductive step, without changing the parent path. As cascading-cut is the only function that uses this induction scheme, a very specialized scheme only suited for the needs of cascading cut was proven instead. It looks as follows:

```
lemma parent_fib_cut_induct:
assumes
  "curr_addr ∈ aset"
  "abs_fib_prnt nm curr_addr = (Some prnt)"
  "⋀ x. x ∈ aset ⟹ distinct (abs_fib_chldrn nm x)"
  "fib_heap_struc_inv aset nm rl"
  "⋀ nm rl curr_addr prnt.
     abs_fib_prnt nm curr_addr = (Some prnt) ⟹
     abs_fib_prnt nm prnt = None ⟹
     P nm rl curr_addr prnt"
  "⋀ nm rl curr_addr prnt gp.
     abs_fib_prnt nm curr_addr = (Some prnt) ⟹
     abs_fib_prnt nm prnt = (Some gp) ⟹
     P (abs_fib_cut nm curr_addr prnt) (rl@[curr_addr]) prnt gp ⟹
     P nm rl curr_addr prnt"
shows
  "P nm rl curr_addr prnt"
```

The structure of the proof is similar to the previous induction scheme, though there is now an additional base case. A few more lemmas are required as well. Firstly, it needs to be shown that a cut doesn't change the parent path starting at the parent of the cut needs to be proven. Secondly, the four first assumptions also hold for the abstract map after the cut (and now for the address of the parent).

With this scheme it is now possible to verify the Hoare triple for the case where cascading-cut transfers the heap from a rank exception state into a root exception state:

```
lemma fib_cascading_cut_rule:
assumes
   "addr ∈ aset"
   "abs_fib_prnt nm addr = (Some prnt)"
   "fib_heap_struc_inv aset nm rl"
   "fib_heap_rank_exc_logic_inv aset nm rl val_map root_exc addr"
shows
   "<fib_heap_assn aset nm rl heap_ptr>
    fib_cascading_cut heap_ptr addr
    <λ x. ∃A nm rl. fib_heap_inv_root_exc nm rl val_map heap_ptr root_exc>"
```

In the other case cascading-cut is just a noop, so in that case the Hoare triple is trivial to prove.

4.3.7 Verification of Decrease-Key

With the two Hoare-triples for cascading cut, it is now possible to verify decrease key. This function is again split because cascading-cut has a post condition with an existentially quantified heap state. To be able to describe the changes of the actual decrease (and its effects on the invariants) one needs to assume that one knows the actual state of the heap. This step is hard to to in the middle of the automation. For that reason, the first part is responsible for the cut, while the second part does the actual decrease. The Hoare triple for the first part is quite easy to prove. If the element is already in the root or when the decreased priority will still be bigger than the priority of the parent, then nothing is done (in the first part). Otherwise cascading cut is called. This case can be easily be verified with the already proven Hoare triples for cascading-cut and with the Hoare triples for the first cut. The Hoare triple for this case of `fib_decrease_key_cut` looks as follows:

```
lemma fib_decrease_key_cut_rule3:
assumes
   "dec_addr ∈ aset"
   "abs_fib_prnt nm dec_addr = (Some prnt)"
   "fib_heap_struc_inv aset nm rl"
   "fib_heap_logic_inv aset nm rl val_map"
   "new_key < abs_fib_vl nm prnt"
shows
   "<fib_heap_assn aset nm rl heap_ptr>
    fib_decrease_key_cut heap_ptr dec_addr new_key
    <λ x. ∃A nm rl. fib_heap_inv_root_exc aset nm rl val_map heap_ptr dec_addr>"
```

So now the second part of decrease-key, where the priority is actually decreased, needs to be verified. Here there are two main cases to consider: either the element to be decreased is not smaller than the value of the parent or the element is in the root list (either through a cut or because it was already there). The latter case can again be split, because the decreased value may now be smaller than the old minimum of the heap. For all cases it holds that changing the value of an element does not change the structural invariant of the heap. The same holds for the logical invariant if the node to be decreased is in the root list and the heap minimum is still smaller than the decreased element:

```
lemma dec_val_logic_inv_root1:
assumes
   "fib_heap_logic_inv_root_exc aset nm rl val_map dec_addr"
   "fib_heap_struc_inv aset nm rl"
   "new_key ≤ val_map dec_addr"
   "abs_fib_vl nm (hd rl) ≤ new_key"
shows "fib_heap_logic_inv aset (change_vl nm dec_addr new_key) rl
                          (fun_upd val_map dec_addr new_key)"
```

The case where the new value is smaller than the heap minimum is more interesting, because in the heap invariant the first element of the abstract root list must always be the minimum. So the abstract list must be rotated to accommodate this. For the assertion part the rotation lemmas shown for the circular linked list come in useful. The structural invariant does not depend on the root list order, so only the logical invariant is affected. The two following lemmas proofs that decreasing the priority to a value smaller than the old minimum and rotating the root keeps the logical invariant intact:

```
lemma dec_val_logic_inv_root2:
assumes
   "fib_heap_logic_inv_root_exc aset nm (rl1@dec_addr#rl2) val_map dec_addr"
   "fib_heap_struc_inv aset nm (rl1@dec_addr#rl2)"
   "new_key ≤ val_map dec_addr"
   "new_key ≤ abs_fib_vl nm (hd (rl1@dec_addr#rl2))"
shows "fib_heap_logic_inv aset (change_vl nm dec_addr new_key) (dec_addr#rl2@rl1)
                          (fun_upd val_map dec_addr new_key)"
```

The last lemma that has to be proven for the logical invariant, is the case where the decreased element is not cut as the parent is still smaller. Since this is trivial, the lemma is not shown here.

With these lemmas for the Boolean parts the Hoare triples for the three mentioned cases can be easily proven. As an example the hoare triple for the case where the decrease node becomes the new minimum is shown:

```
lemma fib_decrease_key_decrease_rule_new_root:
assumes
  "dec_addr ∈ aset"
  "new_key ≤ val_map dec_addr"
  "fib_heap_struc_inv aset nm (rl1@dec_addr#rl2)"
  "fib_heap_logic_inv_rank_exc aset nm (rl1@dec_addr#rl2) val_map dec_addr"
  "abs_fib_vl nm (hd (rl1@dec_addr#rl2)) > new_key"
 shows
  "<fib_heap_assn aset nm (rl1@dec_addr#rl2) heap_ptr>
   fib_decrease_key_decrease heap_ptr dec_addr new_key
   <λ x. fib_heap_inv aset (change_vl nm dec_addr new_key) (dec_addr#rl2@rl1) (fun_upd
     val_map dec_addr new_key) x>"
```

With these Hoare triples together with the Hoare triples of the cut part of decrease key the whole invariant can be proven:

```
lemma fib_decrease_rule:
assumes
   "dec_addr ∈ aset"
   "new_key ≤ val_map dec_addr"
shows
   "<fib_heap_inv aset nm rl val_map heap_ptr>
    fib_decrease_key heap_ptr dec_addr new_key
    <λ ptr. ∃_A nm rl. fib_heap_inv aset nm rl (fun_upd val_map dec_addr new_key) ptr>"
```

4.3.8 Verification of Other Heap Functions

To verify that the introduced invariant is completely sound one would need to verify all other heap functions (and also their running times) and preferably use them in another algorithm, like Djkstra's algorithm [6]. Due to time constraints this was not done in this work, but at least a few of the other heap functions could be proven. The functions are make-heap, heap-singleton (creating a heap with only a single element), get-min, and heap-union. The first and probably most important ones out of these are the functions that create a new heap. These show that the invariant is achievable and not equivalent to `false`. The first way to create a heap is the function that creates an empty heap:

```
definition make-heap where
"make-heap ≡ return None"

lemma make_heap_rule:
"<emp> make-heap <λ ptr. fib_heap_inv {} (λ x. undefined) [] (λ x. undefined) ptr>"
```

The second function that creates a new heap, creates it with a single given value stored in it. This is theoretically not necessary, as it can be implemented by creating an empty heap and then adding a single element, but as the add function was not implemented as part of this work, this function is quite useful:

```
definition heap_singleton where
"heap_singleton val ≡ do {
   ptr ← cdll_singleton (Fib_Node 0 False val None None);
   return (the ptr, ptr)
}"

lemma heap_singleton_rule:
shows
  "<emp>
   heap_singleton val
   <λ (addr, ptr). fib_heap_inv {addr} (fun_upd (λ x. undefined) addr
                                (Abs_Fib_Node 0 False val [] None)) [addr]
                                (fun_upd (λ x. undefined) addr val) ptr>"
```

Note that this function returns the heap pointer and the address of the newly allocated element. The latter is not necessary as the value of the heap pointer is the address of the new element, but emulates the behavior of the add function which has to return the

address of the newly allocated element so that the user can use it in his implementation of the address-value map.

The next function that was verified is get-min, which is also quite simple as it just returns the value of stored at the element of the heap pointer. The proof that the returned minimum is actually the smallest element, can be done with an induction over the recursive set that refines the child relation (see Section 4.3.3).

```
definition get_min where
"get_min heap_ptr = do {
   min_el ← !the heap_ptr;
   return (fib_get_val min_el)
}"

lemma get_min_rule:
fixes heap_ptr :: "('a::{linorder, heap}) fib_tree dll ref option"
assumes "heap_ptr ≤ None"
   "fib_heap_struc_inv aset nm rl"
   "fib_heap_logic_inv aset nm rl val_map"
shows
   "<fib_heap_assn aset nm rl heap_ptr>
   get_min heap_ptr
   <λ min. ↑(min = val_map (the heap_ptr) ∧ (∀ x ∈ aset. min ≤ val_map x))
             * fib_heap_inv aset nm rl val_map heap_ptr>"
```

The next and last heap function that was verified as part of the work for this thesis is heap-union. Algorithmically it is quite easy to implement, but the used heap abstraction as before makes the verification a bit more complicated. Before describing these problems it makes sense to look at the Hoare triple of the function (which is preceded by the implementation):

```
definition heap_union where
"heap_union heap_ptr1 heap_ptr2 = do {
   cdll_append_cdll heap_ptr1 heap_ptr2;
   if heap_ptr1 = None then
   return heap_ptr2
   else do {
      if heap_ptr2 = None then
      return heap_ptr1
      else do {
         min1 ← !the heap_ptr1;
         min2 ← !the heap_ptr2;
         if (fib_get_val min1) ≤ (fib_get_val min2) then
         return heap_ptr1
         else
         return heap_ptr2
      }
   }
}"
```

```
lemma heap_union_rule:
assumes "fib_heap_struc_inv aset1 nm1 rl1"
  "fib_heap_logic_inv aset1 nm1 rl1 val_map1"
  "fib_heap_struc_inv aset2 nm2 rl2"
  "fib_heap_logic_inv aset2 nm2 rl2 val_map2"
shows
  "<fib_heap_assn aset1 nm1 rl1 heap_ptr1 * fib_heap_assn aset2 nm2 rl2 heap_ptr2>
   heap_union heap_ptr1 heap_ptr2
   <λ ptr. ∃A rl. fib_heap_inv (aset1 ∪ aset2) (override_on nm1 nm2 aset2) rl
                               (override_on val_map1 val_map2 aset2) ptr>"
```

The root list is existentially quantified because the minimum of the merged heap is the smaller of the two minimums of the original heaps. This means that the order of the root list might have to be changed (as it was the case for decrease-key when the decreased element was the new minimum of the heap). There might be a slight problem with this Hoare triple, which is the union of the address sets. It might make sense to add a pure assertion to the postcondition that asserts that the two address sets do not overlap. This is required by the precondition (since the address sets are the footprints of the heaps), but this information is currently lost in the postcondition.

The main difficulty of this proof was dealing with the updates of the abstract map. As each heap brings its own abstract map, these have to be merged with the help of `override_on`, which takes a map and overrides each value that is stored at an address found in the given address set (the last parameter) with the value stored at the same address in the second map. The difficulty lay in showing that the merged (and changed) abstract map still represents a valid heap. This can be proven with the help of the footprints, which is used to show that `override_on` does not overwrite any values of one heap with values from the other heap.

The two heap functions that were not implemented (if delete is ignored, as it can be implemented with decrease-key and delete-min) are insert and delete-min. The functionality of insert can actually be implemented by creating a new heap with the element to be added using heap-singleton and then merging that heap with the original heap with heap-union. As the directly implementing heap-insert is more efficient, it should still be implemented as a standalone function as well. The problems with implementing heap-insert and of course delete-min, which, together with decrease-key, is one of the two most complex heap operations, is discussed in Chapter 5, where it is outlined how work done for this thesis can be continued.

4.3.9 Summary

With the presented map-based approach it was possible to verify parts of the Fibonacci heap with separation logic. To summarize the achieved work, the Hoare triples with the invariant that is exposed to clients that use the heap functions can be seen in Listing 4.6.

There is still some work left to do to finish the verification of the Fibonacci heap with this approach, but with the verification of decrease-key and with the work already done by Stüwe in his thesis [17], the most difficult parts should now be done. The exact list

```
lemma heap_empty_rule_user:
   "<emp> heap_empty <λ ptr. fib_heap_user_inv {} λ(x. undefined) ptr>"

lemma heap_singleton_rule_user:
   "<emp>
    heap_singleton val
    <λ (addr, ptr). fib_heap_user_inv {addr} (fun_upd (λ x. undefined) addr val) ptr>"

lemma get_min_rule_user:
assumes "heap_ptr ≠ None"
shows
   "<fib_heap_user_inv aset val_map heap_ptr>
   get_min heap_ptr
   <λ min. ↑(min = val_map (the heap_ptr) ∧ (∀ x ∈ aset. min ≤ val_map x))
    * fib_heap_user_inv aset val_map heap_ptr>"

lemma heap_union_rule_user:
   "<fib_heap_user_inv as1 val_map1 heap_ptr1
     * fib_heap_user_inv as2 val_map2 heap_ptr2>
    heap_union heap_ptr1 heap_ptr2
    <λ ptr. fib_heap_user_inv (as1 ∪ as2) (override_on val_map1 val_map2 as2) ptr>"

lemma fib_decrease_key_rule_user:
assumes
   "dec_addr ∈ aset"
   "new_key ≤ val_map dec_addr"
shows
   "<fib_heap_user_inv aset val_map heap_ptr>
    fib_decrease_key heap_ptr dec_addr new_key
    <λ ptr. fib_heap_user_inv aset (fun_upd val_map dec_addr new_key) ptr>"
```

Listing 4.6: An overview over the proven Hoare triples in this thesis using the invariant that should be used by callers of these function (see Section 4.3.3)).

that should still be done are detailed in the next chapter on "Future Work" (Chapter 5).

The map based approach that was used to achieve this is not the standard approach for verifying data structures in separation logic, because here data structures are normally defined with recursive assertion generating functions as was done in Reynolds original paper on separation logic [16]. One reason for that is probably that most data structures either have a very clear traversal direction (like normal tree data structures) or are relatively simple (like lists). There is one paper by Hobor and Villard [10] on pointer aliasing in separation logic where a map based approach is used to generate the assertion for a graph structure, but here this assertion is only a side note. It is only used to proof some lemmas and not as the main representation of the structure.

One key problem of the map based approach is that one loses the inductive structure that is given by the recursive generating functions, which makes dealing with the assertion more complicated. In the presented approach for the heap, this lead to an additional structural invariant, which is not needed for the approach based on a functional data structure (see Section 4.2.2). This invariant has to be expressed with a Boolean predicate on the map and with that it lies outside of the scope of separation logic (whose key feature is the separating conjunction).

As using the map-based approach for the verification of complex data structures seems to be somewhat uncharted territory there is probably a lot more research to be done on this approach. Some ideas for possible improvements are described in the next chapter on "Future Work" (Chapter 5).

4.4 Path Based Approach

Another approach, that was considered very shortly, can be seen as an intermediate approach between the structured one, where the heap is represented by a functional data structure, and the unstructured one, where just a node map is used. Here the idea was to represent the heap as a composition of the unique paths to the nodes. The idea why this representation might be useful was that a big problem with the approach using a functional data structure, was that it was hard to split apart. In contrast, each path is just a simple list, which can easily be split. Additionally, compared to the unstructured approach, the paths might make reasoning about the structure of the heap simpler.

This approach was not considered for very long as there is an issue with its representation in separation logic. The best idea for a path assertion would be that it describes all lists the path goes through. As each path describes only a part of the heap, one needs to use the construction `path_assn` $*$ `true`, which is the intuitionistic version of the path assertion. It is true for any heap which contains the path (and not just the heap that represents only that path). Combining the assertions for all paths to all nodes in the heap could be done with the $\wedge_A$ operator, so the heap assertion would look something like this: `path_to_node1 * true` $\wedge_A$ `path_to_node2 * true` $\wedge_A$ `....` This assertion describes the whole Fibonacci heap, but the assertion would then hold for all memory heaps which contain the heap. To describe the exact heap and nothing else one would need to add an-

other term, which contains all points-to assertions for all addresses. It would need to look similar to the heap assertion for the map-based invariant. This means that this approach would only makes sense if dealing with the path assertions was easier than dealing with the Boolean structural invariant used in the map-based approach. This was deemed very unlikely. One reason for this assessment is that the Hoare triple automation does not work well with the $\wedge_A$ operator. So that it can apply the read and write rule one would need to extract the points-to assertion from all path assertions where it occurs explicitly using the following rule:

```
lemma and_assn_star_distrib:
assumes "weak_fixed_footprint_t A as"
shows "A * B * true ∧A A * C * true = A * (B * true ∧A C * true)"
```

And after the change the separated points-to assertion would have to be put back into the paths using the same rule. Due to these mentioned complications, this approach was not pursued further.

5 Future Work

5.1 Finishing the Map Based Approach

As mentioned, there are still things left to do before the heap is fully verified with the map-based approach. First it should be checked, how the existing proofs can be further optimized. One big fundamental change, that should be looked at, is using the map datatype as it is implemented in Isabelle/HOL for the abstract map (as mentioned in the beginning of Section 4.3). There is also the possibility that using a graph library for the implementation of the path related definitions could simplify some proofs. Before changing the paths, it may be a sensible idea to change the implementation of the reachability concept. Here the recursive set could be replaced with a reflexive transitive closure over the child relation.

Then the missing functions heap-insert and delete-min need to be verified. They both offer unique challenges. The verification of insert differs from the other heap functions, because it allocates a new node that is added to the already existing heap. For all of the proofs of the Hoare triples it was essential to show that the address of the node to be changed occurs only in a single list at a single index. In heap-insert this address is not known before the function is executed, as it is newly allocated. That means that it cannot be used in manual proofs before the proof automation processed the allocation. The problem is that the automation generally badly mangles the proof goals, so a manual proof should not be done after the automation has run. The solution is the approach that was already used previously to deal with problems in the automation: one should split the allocation part of the function from the rest of it and define manual Hoare triples for both parts. The postcondition for the allocation should look like this: `<`λ `new_addr. [heap_inv] * new_addr` $\mapsto_r$ `new_el`. This should be easily provable. The precondition of the main part of heap-insert then is `<[heap_inv]` $*$ `new_addr` $\mapsto_r$ `new_el>`. Here the newly allocated address can be used from the start in the proofs and with that heap-insert should be easily provable.

The function delete-min is already quite challenging due to its algorithmic complexity, but there is also another problem that is due to the used formulation of the invariant. During delete-min, Fibonacci trees of the same rank are linked together, meaning that the tree with the smaller value stored in the root is added to the child list of the other tree. The problem is that the current version of the invariant for the heap is not easily splittable into invariants for single trees. It is possible to define the invariant of a single tree, by using a singleton root list `[prnt]`, but it is currently unnecessarily verbose:

```
↑(fib_heap_struc_inv aset nm [new_child] ∧ fib_heap_logic_inv aset nm [prnt] val_map)
* children_assn aset nm [prnt]
```

Note that the address set needs to be restricted to the addresses that are only occupied by the tree itself. As Stüwe already proved the correctness delete-min function with his approach [17], his work should be consulted before starting with the verification. .

After the missing functions are verified, the running time bounds of the heap should be next. For decrease-key (which should have an amortized $O(1)$ time complexity) the most difficult part is the running time of cascading-cut (as the rest of decrease-key already has a $O(1)$ running time). Here the change in the potential (which takes the marked nodes and the number of elements in the root list into account) used for the amortized analysis negates the time taken by the operations in cascading-cut [5]. For the running time proofs for the other functions it should be possible to reuse (most of) the work done by Stüwe [17].

In addition to the running time verification the heap functions should be used to implement and then verify a more complex algorithm, to test the usability of the heap invariant. A prime example that makes use of decrease-key is Dijkstra's-algorithm [6].

5.2 General

There are also some more general ideas how the map-based approach could be improved. The first thing to look at is probably whether it makes sense to scrap the abstract map and directly define the Fibonacci heap invariant on the heap memory itself. This would require a change in the Imperative HOL as accessing the heap from the pre- and postconditions is currently not possible. This might allow a better integration between the Boolean parts of the invariant and the footprint of the assertion, though this is just speculation. Generally, it should be looked at how the Boolean predicates can be better integrated with separation logic. The most interesting candidate for this integration is probably the reachability constraint. As reachability is a fundamental concept for all pointer based structures, there might already be some work on this topic that could also be useful here. As the search for related work was mostly focused on finding approaches that also verify a complex data structure like the Fibonacci heap or at least do not use recursive predicates, there might still be a some concepts found in other approaches that could be used to improve the presented approach.

Additionally, one could look into other approaches. While the functional (Section 4.2) and the path-based approach (Section 4.4) were deemed to complex, there might be a way to make them work, that was not found during the work for this thesis. There might also be completely different approaches that might work.

6 Conclusion

In this thesis, we showed that is possible to verify the Fibonacci Heap decrease-key operation with separation logic. Since Stüwe [17] already proved the functional correctness of the other heap functions, this means that the complete heap functionality was proven with the help of separation logic, but as the used approaches are not directly compatible, there is still some work left do to before there is a full set of verified Fibonacci heap functions that can be used together.

The key difficulties in the verification of decrease-key lay in the complex structure of the heap in combination with the fact, that an arbitrary element can be directly modified by decrease key. This seems to necessitate the explicit storage of the addresses of the heap nodes in the invariant. The mentioned points make using a recursive representation of the heap for the verification, which is the standard approach of defining invariants in separation logic, seemingly impractical. So for the verification a different approach was chosen, where the heap is represented as a map from addresses to heap nodes. While this approach does not seem to be the best match for separation logic, as it requires more Boolean predicates, it might still be the best way of verifying complicated imperative structures like the Fibonacci heap with separation logic, but further research has to be done on whether this is actually the case.

Bibliography

[1] R. Bird. *Introduction to functional programming using Haskell*. 2nd ed. London: Prentice Hall, 1998.

[2] *Boost.Intrusive*. URL: https://www.boost.org/doc/html/intrusive.html.

[3] G. S. Brodal and C. Okasaki. "Optimal Purely Functional Priority Queues." In: *Journal of Functional Programming* 6 (1996), pp. 839–857.

[4] L. Bulwahn, A. Krauss, F. Haftmann, L. Erkök, and J. Matthews. "Imperative Functional Programming with Isabelle/HOL." In: *Theorem Proving in Higher Order Logics*. Ed. by O. A. Mohamed, C. Muñoz, and S. Tahar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 134–149. ISBN: 978-3-540-71067-7.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. 3rd ed. 2009.

[6] E. W. Dijkstra. "A note on two problems in connexion with graphs." In: *Numerische Mathematik* 1 (1959). DOI: 10.1007/BF01386390.

[7] *Fibonacci Heap Imperative HOL*. URL: https://gitlab.lrz.de/griebel/fibonacci-heap-imperative-hol.

[8] M. L. Fredman and R. E. Tarjan. "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms." In: *J. ACM* 34.3 (July 1987), pp. 596–615. ISSN: 0004-5411. DOI: 10.1145/28869.28874.

[9] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming." In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259.

[10] A. Hobor and J. Villard. "The Ramifications of Sharing in Data Structures." In: *SIGPLAN Not.* 48.1 (Jan. 2013), pp. 523–536. ISSN: 0362-1340. DOI: 10.1145/2480359.2429131.

[11] P. Lammich and R. Meis. "A Separation Logic Framework for Imperative HOL." In: *Archive of Formal Proofs* (Nov. 2012). http://isa-afp.org/entries/Separation_Logic_Imperative_HOL.html, Formal proof development. ISSN: 2150-914x.

[12] *Monad*. URL: https://wiki.haskell.org/Monad#do-notation.

[13] T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science 2283. Springer, Berlin, Heidelberg, 2002. DOI: 10.1007/3-540-45949-9.

[14] P. W. O'Hearn. "Resources, concurrency, and local reasoning." In: *Theoretical Computer Science* 375.1 (2007), pp. 271–307. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2006.12.035.

[15] L. Paulson. "The Foundation of a Generic Theorem Prover." In: *Journal of Automated Reasoning* 5 (Jan. 1999). DOI: 10.1007/BF00248324.

[16] J. C. Reynolds. "Separation logic: a logic for shared mutable data structures." In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. 2002, pp. 55–74.

[17] D. Stüwe. "Verification of Fibonacci Heaps." Master's Thesis. Technische Universität München, Apr. 2019.

[18] M. M. Wenzel. "Isabelle/Isar — a versatile environment for human-readable formal proof documents." PhD thesis. Munich: Technische Universität München, 2002.