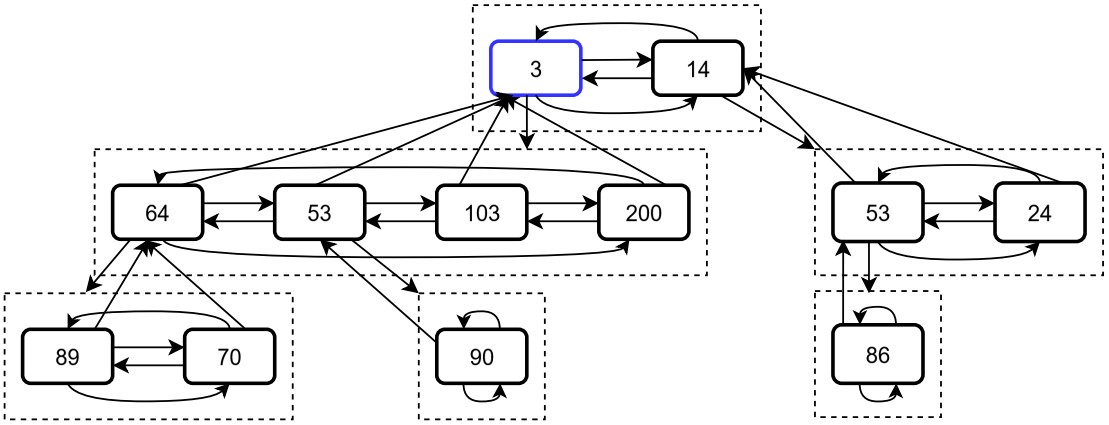
CHALMERS
UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG



A System-Level Implementation of a Verified Fibonacci Heap as Priority Queue

A System-Level Verification of a Forest of Trees

Master's thesis in Computer science and engineering

Tobias Treuheit

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2026

Master's thesis 2026

A System-Level Implementation of a Verified Fibonacci Heap as Priority Queue

A System-Level Verification of a Forest of Trees

Tobias Treuheit

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2026

A System-Level Implementation of a Verified Fibonacci Heap as Priority Queue
A System-Level Verification of a Forest of Trees
Tobias Treuheit

© Tobias Treuheit, 2026.

Supervisor: Magnus Myreen, Computer Science and Engineering
Examiner: Peter Damaschke, Computer Science and Engineering

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Figure of a small Fibonacci Heap. See Chapter 2 for more detail.

Typeset in LaTeX
Gothenburg, Sweden 2026

A System-Level Implementation of a Verified Fibonacci Heap as Priority Queue
A System-Level Verification of a Forest of Trees
Tobias Treuheit
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This thesis presents the functional correctness of the insert, meld, and extract minimum operation of the Fibonacci heap in the system-level programming language Pancake. The verification is carried out with the interactive theorem prover HOL4 and Pancake's associated separation logic framework. Our approach is defined from a structural design of the Fibonacci heap and a separation of the verification task over multiple levels. These verification levels start at the logical reasoning about the Fibonacci heap and end in correctness properties stated in terms of Pancake's operational semantics. We target the Pancake language because it is supported by a verified compiler in HOL4.

Keywords: Computer, Science, Computer Science, Formal Verification, Verification, Fibonacci Heap, Separation Logic, HOL4, End-To-End Verification

Acknowledgements

Firstly, I would like to thank my family for their support during my time at university. I would also like to thank my supervisors at the University of Bamberg and Chalmers for our awesome teamwork. In particular, I would like to thank Magnus Myreen for introducing me to verification with HOL4. Finally, I would also like to thank the HOL4 community for their help with questions on how to use record types in proofs.

Tobias Treuheit, Gothenburg, 2026-06-12

Contents

List of Figures

1

Introduction

Recently, system-level verification became very interesting due to the novel system-level programming language called Pancake. The Pancake language and its compiler live inside the interactive theorem prover HOL4 and thus allow end-to-end verification. Actually, studies of end-to-end verification are rare and many system-level verifications do not imply executable code [1]. However, with Pancake it is possible to verify an algorithm and prove the correctness down to the compiled machine code.

In particular, Pancake is a system-level programming language similar to the well-known C programming language. The language supports pointers, machine words, and struct-like data structures. Recently, it has been used to study the verification of device drivers [1], since these are a critical infrastructure of operating systems.

However, the authors identify that studies only verify easy device drivers, usually with non-foundational verification approaches. Indeed, verification of a more complex data structures is difficult, since these may require explicit formal reasoning about the memory layout. Such proofs can be constructed with Pancake since the language and its compiler are integrated into the interactive theorem prover HOL4.

Therefore, this master thesis studies the foundational verification of a complex data structure. The Fibonacci heap seems suitable as it composes lists and trees but also has a numerically ordering relation. So, it integrates multiple data structures and is a partially sorted structure. For this study, we choose to refine and verify the Fibonacci heap as an example data structure.

The data structure of the Fibonacci heap can be seen as an efficient priority queue and was originally discussed as such with the famous Dijkstra algorithm. Actually, the Fibonacci heap was designed to fit the needs of Dijkstra's algorithm.

The project focuses on two main operations of the Fibonacci heap – melding two heaps and extracting the minimum. Further, our project studies the system-level verification potential of Pancake and the construction of an end-to-end proof with Pancake.

The verification of Fibonacci heaps in on a memory level setting has been studied previously. Especially, two other projects implement and verify the Fibonacci Heap in ISABELLE/HOL [2, 3]. Both use a similar approach of representing their Fibonacci Heaps as function or map updates. In particular Stüwe focuses on the functional correctness of the extract minimum operation [2] and Griebel studies the correctness

of the decrease key operation [3].

In contrast, our approach focuses on the end-to-end verification with Pancake. Hence, we verify the functional correctness of our implementation but also reason about mutable memory in separation logic. Moreover, we prove the operational semantics of Pancake for our implementation. Therefore, we split the verification into different abstraction levels accordingly to ensure a clear outline of an end-to-end verification with Pancake.

Our study focuses on the following goals:

G1: A structural separation of different verification levels for the verification of the Fibonacci heap.

G2: Verification of the functional correctness for a system-level Fibonacci heap implementation.

G3: The showcase of a formal end-to-end verification for Pancake.

Finally, we give a short overview of our chapters. The following Chapter 2 describes the background of this thesis by describing keywords, tools, and frameworks that are used to accomplish the goals. Afterward, Chapter 3 will focus on our system-level design of the Fibonacci heap. It will also explain the design of our verification approach and the separation of logical and heap verification. The explanation is followed by a Chapter 4 that shows our implementation of the corresponding operations on the heap. Then, Chapter 5 focuses purely on the verification of the operations. This chapter also includes one end-to-end verification. Finally, in the last Chapter 6 we conclude the thesis with a summary, a discussion about lessons learned and related work.

AI Disclaimer: In this thesis and project, we have used Artificial Intelligence for:

- debugging errors of HOL4 at the start of the project (ChatGPT)
- improving the corresponding latex template (ChatGPT)
- helping with grammar and sentences for the presented text (DeepL)

The original text and all artifacts of the project were crafted manually.

2

Background

This chapter describes the technical concepts used throughout this thesis. It starts with the concept of the Fibonacci heap and continues with the Dijkstra algorithm. Afterward, the chapter describes the system-level programming language Pancake and its connected proof assistant HOL4 in more detail. Finally, this chapter closes on a short explanation of separation logic and its corresponding framework in HOL4.

2.1 Fibonacci Heap

The Fibonacci heap (FH) is a heap data structure that can be visualised as a list of trees. Each tree node may hold children of the same shape as itself (see Figure 2.1). This structure is also known as a rose tree.

The FH is a special kind of rose tree. It includes a partially sorted relation that ensures the top element is the smallest or largest in the heap. In our case, this will always be the smallest element in the heap.

Furthermore, the shape of a single tree follows a particular pattern. Due to the way the FH's operations are designed, each tree must hold a specific number of elements related to the Fibonacci numbers. In particular, each tree must be at least of the size of the rank + 2 Fibonacci number. The rank are the number of direct children of the tree. This is where the name *Fibonacci heap* comes from.

The core of the FH is a cycling double-linked list (CDLL), also known as the root list. Each element in the root list contains values, at least one of which is used for the partially sorted relation. Additionally, each element contains its own CDLLs indicating its children. These children are composed in the same way as their parents. As each CDLL element resembles a tree, they are called Fibonacci trees (FTs).

Note that the FH's root list holds multiple FTs. Therefore, one could say that the FH is a forest of trees (see Figure 2.2) [4].

The FH originates from the Binomial heap. Its operations are more complex but ensure overall a better amortized time complexity.

For instance, the *insert key* and *melding* of two heaps to one heap are designed to execute in constant time. Moreover, the *decrease key* operations is constructed to have an amortized time bound of constant time.

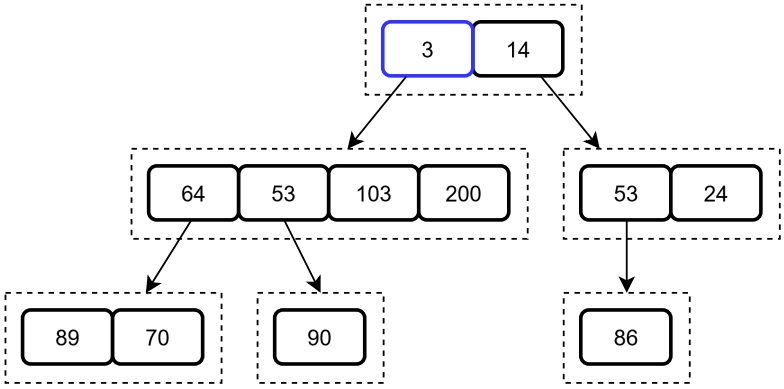


Figure 2.1: This figure depicts a Fibonacci heap with two elements in the root list. The blue colored element is the current top element of the heap.

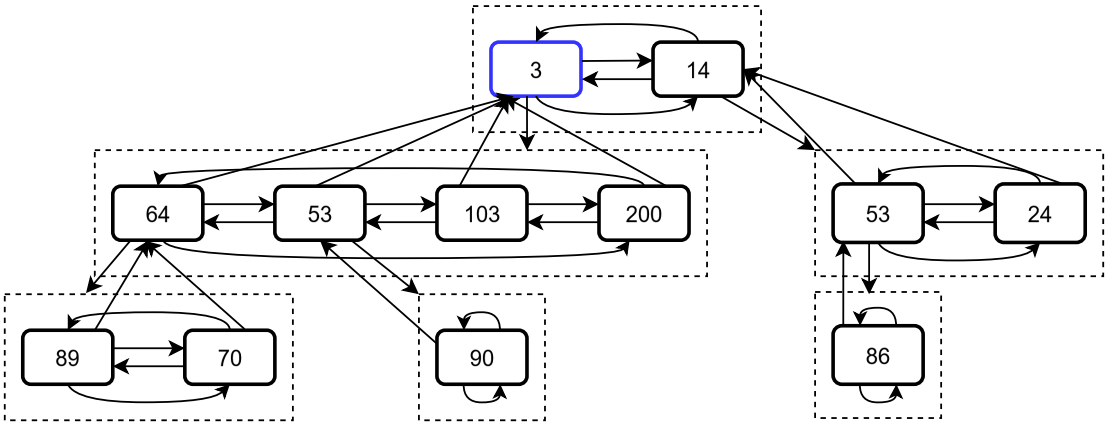


Figure 2.2: This figure highlights the complexity of the FH. It focuses on the connections between each tree.

The only operation that does not have an amortized constant time bound is the *extract minimum* operation. Instead, its amortized time bound is $\mathsf{O}(\log n)$ where n is the number of elements in the heap. During the operation, *extract minimum* needs to find the new minimum and reconstructs the trees in the root list to be of different rank. The reconstruct enables the amortized performance of the *extract minimum* operation. However, the worst case time complexity of the operation is $\mathsf{O}(n)$.

In general, the FH corresponds to an efficient priority queue when the *decrease key* operation is called more often than the *extract minimum* operation.

2.2 Dijkstra Algorithm

The Dijkstra algorithm is a well-known method for finding the shortest spanning tree [5]. In order to achieve this, the algorithm keeps track of all nodes in the graph and classifies them as discovered, reachable or unreachable. The spanning tree then grows during the execution of the main loop by selecting the shortest possible path

to a reachable node. This node is then added to the set of discovered nodes.

Reachable nodes are usually stored in an efficient priority queue because the efficiency of the Dijkstra algorithm depends on that of the underlying data structure.

To ensure progress in the algorithm, the paths from the newly discovered node are explored and added to the priority queue in one of two circumstances:

1. The new node yields paths to unreachable nodes. These nodes then become reachable and are added to the queue.
2. The path to a reachable node is shorter when using the new node. In this case, the reachable node is updated inside the priority queue to the new value.

The original paper on FHs already highlights the composition of Dijkstra algorithm with the FH [4]. This is because the Dijkstra algorithm uses the *decrease key* operation at least as often as the *extract minimum* operation. As highlighted in the previous section, the amortized time bound of the *decrease key* operation is its beneficial property.

2.3 Pancake

Pancake is a new system-level programming language [6]. It originates from CakeML. CakeML is a high-level programming language that implements a subset of Standard ML, and comes with a verified compiler. In fact, both CakeML and Pancake implement the same verified backend for generating machine code. However, unlike the high-level programming language CakeML, Pancake is a system-level language with direct memory access and a minimal type system. In particular, Pancake uses pointers, machine words and structures as valid types (see Figure 2.4). Actually, the Pancakes type system is similar to the popular low-level programming languages C and Rust. An example program is shown in Figure 2.3.

Furthermore, the Pancake compiler is implemented and verified with HOL4. This simplifies the foundational verification of Pancake programs and makes it ideal for implementing and verifying system-level dependencies, such as device drivers.

2.4 HOL4

The HOL4 tool is an interactive theorem prover (ITP) that helps users construct proofs semi-automatically [7]. Users apply tactics to modify assumptions or goals using theorems, definitions or lemmas. The intention is to rewrite both so that they are identical, indicating that the goal has been proven. The interaction is directed by the user. The user selects which rules to apply, and HOL4 calculates the new assumptions or goal statement for the proof. Therefore, theorems and definitions are key to constructing proofs in HOL4. To improve the usability of HOL, the ITP comes with many theories. In this thesis, for instance, we will use library theories with standard definitions and theorems about natural numbers, lists, sets, finite

```
fun main() {
  var n = 0;
  !ldw n,@base;
  var r = fib(n);
  !stw @base, r;
  return 0;
}
fun fib(n){
  if (n <= 1) {
    return n;
  } else {
    var x = fib(n - 1);
    var y = fib(n - 2);
    return x + y;
  }
}
```

Figure 2.3: A Pancake code example. The code computes the nth Fibonacci number. The keyword *@base* is used to load and store a value from shared memory.

```
exp := Const word | Var string
     | Label string
     | Struct exp* | Field num exp
     | Load shape exp | LoadByte exp
     | Op binop exp* | Cmp cmp exp exp
     | Shift shift exp num | BaseAddr

prog := Skip | Break | Continue | Tick
     | Dec string exp prog | Return exp
     | Assign string exp | Store exp exp
     | StoreByte exp exp | Seq prog prog
     | If exp prog prog | While exp prog
     | Raise string exp
     | Call ret exp exp*
     | ExtCall string exp exp exp exp
```

Figure 2.4: The abstract syntax of Pancake.

maps and machine words, in order to construct our own theory about a possible FH design.

HOL4 is implemented in SML and supports higher-order logic (HOL). This includes data types, functions and theorems [8]. Therefore, HOL4 code resembles a high-level programming language, but has a different use case. Specifically, the intention is to use HOL to construct theorems and their associated logical theories rather than to execute code.

The structure of a theorem is always the same. It starts with a name and the logical statement to be shown. The user then uses tactics to modify the statement such that its correctness is obvious. This is achieved by expanding definitions or applying previous proof results. Figure 2.5 shows this work flow with a small example.

This paper will show many definitions and logical statements of HOL4. Hence a brief overview of its syntax.

The HOL is written in a functional programming style. For instance, the cons and append operator for lists is identical to functional programming languages like SML ($x :: xs$, $xs +\!\!+ ys$). However, we use multiple other operators. For instance, for lists we use list indexing $list(i)$ that retrieves a value at index i, and the list update operator $list(i \mapsto x)$ that updates the value at index i with x.

Furthermore, we use finite maps. Finite maps work like hash maps in programming languages and allow the insertion and deletion of elements ($m\langle k \mapsto v\rangle$, $m \setminus\setminus k$). Finite maps also support lookups ($\mathsf{lookup}\ m\ k$), reasoning about their domain $\mathsf{dom}\ m$ and union of multiple maps ($m_1 \uplus m_2$).

```
open arithmeticTheory listTheory;


Theorem transitivity_of_integers:
  !a b c.
    a ≤ b /\ b ≤ c
    ==>
    a ≤ c
Proof
  rpt strip_tac >>
  simp[]
QED
```

```
HOLLoadSendQuiet arithmeticTheory listTheory c
ompleted
Vim input /tmp/vimhol0Script.sml
val it =
   Proof manager status: 1 proof.
   1. Incomplete goalstack:
        Initial goal:
        ∀a b c. a < b ∧ b < c ⇒ a < c
   : proofs
Vim input /tmp/vimhol0Script.sml
OK..
1 subgoal:
val it =

    0.  a < b
    1.  b < c
   ------------------------------------
        a < c

   : proof
Vim input /tmp/vimhol0Script.sml
OK..

Goal proved.
 [..] ⊢ a < c
val it =
   Initial goal proved.
   ⊢ ∀a b c. a < b ∧ b < c ⇒ a < c: proof
```

Figure 2.5: A small example that shows verification with HOL4.

In rare occasions we use the set membership operation ($e \in s$), the disjointness relation of sets (disjoint s_1 s_2) and, finally, also function updates ($f(\!|x \mapsto z|\!)$).

Important for proofs is also the symbol $\vdash$. This sign is used to show that the logical statement has been proven. Our proofs are usually implications. Hence, our assumptions are not in front of the symbol but are the antecedent of the theorem or lemma. The goal is the consequent.

2.5 Separation Logic

For this project, an understanding of the theory of Separation Logic is essential for verifying the system-level design of an FH. Separation Logic is an extension of Hoare Logic that enables reasoning about heap cells [9, 10]. Notably, it incorporates heap pointers ($\mapsto$) and the separation conjunction ($*$) into the standard Hoare logic rules. Several implementations of Separation Logic have been integrated into HOL4. For example, Tuerk has replicated and ported the SmallFoot framework to HOL [11].

Another implementation of separation logic uses sets to represent the separation conjunction [12]. This separation logic theory will be used in this project. Specifically, this logic framework identifies heap cells as a function called one that takes an address and outputs the value at that address (one (*addr*, *value*)). This will be the representation of our heap cells.

3

Fibonacci Heap Design

The FH is a rather complex data structure. Its difficulty comes from the rose tree shape and the complex design of its operations.

Therefore, it is important to choose an intuitive design. This design will be based on monolithic nodes. Their concept will be described in the next sections.

Moreover, we choose a structure-based approach for verifying the FH operations. This allows us to separate the verification into several levels (see Figure 3.1). The highest level will be the most abstract view and the final level is a concrete implementation.

Logical Level. This level reasons about the FH on the highest abstraction level. This abstraction is done with finite maps and set relation. Furthermore, lemmas and theorems refine the finite maps into forests of trees.

Algorithm Level. This next level introduces an high-level algorithm that describes a transformation on the FH. The algorithm itself uses an immutable data structures. In our case, the data structure is a list of trees.

Memory Level. The memory level introduces a refined implementation of the algorithm in form of system-level code. The code uses addresses and pointer and mutates memory. Actually, the memory level code is already similar to a Pancake implementation.

Concrete Implementation Level. The final level reasons about the similarities of the memory level code and a concrete implementation in Pancake. Proofs on this level consists of the semantic representation of the Pancake code which is also known as Abstract Syntax Tree (AST).

While all four levels are structurally separated, the verification works with interleaved definitions, lemmas and theorems. This is required to connect the levels with each other. In particular, each level has its own definitions and logical reasoning. Then the definitions are used in logical statements and applied to the respective code.

In each level the code is based on a different data structures. Thus, the heap representation needs to be refined between each level as well. Moreover, the code between each level needs to be semantically identical such that the code from the previous implementation level implies the correctness of the next level.

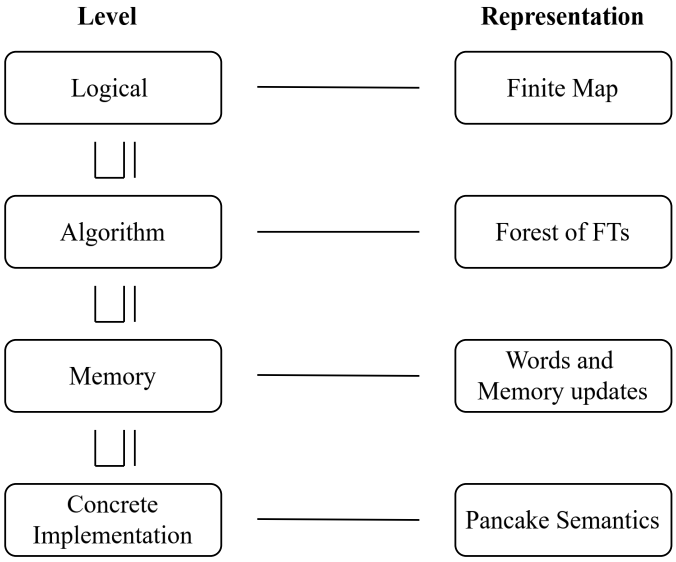


Figure 3.1: The structural setup of the project. Each level is identified by a different structure. Our goal is the correctness proof over all levels. Such a proof over so many level requires multiple refinements ($\sqsubseteq$).

Therefore, our approach focuses on the construction of each level, their verification and finding suitable refinements between them. This starts with design of the FH on most abstract level. Here it is important to have the use case of the FH in mind.

3.1 Monolithic Nodes

In our FH design, we have chosen to use monolithic nodes. A monolithic node is the unification of all direct data associated with a node. So, instead of splitting data of an element into multiple data structures, the monolithic node is the only component that holds its data. If another data structure wants to have access to the nodes data, it needs to subscribe to the node by holding a reference.

Therefore, on one hand the monolithic node centralizes data, but on the other hand, it complicates access and storage of the data from an outside structures. However, in our context the drawback is acceptable. In a system-level implementation one should avoid copying and storing duplicate data to ensure memory consistency and minimal fragmentation.

During the design of our monolithic node, we separate logical and memory values. The logical values are necessary for algorithm-level verification and are important for operations on the FH or the Dijkstra algorithm. In contrast, memory values hold addresses that indicate the node's position in the FH and are only important during memory verification.

In particular, logical values correspond to:

v = the value of the current shortest edge

e = the edges of the node (a pointer to an array)

f = a flag that indicates whether this nodes is part of the FH or has already been discovered

m = a mark used in the update operation

Note that these values do not refer to the position of the elements in the heap, since higher levels of abstraction do not require knowledge of the arrangement of elements in the heap. Therefore, these values are enough to verify the FH on the algorithm level.

At system level, however, additional information about the position is needed. The cycling double-linked list and tree-like structure must be represented using pointers and memory addresses.

Hence, we refine the logical node to a memory node. This node extends the previous node with the following values:

b = the previous element in the double linked list of the Fibonacci heap

n = the next element in the double linked list of the Fibonacci heap

p = the parent of this node in the Fibonacci heap

c = a pointer to the children of this node

r = the rank of this tree (the number of direct children)

With the exception of **r** all of these values correspond to other nodes in the heap. The value **r** is a special refinement for the memory level since the operations of the FH require knowledge about the rank of a tree. On algorithm level it is easy to know that rank, on the memory level one would need to calculated it by an iteration over the list of children.

Another refinement between the algorithm and memory level is the child pointer. The algorithm level can use a list data structure to represent all direct children. However, the memory level just has addresses and pointers. Hence, the parent points at the first child of its children.

This intrusive design is visually represented in Figure 3.2.

3.2 Datatypes

In order to implement and verify the FH with a monolithic node, we first implement it in HOL4. The HOL4 definitions can be considered as functions of a programming language, enabling abstract reasoning. This is crucial for verifying our FH, since it enables the implementation of auxiliary functions. In particular, the memory level relies on auxiliary functions for verification.

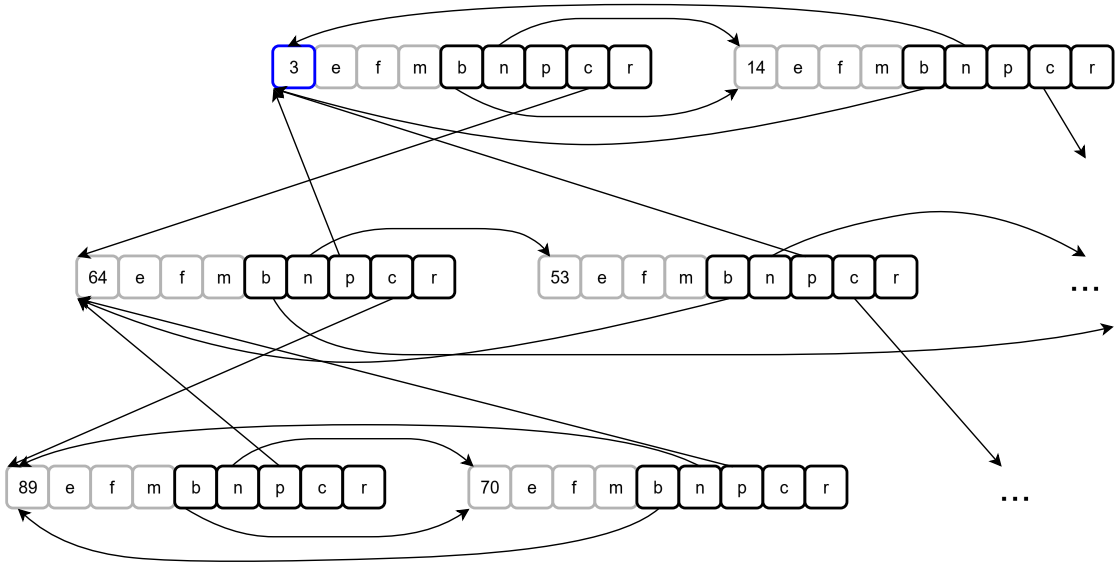


Figure 3.2: Illustrates parts of Figure 2.2 with our monolithic design. Logical values are in grey cells and memory values are separated in black cells.

In total, we implement one data type and two records. The data type corresponds to an FT and holds three different values: (α, β) ft = FibTree α β $((\alpha, \beta)$ ft list).

1. The first value α is referred to as the 'key'. This is the identifer for the element and is semantically analogous to a heap address for the memory level.
2. The second value β represents the content of the node. This can take the form of two different record types, which are described in the next paragraph.
3. The third value in an FT is a list of FTs that yield its children.

Since the monolithic node distinguishes between logical and memory data, we reflect this in the implemented record types. In particular, the implementation considers a data_node to be the logical node, while an annotated_node corresponds to the memory node. Notice that the data_node misses the flag of being in the heap as all of our operations on the FH assume this flag to be set to true.

```
α data_node = <|
  value : α word;
  edges : α word × (α word × num) list;
  mark : bool
|>
```

```
α annotated_node = <|
  data : α data_node;
  before_ptr : α word;
  next_ptr : α word;
  parent_ptr : α word;
  child_ptr : α word;
  rank : num
|>
```

Finally, we construct a auxiliary type called fts that acts as a forest of FTs: fts = (α, β) ft list. It is the primary datatype of the implementation and verification since it connects the logical level and memory level together.

4

Implementation

The implementation of operations on the FH is done in two parts. First, the high-level implementation is implemented and verified. This implementation is carried out HOL4's higher-order logic. Afterwards, we redo the implementation and verification at system level, which includes mutable memory. This implementation should be almost identical to the high-level implementation, meaning that we can reuse the logical reasoning about the FH state and only require additional reasoning about the memory.

The implementations are carried out as follows. First, we implement the insertion and melding operations on the FH. The meld operation is then reused in the other operations. Next, we implement the extract minimum operation.

Unfortunately, due to time constraints, we could not implement and verify the update operation.

4.1 Insert and Meld

The implementation of insert and meld are identical. Note that the meld operation combines two distinct FHs into one. Clearly, a single FH with no children should be an FH in its own right. So, one needs to show that a single empty node without any children corresponds to a FH. We will show this in Chapter 5

Moreover, the meld operation is the most fundamental part of our FH design since it is used later on. This operation takes two disjoint FHs, fts_1 and fts_2. Prior to the meld operation, it is known that the first element in both heaps is the smallest and that both heaps contain distinct elements. The meld operation proceeds as follows. First, the first element of fts_2 is taken and appended to the last element of fts_1. Then, it takes the last element of fts_2 and concatenates it with the first element of fts_1. The root lists of both FHs are now concatenated.

To complete the melding process, a new minimum must be selected. This is determined by comparing the first elements of fts_1 and fts_2. If the value of the first element in fts_1 is smaller, its pointer is returned, and vice versa.

The following code snippet represents the HOL4 code that is verified on the algorithm level.

Definition 1 *Appends two root lists depending on the value of the first tree in the root list.*

```
fts_meld [] fts2 =def fts2

fts_meld (ft1 :: fts1) [] =def ft1 :: fts1

fts_meld (FibTree k1 v1 l1 :: fts1) (FibTree k2 v2 l2 :: fts2) =def
  if v1.value ≤+ v2.value then
    FibTree k1 v1 l1 :: (fts1 ++ FibTree k2 v2 l2 :: fts2)
  else FibTree k2 v2 l2 :: (fts2 ++ FibTree k1 v1 l1 :: fts1)
```

This definition is defined with pattern matching. In the case that one of the FHs is empty, the result is simply the other FH.

In comparision, the system-level code needs to append both root lists with pointers. Hence, the code needs to write to the specific memory address.

This change in the code comes with an additional variable c. The variable c is a boolean that keeps track of the functional correctness. Hence, if there is an error with the memory operation c will be set to F.

In the case of the meld operation, each time a value is read or updated, it must be ensured that the value is present in the heap. This is important to ensure memory correctness. In the code, we keep track of these assertions using the variable c.

If one memory check fails, the function returns with F. At the end of the function, such an error is propagated to the code layer above.

Definition 2 *Melding two heaps on the memory level requires to adjust the pointers of the first and last element in the root lists.*

```
fib_heap_meld (a1, a2, m, dm) =def
  if a2 = 0w then (a1, m, T)
  else
    if a1 = 0w then (a2, m, T)
    else
      let (l_a1, c) = read_mem (a1 + before_off) m dm T;
      (l_a2, c) = read_mem (a2 + before_off) m dm c;
      (m, c) = write_mem (l_a1 + next_off) a2 m dm c;
      (m, c) = write_mem (a2 + before_off) l_a1 m dm c;
      (m, c) = write_mem (l_a2 + next_off) a1 m dm c;
      (m, c) = write_mem (a1 + before_off) l_a2 m dm c;
      (v_a2, c) = read_mem a2 m dm c;
      (v_a1, c) = read_mem a1 m dm c
      in
        if v_a1 ≤+ v_a2 then (a1, m, c) else (a2, m, c)
```

Notice that appending the two root lists a_1 and a_2 does not differentiate between appending a_1 to a_2 or appending a_2 to a_1. The memory result will always be the same. This indicates the main difference between the algorithm and memory level and will be shown as a lemma in Chapter 5.

The corresponding pancake code looks similar:

```
fun meld (a1,a2) {
  if a2 == 0 {
    return a1;
  }
  if a1 == 0 {
    return a2;
  }

  var l_a1 = lds 1 (a1 + 4);
  var l_a2 = lds 1 (a2 + 4);

  st l_a1 + 5, a2;
  st a2   + 4, l_a1;
  st l_a2 + 5, a1;
  st a1   + 4, l_a2;

  var v1 = lds 1 a1;
  var v2 = lds 1 a2;

  if v1 <=+ v2 {
      return a1;
  } else {
      return a2;
  }
}
```

The memory reads and writes are replaced by load (`lds`) and store (`st`) operations respectively. Furthermore, the loads have an additional number before the address field that indicates how many words one wants to load. In our case, we always load one word.

4.2 Extract Minimum

The extract minimum operation performs multiple structural changes to the FH. First, it removes the smallest element from the tree and adds its children to the root list. Secondly, it rebalances the FH. Thirdly, it identifies and sets the new minimum value for the FH, setting its dedicated pointer accordingly.

Removing the smallest element is simple in the design. One removes the element from the heap by extracting it from the root list and appends its children with the root list using the meld operation.

Definition 3 *Removes the top element and merges its children with the remaining heap.*

$$\mathsf{fts\_rm\_min}\ [] \stackrel{\text{def}}{=} (0\mathsf{w}, [])$$

$$\mathsf{fts\_rm\_min}\ (\mathsf{FibTree}\ k\ v\ l :: ts) \stackrel{\text{def}}{=} (k, \mathsf{fts\_meld}\ l\ ts)$$

The memory level code has a pointer to their parent. Hence, the parent pointers of the minimum's children need to be set to `0w`.

The complex part of extract minimum is the rebalancing of the remaining FH. In the original paper, the authors performed this by linking FTs in the root list with the same rank together and then performing a merge for those trees.

A heap design with monolithic nodes makes this linking process very complex. Linking two trees in our low-level design requires extensive reasoning of memory addresses and pointers. The original paper describes an algorithm to link the trees of the root list consecutively with the help of an array.

Since the verification of the rebalancing operation is rather complex, we simplify the task slightly. Originally, the algorithm uses an array of size n, where n is the maximum possible rank of a tree. As explained in Chapter 2, the trees rank is the number of direct children. One can show that the rank is at maximum $\log x$ where x are the number of nodes in the heap [4].

However, establishing the fact that $\log x$ is the maximum rank for a FT is difficult. Since we focus on end-to-end verification and not on time or space complexity, we simplify the maximum rank to be a constant. Hence, the array is bound to this maximum rank as well.

We choose the constant max_rank to be 185. This number is special for FTs as it indicates the breakpoint between 128-bit addresses and the minimal size of a tree. In particular, as mentioned in chapter 2, a FT has at least the Fibonacci number $rank + 2$ as children. Hence, the minimal number of children of a tree with rank 185 is fib_num 187. Now, the Fibonacci number of 187 is very large and also the first Fibonacci number that is larger than 2^{128}. Hence, a FT with rank 187 has not enough addresses to index all its children on a 128-bit word machine.

Binding the maximum rank to a constant has the valuable side effect that the rank of FTs is limited in memory as well. During our verification we will induct on the rank which is stored as a memory word. This memory word is read at the start of a function call, and transformed into a number. Such a transformation calculates the value of the rank modulo the size of the word. If the word size is smaller than the rank, the modulo operation will return a malformed number. Therefore, we need to ensure that the word size is at least the size of the maximum rank.

Returning to the array implementation, we also require that the array shall be empty before and after the rebalancing operation. This allows the reuse of the array such that it is not allocated in each extract minimum operation. Reallocation of the array would be challenging in Pancake since Pancake programs require memory to be statically allocated before execution.

The array needs to represent missing values. Therefore, we choose the Option value None for the algorithm level and the special value `0w` for the memory level.

The rebalancing algorithm uses the array rl in a nested recursions. It begins by removing an FT from the root list and attempting to insert it into the array at its current rank.

If an element is already stored at that location, the two FTs are merged. The new FT then performs the linking step, since the number of children has increased. Otherwise, the array value corresponds to None, indicating the insertion of the current tree.

Finally, the procedure moves on to the next element in the root list and continues recursively until the root list is empty.

Definition 4 *Melds the tree with the higher value with the children of the other tree.*

```
fts_merge_trees (FibTree k1 v1 l1) (FibTree k2 v2 l2) ≝
  if v1.value ≤+ v2.value then
    FibTree k1 v1 (fts_meld l1 [FibTree k2 v2 l2])
  else FibTree k2 v2 (fts_meld l2 [FibTree k1 v1 l1])
```

Definition 5 fts_link_trees *links two trees recursively and terminates when the tree can be added into array gracefully.*

```
fts_link_trees n rl (FibTree k v l) ≝
  if n = 0 then (rl, F)
  else if max_rank ≤ length l then (rl, F)
  else
    case rl(length l) of
      None ⇒ (rl(length l ↦ Some (FibTree k v l)), T)
    | Some (FibTree k' v' l') ⇒
      if max_rank − 1 ≤ length l then (rl, F)
      else
        fts_link_trees (n − 1) rl(length l ↦ None)
          (fts_merge_trees (FibTree k v l)
            (FibTree k' v' l'))
```

Definition 6 fts_link_root_list *links all elements in the root list consecutively.*

```
fts_link_root_list n rl [] ≝ (rl, T)

fts_link_root_list n rl (FibTree k v l::fts) ≝
  if n = 0 then (rl, F)
  else
    let (n_rl, flag) =
      fts_link_trees max_rank rl (FibTree k v l)
    in
      if flag ⟺ F then (n_rl, F)
      else fts_link_root_list (n − 1) n_rl fts
```

The functions fts_link_trees and fts_link_root_list use the natural number n as a clock in its function definition. These clocks are not important for the functionality of the operations but ensure the termination of the algorithm. Furthermore, the clocks are need for the verification between the algorithm and memory level.

Actually, it is possible to design the algorithms without clocks. In the case of fts_link_trees this requires a termination proof. However, during verification one would need to do an additional refinement step, because the verification of the corresponding system-level implementation relies on induction on n. Hence, we omit our initial algorithm implementation and only show the implementation that uses a clock.

After linking all FT in the root list, all FTs are in the array. Now, one reconstructs the FH by recursively melding all single FTs in the array to an accumulator. This reconstruction reuses the meld operation of the FHs.

Definition 7 *Melds all trees to the accumulator until the array is empty.*

```
fts_collect_array r rl acc ≝
 if r = 0 then
  case rl(r) of
   None ⇒ (acc, rl)
  | Some (FibTree k v l) ⇒
   (fts_meld [FibTree k v l] acc, rl(r ↦ None))
 else
  case rl(r) of
   None ⇒ fts_collect_array (r − 1) rl acc
  | Some (FibTree k v l) ⇒
   fts_collect_array (r − 1) rl(r ↦ None)
    (fts_meld [FibTree k v l] acc)
```

Finally, a function called fts_reb encapsulates the whole rebalancing part of the extract minimum operation. Therefore, the final extract minimum operations in the algorithm level only exists of the fts_rm_min and fts_reb functions.

Definition 8 *Composes the linking and collection of FTs.*

```
fts_reb rl fts ≝
 let (l_rl, flag) = fts_link_root_list (length fts) rl fts;
 (fts′, e_rl) = fts_collect_array (length l_rl − 1) l_rl []
 in
  (fts′, e_rl, flag)
```

Definition 9 *Removes the top element of the heap and rebalances the root list.*

```
fts_extract_min fts ≝
  let (min, fts) = fts_rm_min fts;
  (fts', e_rl, flag) = fts_reb emp_rl fts
  in
    (min, fts', e_rl, flag)
```

In Chapter 5, we only show the verification of the algorithm level and argue about necessary refinements for the memory level. Hence, the implementations for the memory level are not shown or discussed in detail.

4.3 Auxiliary Functions

Verifying separation logic can be very laborious. Since manually writing the heap is time-consuming and error-prone, we use auxiliary functions to construct the heap cells instead. These functions take a list of FTs with only data_nodes as their value and produce a list of FTs with annotated_nodes. These functions therefore annotate the FTs with pointers by reading the keys of the FTs. Note that the keys contain the identifier and address of an FT. These auxiliary functions, along with additional theorems and lemmas, help to reduce the effort required for verifying mutable memory.

Definition 10 *Annotation of a segment in a forest of trees.*

```
ann_fts_seg p s b n [] ≝ []

ann_fts_seg p s b n (FibTree k v ys::xs) ≝
  FibTree k (new_anode v b n p (head_key ys) (length ys))
    (ann_fts_seg k (head_key ys) (last_key ys)
      (head_key_t (head_key ys) (TL ys)) ys)::
    ann_fts_seg p s k (head_key_t s (TL xs)) xs
```

Definition 11 *Annotation of the whole forest is propagated to the segment definition.*

```
ann_fts p [] ≝ []

ann_fts p (x::xs) ≝
  ann_fts_seg p (head_key [x]) (last_key (x::xs))
    (head_key_t (head_key [x]) xs) (x::xs)
```

As the elements reason about a cycling double-linked lists and their own children in the form of pointers, the auxiliary functions require their own auxiliary functions to query the designated key from a list of FTs. These auxiliary functions query the head_key and last_key of lists and help to annotate the pointers.

Query function differ between total and partial functions. Total function allow the set of a default value if no key can be queried. On the opposite partial function only query keys that are in the list. If the list is empty, the partial function should return the `0w`. Hence, a partial function is implemented using their corresponding total function. These partial functions improve the readability of logical statements and can be used to weaken statements about the heap cell.

For instance, consider the statement using the total function `head_key_t`:

`head_key_t 0w` l = `head_key_t` x l

Assume x is an arbitrary long term and l is not empty. Then both sides of the equality can be weakened to `head_key` l.

This verification part will not display expanded heap cells.

5

Verification

This chapter focuses on the verification of the operations on the FH. It starts with a small introduction of the verification setup. This introduction describes the connection between the different verification levels. Afterward, a section discusses the design of our FH invariant and its properties. Those properties are the fundamental part of the verification and must be satisfied after each operation on the FH. This procedure is the main focus of this chapter and is spread out over multiple sections after the describing the properties of the FH invariant.

Finally, our verification uses a significant number of theorems and lemmas across all verification levels. The majority of these components are not shown. Instead, this chapter focuses only on the most important lemmas that conclude different verification layers.

5.1 Setup

The verification of the FH is done in three steps, which are visualized in Figure 5.1. The steps are enumerate in descending order.

1. The first step is defining the FH as a composition of logical statements. These logical statements consists of logical level data types and algorithm definitions. The major composition is a FH invariant that must be satisfied after code execution. However, some code has additional invariants which come with further logical and algorithm level definitions.

2. The second step connects the code from the algorithm level with the memory level implementation. In particular, one wants to show that the memory level code implements the algorithm code. Since his logical statement is described as an implication from the algorithm code to the memory level code. The implication allows the reasoning that if the FH invariant holds for the algorithm code, it also holds for the memory level code.

3. Finally, in the last step one needs to show that the memory level code has a valid Pancake representation. This final verification reuse the memory from the memory level implementation. Then, one must show that the reading and writing operations in the memory level are equivalent to loading and storing on the concrete implementation level.

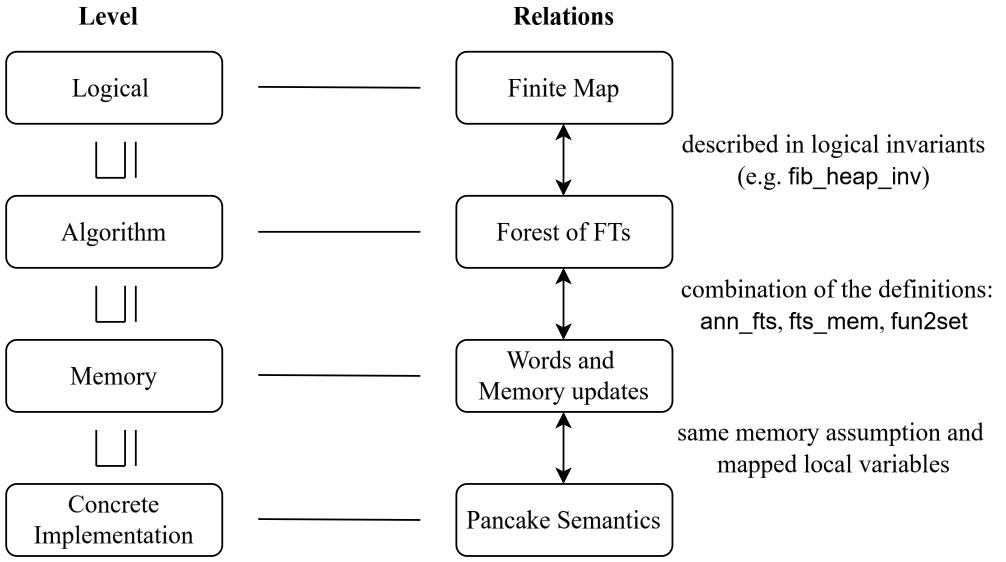


Figure 5.1: The verification setup of the project. Since each level needs some refinement to reason about the logical statements, it is difficult to map our proofs to classify a proof to a particular level. Hence, the verification uses parts of the previous and next level to show logical transformations.

The following sections will focus on the first step and less on the second and third steps. To be precises, all steps will be shown for the Meld operation. However, the Extract Minimum operation is a composition of many operations. Hence, the focus will be on important refinements from the algorithm to the memory level and less on a complete verification. Moreover, the third step is omitted for the Extract Minimum operation since we do not expect it to show any important information. Actually, the verification step from the memory to the concrete implementation level should be the very similar across all implementations. This is further discussed during the verification of the Meld operation.

5.2 Fibonacci Heap Invariant

The first verification step uses FH invariants, that is satisfied in the pre- and post-condition. If the invariant is satisfied, the heap is well-formed. The FH invariant lives between the logical and algorithm level and uses finite maps and forests of FTs. This makes the definition of the logical statements easier and binds the invariant to the algorithm representation However, it requires some connection between both levels.

The following parts displays all directly associated definitions of the invariant and finally the FH invariant itself. Besides the FH invariant, these definitions are not discussed in detail.

Definition 12 *Inductively defined relation for finding a FT in a forest of FTs.*

$$\frac{}{\mathsf{fts\_has}\ k\ v\ (\mathsf{FibTree}\ k\ v\ ts :: rest)}\ \text{head}$$

$$\frac{\mathsf{fts\_has}\ k\ v\ rest}{\mathsf{fts\_has}\ k\ v\ (\mathsf{FibTree}\ k_1\ v_1\ ts :: rest)}\ \text{tail}$$

$$\frac{\mathsf{fts\_has}\ k\ v\ ts}{\mathsf{fts\_has}\ k\ v\ (\mathsf{FibTree}\ k_1\ v_1\ ts :: rest)}\ \text{child}$$

Definition 13 *All elements in the forest are distinct from each other.*

$$\mathsf{fts\_all\_dist}\ [] \stackrel{\text{def}}{=} \mathsf{T}$$

$$\mathsf{fts\_all\_dist}\ (\mathsf{FibTree}\ k\ v\ ts :: fts) \stackrel{\text{def}}{=}$$
$$\mathsf{fts\_has\_inj}\ (\mathsf{FibTree}\ k\ v\ ts :: fts) \wedge$$
$$\forall\ v.\ \neg\mathsf{fts\_has}\ k\ v\ ts \wedge \neg\mathsf{fts\_has}\ k\ v\ fts \wedge$$
$$(\forall\ k\ v.\ \mathsf{fts\_has}\ k\ v\ ts \Rightarrow \neg\mathsf{fts\_has}\ k\ v\ fts) \wedge$$
$$\mathsf{fts\_all\_dist}\ ts \wedge \mathsf{fts\_all\_dist}\ fts$$

Definition 14 *The value of the FTs in the whole forest are smaller or equal to the supplied value.*

$$\mathsf{fts\_is\_min}\ v_0\ [] \stackrel{\text{def}}{=} \mathsf{T}$$

$$\mathsf{fts\_is\_min}\ v\ (\mathsf{FibTree}\ v_1\ n\ ts :: rest) \stackrel{\text{def}}{=}$$
$$v \leq_+ n.\mathsf{value} \wedge \mathsf{fts\_is\_min}\ v\ ts \wedge \mathsf{fts\_is\_min}\ v\ rest$$

Definition 15 *Propagation of a logical predicate inside a forest for all sub-forests.*

$$\mathsf{every\_fts}\ P\ xs \stackrel{\text{def}}{=}$$
$$P\ xs \wedge \forall\ k\ v\ l.\ \mathsf{mem}\ (\mathsf{FibTree}\ k\ v\ l)\ xs \Rightarrow \mathsf{every\_fts}\ P\ l$$

Definition 16 *Affirms the parent value to be smaller than the values of any children. Reuses Definition 14.*

$$\mathsf{fts\_parent\_lower\_eq}\ [] \stackrel{\text{def}}{=} \mathsf{T}$$

$$\mathsf{fts\_parent\_lower\_eq}\ (\mathsf{FibTree}\ k\ v\ l :: ts) \stackrel{\text{def}}{=}$$
$$\mathsf{fts\_is\_min}\ v.\mathsf{value}\ l \wedge \mathsf{fts\_parent\_lower\_eq}\ ts$$

Definition 17 *Based on the Corollary 1 of the original FH paper. A FT must have at least the size of the Fibonacci number (rank + 2).*

```
fib_heap_shape_ok [] ≝ T

fib_heap_shape_ok (FibTree k v ys :: ts) ≝
  fib_num (length ys + 2) ≤ 1 + fts_size ys ∧
  fib_heap_shape_ok ys ∧ fib_heap_shape_ok ts
```

Definition 18 *The FH invariant. Properties are separated by the ∧ operator.*

```
fib_heap_inv fh fts ≝
  (∀ k v. lookup fh k = Some v ⇒ k ≠ 0w) ∧                    P1
  (∀ k v e.
    lookup fh k = Some (v, e) ⟺
      ∃ m. fts_has k (new_dnode v e m) fts) ∧                 P2
  fts_all_dist fts ∧                                          P3
  fts_is_min (fts_hd_value fts) fts ∧                         P4
  every_fts fts_parent_lower_eq fts ∧                         P5
  fib_heap_shape_ok fts                                       P6
```

Our FH invariant (fib_heap_inv) has six properties (**Pn**) that are separated with the conjunction operator.

P1: The first property uses the finite map and ensures that none of the keys in the map are `0w`. The `0w` corresponds to *null* and indicates an empty list or a missing value.

P2: The next property ensures that all elements in the finite map are also in forest of FTs. Notice that the finite map does not hold the mark since the mark is unique to FHs and not causal related to a graph node.

P3: The third property guarantees that all elements in the forest are distinct. Such an assumption is strictly necessary since our implementation splits the root list and combines it after some operations on the elements. The combination of FTs requires the elements to be distinct from each other. It is important to realize that it consists of multiple distinct parts itself (see Definition 13).

P4: The fourth property focuses on the partial order relation of the FH. This partial order relation requires the head value of the root list to be the smallest value in the whole FH. This partial order relation makes use of Definition 14.

P5: In addition, our heap invariant needs a property to reason about the relation between parents and successors. In particular, parents need to be smaller or equal to its successors. This must be the case for all list of FTs in the whole heap. Hence, the predicate of Definition 15 populates the Definition 16.

P6: Finally, the invariant has a property about its shape. This property enforces the forest of trees to follow the shape of a FH (see Definition 17). The shape of a FH is described in Corollary 1 of the original FH paper [4].

5.3 Insert and Meld

The previous chapter claims that inserting and melding two FHs are similar. The following theorem shows this connection by constructing a FH with a valid FH invariant from a single element.

Consider an element k that shall be inserted into a heap fh. Then k is not an element of fh. Moreover, if the DLL pointers of the k node point to itself, the k is a heap by itself.

Theorem 1 *A single node in the heap is identical to a FH with only one element.*

$$
\begin{aligned}
&\vdash k \neq \texttt{0w} \Rightarrow \\
&\quad (\texttt{empty\_node}\ k\ (v, e) * \mathit{frame})\ (\texttt{fun2set}\ (m, dm)) \Rightarrow \\
&\quad\ (\texttt{fts\_mem} \\
&\quad\quad (\texttt{ann\_fts}\ \texttt{0w}\ [\texttt{FibTree}\ k\ (\texttt{new\_dnode}\ v\ e\ \texttt{F})\ []]) * \\
&\quad\ \mathit{frame})\ (\texttt{fun2set}\ (m, dm)) \wedge \\
&\quad\ \texttt{fib\_heap\_inv}\ \texttt{empty}\langle k \mapsto (v, e)\rangle \\
&\quad\quad [\texttt{FibTree}\ k\ (\texttt{new\_dnode}\ v\ e\ \texttt{F})\ []]
\end{aligned}
$$

In this theorem logical parts are separate into propositional statements and separation logic statements. The memory is applied as an argument to $\texttt{fun2set}\ (m, dm)$. Be aware the definition of empty_node links the element to itself in its monolithic node. So the pointers of the cycling double-linked list for the previous and next element point to k respectively. The variable *frame* reoccurs in all memory proofs and indicates the reset of the memory. Essentially, the variable states that our implementation is in the memory and that there could be other artifacts in memory as well.

Therefore, the melding algorithm can be used to insert a single element in to the heap.

Essentially, the insert functionality needs to prove that a single element in the FH satisfies the FH invariant. This is true by the design of our invariant and its propositions. Hence, the melding algorithm needs to be verified.

5.3.1 Melding Heaps

As described in the previous chapter, the meld operation appends two FHs and returns the head of the smaller heap. The verification of this operation has multiple assumptions. First, we know that both heaps satisfy the FH invariant. Second, both heaps must have disjoint elements. Otherwise, the heap could not be represented on the memory level. Finally, after the meld process, the new heap must satisfy the FH invariant as well.

With those assumptions one can construct the following lemma:

Lemma 1 *Appending the FH with the larger head value will result in a valid FH.*

$$\begin{aligned}&\vdash \mathsf{fib\_heap\_inv}\ fh_1\ fts_1 \wedge \mathsf{fib\_heap\_inv}\ fh_2\ fts_2 \wedge\\ &\quad \mathsf{disjoint}\ (\mathsf{dom}\ fh_1)\ (\mathsf{dom}\ fh_2) \wedge\\ &\quad \mathsf{fts\_hd\_value}\ fts_1 \leq_+ \mathsf{fts\_hd\_value}\ fts_2 \Rightarrow\\ &\qquad \mathsf{fib\_heap\_inv}\ (fh_1 \uplus fh_2)\ (fts_1 +\!\!+ fts_2)\end{aligned}$$

The verification of Lemma 1 follows by separating each invariant property. This results in six distinct lemmas. In general, all lemmas use the properties of the invariant in their original heap. The following part will talk about these proofs on a very abstract level and leaves out necessary steps to form matching assumption and goal terms.

1. The first property ensures that all keys must not be `0w`:

   $$\forall\, x.\ \mathsf{lookup}\ (fh_1 \uplus fh_2)\ \texttt{0w} \neq \mathsf{Some}\ x$$

   This follows directly from the definition of lookup and **P1** of the fib_heap_inv that is satisfied for both finite maps. The union of finite maps is left associative so lookup queries the left finite map first. This finite map does not include a key with value `0w` since it satisfies a fib_heap_inv and its property **P1**. Then, lookup queries the right finite map, which does not include the `0w` as well due to the same reasoning.

2. The second property reasons that all elements in the finite map are also in the list of FTs and vice versa.

   $$\begin{aligned}\forall\, k\ v\ e.\ \mathsf{lookup}\ (fh_1 \uplus fh_2)\ k = \mathsf{Some}\ (v, e)\ \Leftrightarrow\\ \exists\, m.\ \mathsf{fts\_has}\ k\ (\mathsf{new\_dnode}\ v\ e\ m)\ (fts_1 +\!\!+ fts_2)\end{aligned}$$

   This proof uses the **P2** of the previous heaps and the disjointness of both finite maps. As the statement is a logical equivalence, one needs to prove the statement in both directions.

   Both directions follow a similar pattern. Lets first discuss the left to right direction. Then the statement about lookup is an assumption and with the disjointness assumption there are two cases. Either, k is in fh_1 or in fh_2. Using **P2** of the previous FH invariant and the its finite map respectfully, one knows that either: $\mathsf{fts\_has}\ k\ (\mathsf{new\_dnode}\ v\ e\ m)\ fts_1 \vee \mathsf{fts\_has}\ k\ (\mathsf{new\_dnode}\ v\ e\ m)\ fts_2$. This is an append theorem of the definition fts_has, that is applied to our goal and concludes both cases.

   The right to left proof is using the same steps as above just the other way around. First, one splits the root list of FTs with the append theorem and gains a case distinction. Then one needs to use the disjointness assumption of the finite maps and the lookup definition. Finally, **P2** of the previous FH invariants will resolve that the element associated with the key k is in the union of finite maps.

3. The third property is the most complex to verify.

   $$\mathsf{fts\_all\_dist}\ (fts_1 +\!\!+ fts_2)$$

First of all, one needs to prove three different parts of the fts_all_dist definition. The proof for the first part, fts_has as injective function, is simple an follows immediately from **P3** of the previous FH invariants. However, the second and third part of the definition requires that all members of the forest are unique. Here the disjointness of fh_1 and fh_2 comes into play. However, to use the disjointness assumption **P2** of the previous FH invariants is strictly necessary. We leave out this rather complex and verbose proof but just show its lemma. It should be intuitive that when both finite maps are disjoint, and each forest fts_1 and fts_2 has unique keys by **P3** of the previous FH invariants, then their concatenation should also have unique keys.

Lemma 2 *Melding two distinct forests of FTs with distinct trees.*

$$
\begin{aligned}
\vdash\ & (\forall\, k\ v\ e.\\
& \quad \mathsf{lookup}\ fh_1\ k = \mathsf{Some}\ (v, e) \iff\\
& \quad\ \exists\, m.\ \mathsf{fts\_has}\ k\ (\mathsf{new\_dnode}\ v\ e\ m)\ fts_1) \wedge\\
& (\forall\, k\ v\ e.\\
& \quad \mathsf{lookup}\ fh_2\ k = \mathsf{Some}\ (v, e) \iff\\
& \quad\ \exists\, m.\ \mathsf{fts\_has}\ k\ (\mathsf{new\_dnode}\ v\ e\ m)\ fts_2) \wedge\\
& \mathsf{fts\_all\_dist}\ fts_1 \wedge \mathsf{fts\_all\_dist}\ fts_2 \wedge\\
& \mathsf{disjoint}\ (\mathsf{dom}\ fh_1)\ (\mathsf{dom}\ fh_2) \Rightarrow\\
& \quad \mathsf{fts\_all\_dist}\ (fts_1 +\!\!+ fts_2)
\end{aligned}
$$

4. The fourth property focuses on the head element of the root list. The statement requires that the value of the head is the smallest in the whole forest.

   $$\mathsf{fts\_is\_min}\ (\mathsf{fts\_hd\_value}\ fts_1)\ (fts_1 +\!\!+ fts_2)$$

   The proof of this lemma is straight forward. First, we use an append theorem to show that the above statement is equivalent to:

   $$\mathsf{fts\_is\_min}\ (\mathsf{fts\_hd\_value}\ fts_1)\ fts_1 \wedge \mathsf{fts\_is\_min}\ (\mathsf{fts\_hd\_value}\ fts_2)\ fts_2$$

   The left side of the conjunction holds by **P4**. Further the right side of the conjunction holds because $\mathsf{fts\_hd\_value}\ fts_1 \leq_+ \mathsf{fts\_hd\_value}\ fts_2$.

5. The fifth property is about the parent and successors relation.

   $$\mathsf{every\_fts}\ \mathsf{fts\_parent\_lower\_eq}\ (fts_1 +\!\!+ fts_2)$$

   Melding two FHs together does not touch this relation. Hence, the first step is simplifying every_fts which resolves to

   $$
   \begin{aligned}
   & \mathsf{fts\_parent\_lower\_eq}\ (fts_1 +\!\!+ fts_2)\ \wedge\\
   & \forall\, k\ v\ l.\ \ \mathsf{mem}\ (k, v, l)\ fts_1 \vee \mathsf{mem}\ (k, v, l)\ fts_2 \Rightarrow \mathsf{fts\_parent\_lower\_eq}\ l
   \end{aligned}
   $$

   Then an append theorem for fts_parent_lower_eq splits the concatenation of the lists and the proof concludes by using **P5** of the previous FH invariants.

6. The last property ensures the correct shape of a FH.

   $$\mathsf{fib\_heap\_shape\_ok}\ (fts_1 +\!\!+ fts_2)$$

   This statement focuses on the trees instead of the FH and thus one can separate the statement into:

fib_heap_shape_ok fts_1 ∧ fib_heap_shape_ok fts_2

Those are the **P6** of the previous FH invariants respectfully.

After the verification of logical lemma, the verification of fts_meld is a direct conclusion.

Theorem 2 *The result of melding two disjoint FHs on the algorithm level satisfies the FH invariant.*

⊢ fib_heap_inv fh_1 fts_1 ∧ fib_heap_inv fh_2 fts_2 ∧
disjoint (dom fh_1) (dom fh_2) ∧
fts_meld fts_1 fts_2 = fts' ⇒
fib_heap_inv (fh_1 ⊎ fh_2) fts'

The main focus of the verification is the if-clause case-distinction between $v_1 \leq_+ v_2$ and $\neg(v_1 <_+ v_2)$ where v_1 and v_2 are the smallest values of the FHs respectfully (see the if-clause in Definition 1).

The first case immediately concludes with our logical lemma for melding two FHs. The second case needs some reasoning on words. In particular, one can show that $\neg(v_1 \leq_+ v_2) \Rightarrow v_2 <_+ v_1$. Moreover, this implies that $v_2 <_+ v_1 \Rightarrow v_2 \leq_+ v_1$. Now, one can reuse the logical lemma by changing the finite maps and list of trees during the quantifier instantiations.

5.3.2 System-Level Verification

Moving on to the verification of the memory level, the main focus is the verification of the separation logic. The proof goes by case-distinction on the forest of FHs. However, now the proof has many possible case.

1. First, there are two cases that one of the heaps is empty. Those cases are rather trivial, since the implementation returns the one full heap or – if both heaps are empty – just an empty heap.

2. Second, there are three cases that one or both heaps only yield one FT in the root list. Each of those cases need to be verified separately, since the heap cells differ in each scenario. In particular, one must reason about the head and the last element of the root list. If there is only one element in the root list, this becomes the same. So the three cases are, two heaps with only one element or either heap with one element and the other heap with at least two elements.

3. Finally, there is the case that both heaps have at least two elements. Here it is important to identify that no matter how the root lists are concatenated, the memory cells stay at the same location (see Lemma 3). This realization shows the main difference between the algorithm and memory level. No matter which element of both heaps is smaller, on memory level the representation of the heap will be identical after the operation. Only the pointer into the heap varies.

Lemma 3 *Appending two forests in either way is identical on the memory level.*

```
⊢ fts_mem (ann_fts p (xs ++ ys)) =
    fts_mem (ann_fts p (ys ++ xs))
```

Theorem 3 *The algorithm implementation with corresponding memory implies the correctness of the memory level implementation.*

```
⊢ fts_meld fts1 fts2 = fts' ∧
  (fts_mem (ann_fts p fts1) * fts_mem (ann_fts p fts2) *
   frame) (fun2set (m, dm)) ⇒
   ∃ m' .
    fib_heap_meld (head_key fts1 ,head_key fts2 , m, dm) =
     (head_key fts' , m' , T) ∧
    (fts_mem (ann_fts p fts') * frame)
     (fun2set (m' , dm))
```

The proof is only possible because the memory implementation follows the exact design of the algorithm level. In particular, the if-clause is semantically identical to the algorithm level. Otherwise, the proof would not be possible. The same applies for the Pancake code. Since the code is semantically identical to the memory level, the following theorem is provable.

Theorem 4 *The correct execution of the memory level implies the correctness of Pancake implementation.*

```
⊢ fib_heap_meld (a1 , a2 , m, dm) = (a' , m' , T) ∧
  lookup s.locals «a1» = Some (ValWord a1) ∧
  lookup s.locals «a2» = Some (ValWord a2) ∧ dimindex (:α) = 8 ∧
  m = s.memory ∧ dm = s.memaddrs ⇒
   ∃ l. evaluate (meld_body, s) =
        (Some (Return (ValWord a')),
         s with <|memory := m'; locals := l|>)
```

The theorems logical statements look more complicated then they are. For instance the first assumption is just the correct execution of the memory level. Important is the T in its result which implies the correct execution of all statements inside the function body. Hence, all read and writes of the function succeed.

Next assumptions are important for the pancake code evaluation. The evaluation uses a state s which holds the local variables and the memory. The second and third assumption supply this state with the input variables for the Pancake implementation. The fourth assumption sets the byte width to 8 (the byte width for words in our verification).

The goal statements combines all of the assumptions. In particular, the goal statements says that if one evaluates the Abstract Syntax Tree (AST) of the Pancake implementation, one will reach the same return value and memory as the memory implementation. Also there are some additional local variables.

The proof of the lemma is straight forward due to the similarity between the memory and concrete implementation. Each statement of the AST is evaluate step by step. For loads and stores to memory the evaluation function requires the address to be in memory. This is implied by the correct execution of the memory implementation.

A subtle in this theorem is the absence of any verification of time complexity. Actually, the Pancake compiler does not guarantee the same time complexity after compilation. Hence, our missing focus on time complexity in this thesis.

After the verification of the Pancake code, the end-to-end verification is possible. The end-to-end proof does not have any implementation as an assumption. Instead, the assumptions are purely based on the Pancake code.

Theorem 5 *The FH invariant with a correct memory imply the correctness the Pancake code.*

```
⊢ fib_heap_inv fh1 fts1 ∧ fib_heap_inv fh2 fts2 ∧ disjoint (dom fh1) (dom fh2) ∧
  (fts_mem (ann_fts 0w fts1) * fts_mem (ann_fts 0w fts2) * frame)
   (fun2set (m, dm)) ∧
  lookup s.locals «a1» = Some (ValWord (head_key fts1)) ∧
  lookup s.locals «a2» = Some (ValWord (head_key fts2)) ∧
  dimindex (:α) = 8 ∧ m = s.memory ∧ dm = s.memaddrs ⇒
   ∃fts' l m'.
    fib_heap_inv (fh1 ⊎ fh2) fts' ∧
    (fts_mem (ann_fts 0w fts') * frame) (fun2set (m', dm)) ∧
    evaluate (meld_body, s) =
      (Some (Return (ValWord (head_key fts'))),
       s with <|memory := m'; locals := l|>)
```

All the assumptions and goals are known from the previous theorems. Be aware that the locals of the pancake code need to be set to the corresponding heads of the forests. The proof is very simple due to the previous theorems. First, one instantiates the existence quantifier in the goal with fts_meld fts_1 fts_2 and abbreviates it with fts'. Then the goal will look identical to the final theorem above without the existence quantifier and one gains the additional assumption fts_meld fts_1 fts_2 = fts'. Now one can apply the Theorems 2, 3, and 4 in this order to resolve the three goal statements in the same order respectively.

This proofs the correctness of the evaluation of the Pancake generated AST. The end-to-end correctness is established with the correctness proof of the Pancake compiler [6].

5.4 Extract Minimum

The verification of the extract minimum operation is split into the verification of multiple functions that encapsulate parts of the algorithm. On the algorithm level, the encapsulated functions are the removal of the minimal element and the rebalancing

of the root list. After the removal of the minimal element of the FH, the heap loses its partial order relation that ensures the smallest element to be at the head of the root list (**P4**). Hence, we construct another FH invariant called fib_heap_inv_weak that corresponds to the original invariant without **P4**. This weakened invariant is a precondition of the rebalancing function. The result of the rebalancing should be a well-formed heap that satisfies the original FH invariant fib_heap_inv.

This separation into multiple parts is shown in Figure 5.2. The figure gives a very general overview of the verification of the different functions on the logical level. The spiked box indicate pre- and postconditions of functions. Square boxes elaborate make those conditions more precise since they are followed by a recursion in the majority of the time. These recursions are indicated with pointed cycling arrows. Between the pointed lines is also always some update, usually to proceed the recursion. The flow of the program and verification is indicated with normal arrows. Notice that the function fts_link_trees has two outgoing arrows depending on the situation.

Moreover, Figure 5.2 only uses finite maps and the name of the associated functions. Hence, this tree only represents the highest possible view on this verification. The image already foreshadows two additional invariants. Both invariants will be discussed in the next section.

Further, the following section shows only select lemmas and theorems of this verification. Showing everything would be too comprehensive. In particular, the algorithm level verification focuses on the merging of two FTs.

Especially, **P6** is interesting since this property ensures the correct shape of a FT and gives the heap its name. Also the collection of the rebalancing array is shown in more detail. This function performs operations on the array and will showcase the rebalancing invariant. The same rebalancing invariant is also used during the verification of fts_link_trees and fts_merge_trees. However, the proofs for link trees is polluted by the additional invariants of the nested recursion. Thus, we decided to omit a detailed analyzes of the link of FTs and just show the resulting lemmas.

Finally, the memory level verification looks at refinements between the algorithm and memory level. Here the focus will be the removal of the minimum element. The verification of the rebalancing function on memory level is omitted completely since it does not need any additional refinements. Actually, we crafted our algorithm level implementation such that the memory level verification is directly implied. However, we show the memory representation of the rebalancing array in detail since this definition implies the re-usability of the array.

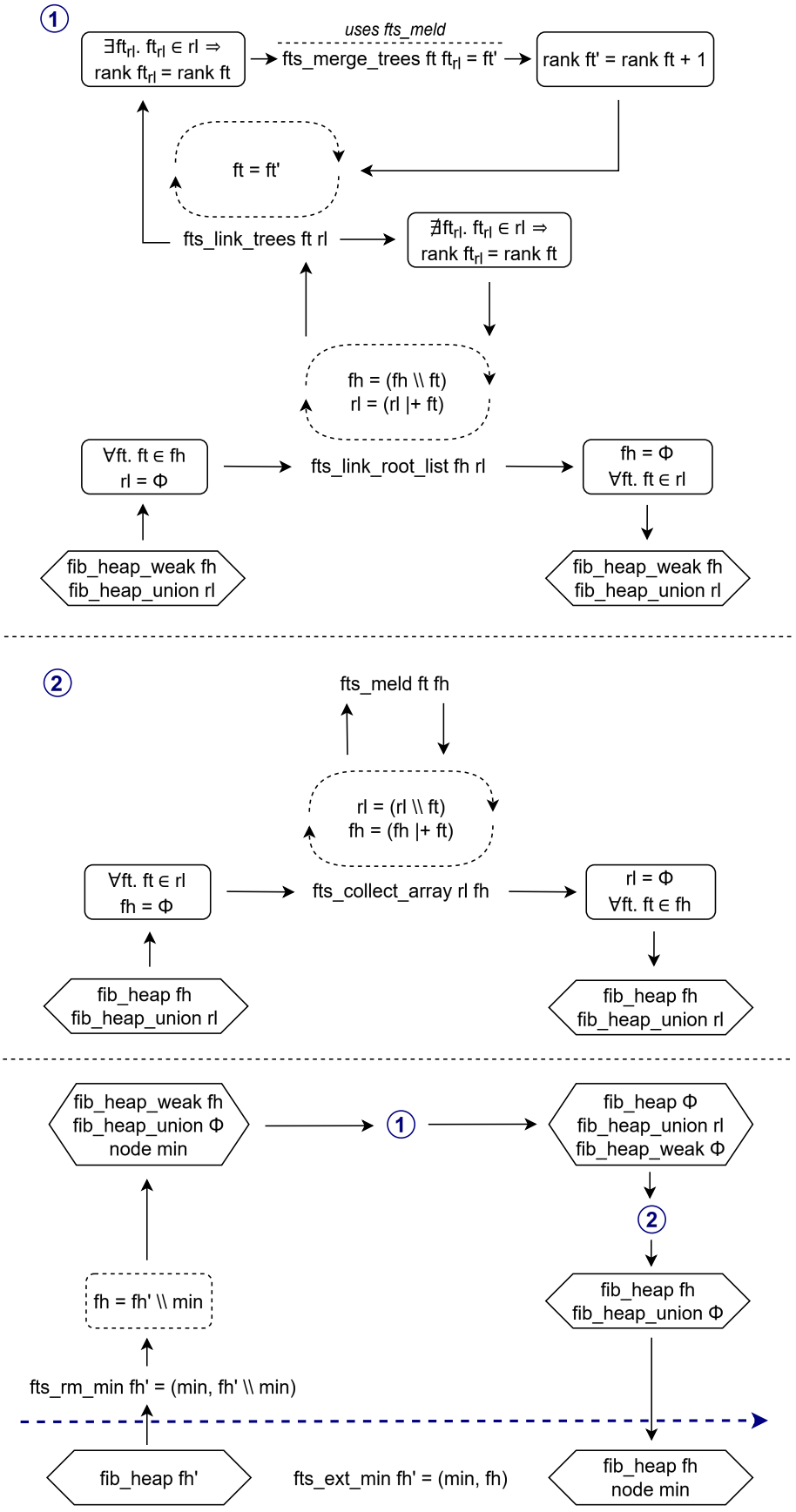


Figure 5.2: A very general overview of the extract minimum operation verification. The verification flow is at the bottom of the figure. The parts ① and ② show the link of the root list and collection of the array respectively.

5.4.1 The Rebalancing Invariant

The rebalancing invariant is called fib_heap_inv_union and is associated with the rebalancing array. This invariant takes a finite map *fh* and a list of pairs *fh_ts*. The pairs inside *fh_ts* consist of another finite map and an optional FT.

Definition 19 *The invariant of the rebalancing array.*

```
fib_heap_inv_union fh fh_ft ≝
  every
    (λ (fh, O_ft) .
       case O_ft of
         None ⇒ fib_heap_inv fh []
       | Some ft ⇒ fib_heap_inv fh [ft]) fh_ft ∧                    UP1
  all_disjoint fh_ft ∧                                              UP2
  fh = fh_union fh_ft ∧                                             UP3
  ∀ n map k v l.                                                    UP4
    n < length fh_ft ∧ fh_ft(n) = (map,Some (FibTree k v l)) ⇒
      length l = n
```

Overall, the invariant union consists of four properties.

UP1: The first property ensures that all pairs in the list satisfy the original fib_heap_inv where the optional FT is the only element in the root list. This is is done with the every relation that ensures the given predicate for all elements in the list. The definition of every is similar to Definition 15.

UP2: The second property states that all finite maps that are part of the pairs are disjoint from each other. Hence, all FTs have nodes with different identifiers.

UP3: The third property constructs a finite map union of all the paired finite maps. This union is equal to the finite map given as additional argument to the list of pairs.

UP4: The fourth property is about the rebalancing array. As described in the previous chapter, the rebalancing operation tries to insert elements into the array at the position of the elements rank. Hence, this statement says that if there is a tree at a certain position, then this tree has the rank of the index position.

The invariant union is a concatenation of all important aspects during the verification. During the merging and collection of the arrays the properties **UP1** and **UP2** are important. Obviously, the proof wants to keep the invariant alive during this procedure and the merging of FTs but also the collection of the array needs the disjointness of the corresponding finite maps to perform the melding operation.

In contrast, property **UP3** is important at the end of the verification. In particular, the final proof wants to reason that all elements before the rebalancing are also there after the rebalancing. Since the code moves all elements from the root list into the

array, this union collects all elements in all FTs. Then, after the rebalancing this union will be the finite map for the final fib_heap_inv.

Finally, **UP4** is the main invariant of the array. It says that a tree at the index n must have the rank n. This property is the most important part for linking trees and strictly necessary to ensure the correct shape of the FTs (**P6**).

5.4.2 Nested Recursion

We implemented the linking of the root list via nested recursion. The first recursion focuses on the root list of the FH and its function is called fts_link_root_list. This function *pops* the current top of the FH and inserts this FT into the array. At the start of the verification the finite map of the array is empty. After every element in the heap has been *popped* and inserted into the array this finite map should be identical to the finite map of assumed fib_heap_inv_weak.

This logical reasoning is summarized in the following theorem.

Theorem 6 *Linking the root list with a disjoint array implies that all elements are inserted into the array.*

$$
\begin{aligned}
&\vdash \mathsf{length}\ rl = \mathsf{max\_rank} \wedge \\
&\quad \mathsf{fib\_heap\_inv\_weak}\ fh_1\ fts \wedge \\
&\quad \mathsf{fib\_heap\_inv\_union}\ fh_2\ (\mathsf{ts\_to\_fhts}\ rl) \wedge \\
&\quad \mathsf{disjoint}\ (\mathsf{dom}\ fh_1)\ (\mathsf{dom}\ fh_2) \wedge \\
&\quad \mathsf{fts\_link\_root\_list}\ n\ rl\ fts = (rl', \mathsf{T}) \Rightarrow \\
&\qquad \mathsf{fib\_heap\_inv\_union}\ (fh_1 \uplus fh_2)\ (\mathsf{ts\_to\_fhts}\ rl') \wedge \\
&\qquad \mathsf{length}\ rl = \mathsf{length}\ rl'
\end{aligned}
$$

Theorem uses the definition of ts_to_fhts. This definition reconstructs a finite map from a tree representation. While not further explained it is also part of the fts_collect_array verification and thus defined in Definition 21.

The function fts_link_root_list uses the function fts_link_trees (see Definition 6). The function itself focuses on the insertion of a tree into the rebalancing array.

There are two cases possible. Either, the array is free or there is already an FT with the same rank as the tree to be inserted. Then the function fts_link_trees organizes the merge of both trees to one tree. This increments the children of one of the trees. Finally, fts_link_trees tries to insert the FT with the increment rank into the array. Here appears the second recursion.

This procedure is described in the following algorithm level theorem. Notice that the assumptions yield a modified **UP4** which is shown as an implication. This implication is the unique invariant of this function verification.

Theorem 7 *Linking a tree into the array will add these trees elements to the union invariant.*

```
⊢ fib_heap_inv fh₁ [FibTree k v l] ∧
  fib_heap_inv_union fh₂ (ts_to_fhts rl) ∧
  disjoint (dom fh₁) (dom fh₂) ∧
  length rl = max_rank ∧
  fts_link_trees n rl (FibTree k v l) = (rl', T) ⇒
    fib_heap_inv_union (fh₁ ⊎ fh₂) (ts_to_fhts rl') ∧
    length rl = length rl'
```

Finally, the nested recursion uses the fts_merge_trees function, which, in its part, uses the fts_meld function. The verification of fts_merge_trees is very similar to the verification of the fts_meld.

Only two properties change slightly in their verification: **P5** and **P6**. The property **P5** must be reassured for the newly added child. This proof relies on the if-clause of the function definition (see Definition 4). The reuse of Definition 14 in fts_parent_lower_eq makes this proof almost identical to the verification of **P4** during the fts_meld verification.

However, the lemma of **P6** is more interesting since the merging procedure increases the number of children. In particular, **P6** states that the size of a FT must be at least of the $rank + 2$ Fibonacci number (see Definition 17). This property gives the FH its name.

Therefore, the following proof shows the verification of this property in detail. Let the FTs be ft_1 and ft_2 and for the sake of consistency, we always add ft_2 to children ($succ$) of ft_1.

The proof starts with a case distinction on the list of direct children.

1. First, let us assume that $succ$ was empty. Then both FTs have no children. So, ft_2 is added to the empty list $succ$ which results in fib_num (length [ft_2] + 2) ≤ 1 + fts_size [ft_2]. After resolving the length and fts_size this is fib_num 3 ≤ 2 which is true since fib_num 3 = 2.

2. Second, let us assume that the $succ$ is not empty. Then, the children of ft_2 will be $h :: t$ where h is the head of the list and t the tail. Furthermore, let l be the children of h and ys the children of ft_1. Then, length t + 1 = length ys because both FTs have the same rank. In addition, both FTs satisfy the **P6**. Hence, we need to prove the following lemma.

   Lemma 4 *Adding a FT to the the children of another FT with the same rank satisfies **P6**.*

   ```
   ⊢ length t + 1 = length ys ∧
     fib_num (length t + 3) ≤ fts_size l + (fts_size t + 2) ∧
     fib_num (length ys + 2) ≤ fts_size ys + 1 ⇒
       fib_num (length ys + 3) ≤ fts_size l + (fts_size t + (fts_size ys + 3))
   ```

To prove this lemma, one resolves length ys to be length $t + 1$ in the assumptions and in the goal. Hence the goal will be:

fib_num (length $t + 4$) $\leq$ fts_size t + fts_size ys + fts_size $l + 3$

Then, we can unwind the definition of fib_num once to change the goal to:

fib_num (length $t + 3$) + fib_num (length $t + 2$) $\leq$
fts_size t + fts_size ys + fts_size $l + 3$

Looking at our assumptions, this is true, since:

fib_num (length $t + 3$) $\leq$ fts_size t + fts_size $l + 2$ $\wedge$
fib_num (length $t + 3$) $\leq$ fts_size $ys + 1$

and

fib_num (length $t + 2$) $\leq$ fib_num (length $t + 3$)

This concludes the verification of fts_merge_trees. The following paragraphs will discuss the collection of the rebalancing array.

5.4.3 Heap Reconstruction

After all elements in the root list are linked together or are of different rank, one needs to collect the array to form the new FH. Since the nested recursion ensured to union the finite maps of merged trees and the fib_heap_inv_union includes a union of all FTs in the array, we know that no elements have been lost but are somewhere in the array of FTs. Furthermore, the union invariant affirms that the array is a collection of FHs with a single FT in their root list. All of these separate FHs satisfy the fib_heap_inv. Hence, the algorithm can reuse the fts_meld operation.

The following theorem needs to be verified:

Theorem 8 *Having a disjoint accumulator and the array, the* fts_collect_array *will union all of these elements such that all elements are in the accumulator. Furthermore, the array is empty and still of the same size as before the execution.*

$\vdash$ fib_heap_inv fh_1 acc $\wedge$
fib_heap_inv_union fh_2 (ts_to_fhts rl) $\wedge$
disjoint (dom fh_1) (dom fh_2) $\wedge$
length rl = max_rank $\wedge$ $r <$ length rl $\wedge$
($\forall\, x.\ x <$ length $rl \wedge r < x \Rightarrow rl\,(x)$ = None) $\wedge$
fts_collect_array r rl acc = (fts, rl') $\Rightarrow$
fib_heap_inv ($fh_1 \uplus fh_2$) fts $\wedge$
($\forall\, x.\ x <$ length $rl' \Rightarrow rl'\,(x)$ = None) $\wedge$
length rl = length rl'

At the first expression, this theorem can be very difficult and complex to read. However, it becomes more clear when one abstracts from the visual representation and focuses on the conjunction parts of the assumptions and the goal.

The first two assumptions talk about a valid fib_heap_inv and the rebalancing invariant fib_heap_inv_union. Those correspond to an accumulator and the rebalancing array respectfully. The idea is to accumulate all elements in the array in a particular way that will satisfy the fib_heap_inv at the end. Hence, the arguments of both definition are used in the function call of fts_collect_array. Jumping ahead to the first statement in the goal, one can see that the union of their finite maps shall be part of the resulting invariant. Hence, when the accumulator is empty during the first call of the fts_collect_array function, the resulting FH will contain only and all values from the rebalancing array.

The next assumption talks about the disjointness of the accumulator and the rebalancing array which is important during the reconstruction of the heap. The reconstruction of the heap reuses the fts_meld function, and thus needs disjointness of these finite maps.

Finally, all other assumptions talk about the processing of the rebalancing array. The array is collected from the largest index to the smallest index. Hence, the theorem has a statement that all elements above the currently collect index are None. This must be true, because all elements above the current index have been collect already. Using this function in another way would not result in its desired functionality.

From those assumption about the rebalancing array, it becomes imminent that the returned rebalancing array should be empty and of the same length as the original array. This is summarizes in the two last statements of the goal. Notice that this goal allows the reuse of the rebalancing array on the memory level.

The following paragraphs perform a detail analysis of the lemmas and logical transformation of the assumptions and the goal. This shall give a deep inside on the verification effort of one function in HOL4. The paragraphs will not talk about the base case of $r = 0$, since the base case and the inductive step are very similar in their verification.

5.4.4 Inductive Step of the Heap Reconstruction

The inductive step instantiates the theorem of the previous section where r is replaced by Suc r. Further, the theorem will appear as an implication for an arbitrary r as an assumption (see Figure 5.3).

Since our assumptions are phrased with Suc r, the first change towards this proof setup is the unwinding of the fts_collect_array function once. This will lead two possible cases. Either the array has at index of Suc r the value None or Some (FibTree k v l).

The following section includes definitions and lemmas that are used in this case distinction.

```
0.  ∀rl'' fh2' fh1' acc'.
      fib_heap_inv fh1' acc' ∧
      fib_heap_inv_union fh2' (ts_to_fhts rl'') ∧
      DISJOINT (FDOM fh1') (FDOM fh2') ∧ LENGTH rl'' = 185 ∧
      r < LENGTH rl'' ∧ (∀x. x < LENGTH rl'' ∧ r < x ⇒ rl''(x) = NONE) ∧
      fts_collect_array r rl'' acc' = (fts,rl') ⇒
      fib_heap_inv (fh1' ⊎ fh2') fts ∧
      (∀x. x < LENGTH rl' ⇒ rl'(x) = NONE) ∧ LENGTH rl' = 185
1.  fib_heap_inv fh1 acc
2.  fib_heap_inv_union fh2 (ts_to_fhts rl)
3.  DISJOINT (FDOM fh1) (FDOM fh2)
4.  LENGTH rl = 185
5.  SUC r < LENGTH rl
6.  ∀x. x < LENGTH rl ∧ SUC r < x ⇒ rl(x) = NONE
7.  fts_collect_array (SUC r) rl acc = (fts,rl')
------------------------------------
    fib_heap_inv (fh1 ⊎ fh2) fts ∧ (∀x. x < LENGTH rl' ⇒ rl'(x) = NONE) ∧
    LENGTH rl' = 185
```

Figure 5.3: The starting state of the induction step verification. Assumption 0 is the Induction Hypothesis.

Lemma 5 *If no element is present at the current index of the heap, one can reduce the index of the invariant by one.*

$$\vdash (\forall\, x.\ x < \text{max\_rank} \wedge \text{Suc}\ r < x \Rightarrow rl(x) = \text{None}) \wedge rl(\text{Suc}\ r) = \text{None} \Rightarrow \forall\, x.\ x < \text{max\_rank} \wedge r < x \Rightarrow rl(x) = \text{None}$$

Lemma 6 *Removing an element from index r yields the corresponding union invariant properties with the update array.*

$$\begin{aligned}\vdash\ & \text{length}\ rl = \text{max\_rank} \wedge (\forall\, x.\ x < \text{length}\ rl \wedge r < x \Rightarrow rl(x) = \text{None}) \wedge \\ & r < \text{length}\ rl \wedge rl(r) = \text{Some}\ (\text{FibTree}\ k\ v\ l) \wedge \\ & \text{fib\_heap\_inv\_union}\ fh\ (\text{ts\_to\_fhts}\ rl) \Rightarrow \\ & \exists\, fh_1\ fh_2\,. \\ & \quad \text{fib\_heap\_inv}\ fh_1\ [\text{FibTree}\ k\ v\ l] \wedge \\ & \quad \text{fib\_heap\_inv\_union}\ fh_2\ (\text{ts\_to\_fhts}\ rl(r \mapsto \text{None})) \wedge \\ & \quad \text{disjoint}\ (\text{dom}\ fh_1)\ (\text{dom}\ fh_2) \wedge fh = fh_1 \uplus fh_2\end{aligned}$$

Lemma 7 *A* fib_heap_inv *with an empty list implies an empty finite map.*

$$\vdash \text{fib\_heap\_inv}\ fh\ [] \Rightarrow fh = \text{empty}$$

Definition 20 *Flattening a forest to a list takes the node and its map related values and proceeds recursively with the children and rest of the list.*

```
flat_fts [] ≝ []

flat_fts (FibTree k v ts::rest) ≝
  [(k, v.value, v.edges)] ++ flat_fts ts ++ flat_fts rest
```

Definition 21 *Reconstructs a finite map from an optional tree and pairs them. This is done for the whole list. The definition uses* alist_to_fmap *from the* alist *Theory.*

```
ts_to_fhts ts ≝
  map
    (λ n.
       case n of
         None ⇒ (empty, None)
       | Some t ⇒ (alist_to_fmap (flat_fts [t]), Some t))
    ts
```

Definition 22 *The finite maps in the list are part of a recursive union on the list*

```
fh_union [] ≝ empty

fh_union ((fh, fts) :: rest) ≝ fh ⊎ fh_union rest
```

Theorem 9 *Appending two list of unioned heaps is the union of both list unions.*

```
⊢ fh_union (xs ++ ys) = fh_union xs ⊎ fh_union ys
```

Theorem 10 *If the whole list is empty, then the union of finite maps is empty as well.*

```
⊢ (∀ x. x < length list ⇒ list(x) = (empty, None)) ⇒ fh_union list = empty
```

Lemma 8 *If the whole list is empty, the reconstructed list is also empty.*

```
⊢ (∀ x. x < length rl ∧ r < x ⇒ rl(x) = None) ⇒
    ∀ x. x < length rl ∧ r < x ⇒ (ts_to_fhts rl) (x) = (empty, None)
```

Lemma 9 *If all elements in a reconstructed list above an index are empty, that part of the list can be cut of and said to be empty.*

```
⊢ (∀ x. x < length (xs ++ [t] ++ ys) ∧ length xs < x ⇒
        (ts_to_fhts (xs ++ [t] ++ ys)) (x) = (empty, None)) ⇒
    ∀ y. y < length (ts_to_fhts ys) ⇒ (ts_to_fhts ys) (y) = (empty, None)
```

Lemma 10 *If all elements of the array above an index are empty, the finite map of the array only holds elements from below the index.*

$$\begin{aligned}&\vdash (\forall\, x.\ x < \mathsf{length}\ (xs +\!\!+ [y] +\!\!+ ys) \wedge \mathsf{length}\ xs < x \Rightarrow\\ &\qquad (xs +\!\!+ [y] +\!\!+ ys)\,(x) = \mathsf{None}) \wedge\\ &\quad \mathsf{fib\_heap\_inv\_union}\ fh\\ &\quad (\mathsf{ts\_to\_fhts}\ xs +\!\!+ (\mathsf{empty}, \mathsf{None}) :: \mathsf{ts\_to\_fhts}\ ys) \Rightarrow\\ &\quad fh = \mathsf{fh\_union}\ (\mathsf{ts\_to\_fhts}\ xs)\end{aligned}$$

Case: None. This case is the easier part, since it is not necessary to logical transform the accumulator and the rebalancing array. Actually, we will reuse both invariants as they stay unmodified. Now, the focus goes to use the IH. Obviously, it is true that:

$\mathsf{Suc}\ r < \mathsf{length}\ rl \Rightarrow r < \mathsf{length}\ rl$

This allows one to instantiate the IH with all parts of the antecedent except of

$\forall\, x.\ x < \mathsf{length}\ rl \wedge r < x \Rightarrow rl\,(x) = \mathsf{None}$

since our assumptions state the following:

$\forall\, x.\ x < \mathsf{length}\ rl \wedge \mathsf{Suc}\ r < x \Rightarrow rl\,(x) = \mathsf{None} \wedge rl\,(\mathsf{Suc}\ r) = \mathsf{None}$

Therefore, one needs to construct a small lemma that shows the final part of the antecedent. This is done in Lemma 5 which corresponds ot the logical level proof of the None case.

The proof of the lemma is simple. After moving all of the antecedents of the goal into the assumption, a case distinction on $\mathsf{Suc}\ r < x$ solves this lemma. When $\mathsf{Suc}\ r < x$ then one can use the first assumption of the lemma. However, if it is the case that $\neg(\mathsf{Suc}\ r < x)$ but $r < x$ then $\mathsf{Suc}\ r = x$. This case is the second assumption of the lemma. Hence, this resolves the goal.

Now, one can instantiate the IH completely and the None case is solved.

Case: Some (FibTree $k\ v\ l$). This case is rather complex. Since there is an element in the array, the element must be removed and merged into the accumulator. The result will be the new accumulator and is used with the IH. Moreover, our invariant of the rebalancing array needs to change accordingly such that the disjointness assumption is not violated. Those changes towards the rebalancing array must also reflect the following unrolled function expression of the function fts_collect_array:

$\mathsf{fts\_collect\_array}\ r\ rl\,(\mathsf{Suc}\ r \mapsto \mathsf{None})\ \ (\mathsf{fts\_meld}\ [\mathsf{FibTree}\ k\ v\ l]\ \ acc) = (fts, rl')$

We prove the transformation of the first three assumption on the logical level with the Lemma 6.

The lemma is somewhat intuitive to read. It reuses the assumptions of the original fts_collect_array case. Using these assumptions the first three goal statements become an imminent conclusion. Obviously, when one removes the

element at index r, this index should satisfy **UP1**. Furthermore, removing the element should not invalidate the union invariant and by **UP2** the finite maps of the separated FH and the new union of FHs should be disjoint.

It remains the last statement in the goal. It says that the original finite map of the whole rebalancing array is a union of the separated finite map with the new finite map of the rebalancing array. This statement is strictly necessary to make the IH match the goal of this induction step case in the original proof. Otherwise, the finite maps of the invariants would not match.

The proof idea for this final statement is based on **UP3** and the possibility of splitting the rebalancing array at the position r. Then it is possible to use theorems about the append case of fh_union.

By splitting on the index r, property **UP3** of the fib_heap_inv_union and Theorem 9, we gain the following statement:

$$\mathsf{fh\_union}\ xs \uplus fh_1 \uplus \mathsf{fh\_union}\ ys = fh_1 \uplus \mathsf{fh\_union}\ xs$$

Since fh_union xs and fh_1 are disjoint from each other, they are commutative. Hence, fh_union ys must correspond to an empty finite map that acts as neutral element on the union of finite maps.

That fh_union ys is an empty finite map is also hidden in the assumptions of our original logical Lemma 6:

$$\forall\, x.\ x < \mathsf{length}\ rl \wedge r < x \Rightarrow rl(x) = \mathsf{None}$$

Though this assumption only reasons about all entries above the index r of the rebalancing array being None, one needs to remember that the reconstruction of a not existing FT also needs to satisfy the fib_heap_inv by **UP1** of the fib_heap_inv_union. Semantically, a FH with only a none existing FT in its root list is equivalent to an empty FH. Obviously, empty FHs do not have elements, and thus have empty finite maps in their fib_heap_inv statements. In particular, one can show Lemma 7 by a case distinction on finite maps.

Overall, to finish the proof of Lemma 6 it is necessary to reason about finite map reconstruction, construct a fh_union_empty theorem proof, and then the union of finite maps such that it is possible to apply the fh_union_empty theorem in this context. However, this would distract from the original goal which is the verification of the inductive step of the function fts_collect_array. Therefore, we only give the necessary definitions and lemmas above this case distinction such that an interested reader can look at the correctness of this proof.

After proving that one can extract a FT with its small FH from the rebalancing array, one can focus on the IH (see Figure 5.3). With this logical lemma the fib_heap_inv_union is in the correct shape for the IH. Moreover, now it is possible to meld the extract FT from the array with the accumulator. Here, the implementation reuses the fts_meld operation since a single FT satisfies the fib_heap_inv.

Since Theorem 8 ensures that the array and the accumulator are disjoint from

each other, the extract tree must also be disjoint from the accumulator. This is enough to reuse the verification proof of fts_meld.

To reason about the finite maps before and after fts_meld, we assign them clear identifiers. Let the finite map of the accumulator be fh_1, of the separated tree with its invariant fh_2, and of the updated rebalancing array fh_3. Then the result of fts_meld will state fib_heap_inv ($fh_2 \uplus fh_1$) (fts_meld [FibTree k v l] acc). This new assumption is used in the IH with the updated fib_heap_inv_union to resolve the majority of its antecedents.

The only remaining antecedent is

$\forall\, x.\ x < \mathsf{length}\ rl\,(\mathsf{Suc}\ r \mapsto \mathsf{None}) \wedge r < x \Rightarrow rl\,(\mathsf{Suc}\ r \mapsto \mathsf{None})\,(x) = \mathsf{None}$

The proof of this assumptions is very similar to a lemma in the None case of the induction step (see lemma 5), and thus is not further mentioned.

With all antecedents of the IH resolved, its consequent is added to the assumptions: fib_heap_inv ($fh_2 \uplus fh_1 \uplus fh_3$) fts' where fts' is the result of the function fts_collect_array in the IH.

Again, all finite maps are disjoint from each other. Therefore, by applying the law of commutativity on finite map union the statement fib_heap_inv ($fh_1 \uplus (fh_2 \uplus fh_3)$) fts' is true as well. This is what we wanted to show.

After the rebalancing the array is empty and the FH invariant is satisfied with the same elements as before (except of the removed minimum). Hence, this marks the end of the extract minimum operation.

Theorem 11 *Extract minimum removes the top element from the forest and returns a valid FH.*

$\vdash$ fib_heap_inv_weak fh_1 fts $\wedge$
fts_reb emp_rl fts = (fts', e_rl, T) $\Rightarrow$
fib_heap_inv fh_1 fts' $\wedge$ e_rl = emp_rl

5.4.5 Memory Level Refinements

The previous section skipped the verification of the function fts_rm_min since its definition is rather simple (see Definition 3). However, on the memory level removing an element from the heap requires modification of associated pointers. Rather obvious are the pointers of the previous and next element. So when popping off the first tree of the FH, the last nodes needs to point to the second node of the heap and vice versa. This functionality is described in a fib_heap_pop function. The proof is omitted since it is essentially the verification of fib_heap_meld reversed with a tree and a forest instead of two forests.

However, Definition 3 does not only remove the first FT of the heap, but also adds all its children to the root list of the FH. While adding the children to the FH uses the meld operation, all children have still a pointer to its parent – the extracted minimum. Hence, the parent pointer of the children must be set to the `0w`.

This setting of the parent pointer is done with recursion over all annotated children. The proof uses two annotated lists of FTs xs and ys that are concatenated. The verification uses induction on the length of the children, and thus the induction invariant needs to reason about the segmented annotation (ann_fts_seg). In particular, one needs to reason about the annotation of an arbitrary child in the list of children. First this arbitrary child gets a new parent, then it is added to the list of the children with new parents. This procedure is described in the following theorem. The variable p indicates the old parent and np the new parent. Hence, at the start of the verification xs is empty and ys is full. At the end xs is full and ys is empty.

Theorem 12 *The function* fib_heap_set_parent *overwrites the pointer p to np for all elements inside ys.*

```
⊢ n = length ys ∧
   (fts_mem
     (ann_fts_seg np (head_key_t (head_key xs) ys)
       (last_key_t (last_key xs) ys)
       (head_key_t (head_key xs) (TL xs ++ ys)) xs) *
    fts_mem
     (ann_fts_seg p (head_key_t (head_key ys) xs)
       (last_key_t (last_key ys) xs)
       (head_key_t (head_key_t (head_key ys) xs) (TL ys))
       ys) * frame) (fun2set (m, dm)) ∧
  fib_heap_set_parent n (head_key ys, np, m, dm) = (a, m', c) ⇒
   (fts_mem (ann_fts np (xs ++ ys)) * frame)
    (fun2set (m', dm)) ∧ head_key ys = a ∧ c
```

After the minimal element is removed, the memory level proceeds with the rebalancing of the FH. Since this memory operation uses only addresses the elements of the rebalancing array must be annotated as well. Hence, we construct a function called reb_array_mem which annotates a FT if present. Further, the array holds the trees identify add the offset of its rank (see Definition 23).

Definition 23 *Constructs the memory of the rebalancing array recursively. Either there is only one heap cell if no element is present. Otherwise there is the whole associated tree in addition to the array heap cell.*

```
reb_array_mem a off [] ≝ emp

reb_array_mem a off (op::rest) ≝
 (case op of
    None ⇒ one (a + off, Word 0w)
  | Some (FibTree k v l) ⇒
    one (a + off, Word k) * fts_mem (ann_fts 0w [FibTree k v l])) *
  reb_array_mem a (off + 1w) rest
```

Finally, a very important achievement for the separation logic verification is the

reuse of the rebalancing array in multiple function calls. Otherwise the extract minimum operation needs to reallocate memory in each execution. Even though the array is capped to 196 words, the extract minimum operation is executed n times during the execution of the Dijkstra loop.

This is unnecessary fragmentation of the memory and strictly unwanted in Pancake, where memory cannot be allocated during the execution of the program.

Hence, we verify that our implementation can reuse the rebalancing array. This verification is done on the algorithm level. Due to the Definition 23 the result of the algorithm level implies that the rebalancing array is also empty on the memory level.

Theorem 13 *The rebalancing array is equal to the rebalancing array before the execution.*

$\vdash$ fts_reb emp_rl *fts* = (*fts′*, *rl*, *flag*) $\Rightarrow$ emp_rl = *rl*

6

Discussion & Conclusion

This chapter concludes the thesis. The first section will be a short summary of the project and comparision of the completed work with the goals of the project. The second section gives a small insight into HOL4. In particular, we discuss our learning experience and starting difficulties. Afterward, there is a section about related work which focuses on other Fibonacci heap implementations that use a separation logic framework. Finally, we present ideas for possible future work.

6.1 Summary

This project verifies the extract minimum operation of the Fibonacci heap. The design of the Fibonacci heap uses monolithic nodes which hold all associated pointers and data. With the monolithic nodes and a structural approach of the verification strategy, we are able to separate the verification into four different levels. The separation of levels allows a clear verification for a rather complex data structure.

The verification is carried out in HOL4 and uses the memory implementation of the Pancake compiler. Hence, after verifying a memory level implementation, we can show that there exists a similar Pancake program. Since Pancake uses a verified compiler, our verification approach concludes in an end-to-end verification.

Between the verification levels we needed multiple refinements. These refinements go from a finite map to a forest of trees and end in the separation logic representation of monolithic nodes. These refinements are present in the logical definitions (fib_heap_inv), proof setups (ts_to_fhts), and entire code parts (fib_heap_set_parent).

The thesis aimed for the following goals:

G1: A structural separation of different verification levels for the verification of the Fibonacci heap.

G2: Verification of the functional correctness for a system-level Fibonacci heap implementation.

G3: The showcase of a formal end-to-end verification for Pancake.

Even though the project talks about the major refinements, we verified all implemented functions up to the memory level (**G1**) and showed on end-to-end proof for the melding operation (**G3**). This one end-to-end proof serves as an example for

the other functions. Unfortunately, we could not achieve **G2** completely, since we ran out of time to verify the decrease-key operation.

6.2 Lessons Learned

During the early stages of the project, HOL4 was new to me. Hence, we need to learn the design and proving logical statements during the start of the project.

This was a difficult process since learning the usage of an interactive theorem prover differs from learning programming languages. Instead, one should rather think about HOL4 as a logical system that is embedded in a programming language environment. For instance all of our definitions are logical statements. However, these definitions can be used in an expression that expand or evaluate them. So the HOL4 framework simulates a programming environment and then allows reasoning about the constructed definitions in higher-order logic.

This concept was not intuitive to me at the start. We approached the tool with a try and error method that worked out worse than expected since error messages from HOL4 can be cryptic without any experience. If an error occurs it can be one of the following:

1. The type inference of HOL4 made an error and one should specify the type in the definition.
2. Some parentheses are set at the wrong place which leads to a malformed type expression.
3. One does not understand the underlying definition, data type or HOL4 theory correctly and should read it up in the HOL4 description manual [13] or the HOL4 reference pages [8].

Especially in the first stage of the project the description manual [13] was the key to understand and use HOL4.

An additional difficulty was the finite map data type. Maps are a common data structure in programming languages, but the logic behind the data type was new to me. During programming the interaction between the data structure and the program is usually adding or looking up an element. On a logical level, the reasoning becomes more comprehensive.

In our logical lemmas, we need to reason about its domain, the disjointness of two maps, the splitting of a map, the insertion of an element, the removal of an element, its construction from a list, and the union of two maps. The majority of our logical lemmas include multiple of the above operations and propositions simultaneously.

After a significant amount of verification of finite maps, we incorporated the style of forming our proof to a state that enables the use of the fmap_eq_flookup theorem. This changes the proof style from reasoning about a plain finite map to looking up an element in the map. Then, the proof is more intuitive to programming and usually easier.

Finally, another difficulty was the monolithic node in separation logic. In the verification of the memory level one needs to expand these nodes such that the system code can perform read and write operation. Due to the large size of our monolithic node, such heap expression can easily fill out more than half the screen.

Moreover, our code uses other memory functions and inductions frequently. Each time one introduces an implication with memory, one must show that the previous memory and the memory of the antecedent is identical. However, usually those two expression differ and one needs to apply some lemmas to make the heap cells match. This usually means that the proof becomes cluttered with heap expressions and finding the mismatch between the heap cells is a needle-in-a-haystack problem.

6.3 Related Work

Attempts have been made before to verify the Fibonacci heap. Both Stüwe and Griebel have attempted to verify the Fibonacci heap. Stüwe focused on the extract minimum operation [2], while Griebel focused on the decrease-key operation [3]. Both used the ISABELLE/HOL framework, with Griebel attempting to use Stüwe's setup for the decrease-key operation.

However, despite using Stüwe's setup, their implementations and verifications are not compatible. For the decrease-key operation, Griebel requires links between parents and children. These links were omitted from Stüwe's verification setup.

Furthermore, their framework does not distinguish between multiple verification layers in a clear manner, combining the logical, algorithmic and memory levels into a single verification. This makes their proofs significantly more complicated and difficult to follow.

Since both use a separation logic framework, their nodes include pointers, as well as a cycling double-linked list with the relevant lemmas. Our approach adopts this since our Fibonacci heap is represented in the algorithm and at memory level as a cycling double-linked list.

Both of their approaches have clear focuses. Stüwe verifies functional correctness and time complexity for their extract minimum operation, while Griebel uses a map-based implementation to only verify the correctness of the decrease-key operation.

By contrast, our project verifies functional correctness by separating logical transformations and implementations into four levels. Unique is our memory and concrete implementation level which is not present in both previous works even though they use a separation logic framework. Stüwe and Griebel still use high-level data types for their implementations. Instead, our system-level implementation only uses words, addresses, and memory updates. For this implementation we design a monolithic nodes with an intrusive list, which Griebel deems impossible in ISABELLE/HOL.

Additionally, our monolithic node uses parent and child pointers which require additional refinement at the memory level. These pointers are missing in Stüwe's implementation. Also, our system-level implementation requires the rebalancing

array to be reusable after execution. This property is crucial for a system-level implementation.

Finally, our approach to verifying operations on the Fibonacci heap at multiple levels involves separating the formal reasoning. Our refinement proof between the logical and algorithm levels is similar to Hoare logic, while our proof between the algorithm and memory levels uses separation logic to reason about memory. This separation of logical and memory transformations enables smooth end-to-end verification from the logical level to the concrete implementation.

To our knowledge, we are the first to publish foundational end-to-end verification of Fibonacci heap operations in a system-level programming language.

6.4 Future Work

Unfortunately, our project is only partially finished. Initially, we had intended to undertake the comprehensive verification of the Fibonacci heap with a Dijkstra algorithm for shortest paths. The final missing operation for the Fibonacci heap is the decrease-key operation. We have already theorized about verifying the operation using Zippers [14]. Therefore, this verification task could be taken on in future work.

Additionally, one could try to analyze our separation logic framework in more detail. While the framework models memory in an intuitive way, we struggled to separate proof parts into several lemmas. It might be possible to extend the theorem set for the framework to make separation logic verification with a comprehensive data structure easier.

Finally, the Pancake compiler is a relatively new and unexplored compiler. Currently, its only use case is end-to-end verification of system-level implementations. However, it lacks both a security inspection and a performance evaluation for cryptography.

Such a performance evaluation could use well-known algorithms and data structures for example, a fully verified Fibonacci heap with a Dijkstra algorithm for finding the shortest path.

Bibliography

[1] Junming Zhao, Miki Tanaka, Johannes Åman Pohjola, Alessandro Legnani, Tiana Tsang Ung, H Truong, Tsun Wang Sau, Thomas Sewell, Rob Sison, Hira Syeda, et al. Verifying device drivers with pancake. *arXiv preprint arXiv:2501.08249*, 2025.

[2] Daniel Stüwe. Verification of fibonacci heaps in imperative hol. 2019.

[3] Simon Griebel. Verification of the decrease-key operation in fibonacci heaps in imperative hol. 2020.

[4] Michael L Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

[5] Edsger W Dijkstra. A note on two problems in connexion with graphs. In *Edsger Wybe Dijkstra: his life, work, and legacy*, pages 287–290. 1959.

[6] Johannes Åman Pohjola, Hira Taqdees Syeda, Miki Tanaka, Krishnan Winter, Tsun Wang Sau, Benjamin Nott, Tiana Tsang Ung, Craig McLaughlin, Remy Seassau, Magnus O Myreen, et al. Pancake: verified systems programming made sweeter. In *Proceedings of the 12th Workshop on programming languages and operating systems*, pages 1–9, 2023.

[7] Konrad Slind and Michael Norrish. A brief overview of hol4. In *International Conference on Theorem Proving in Higher Order Logics*, pages 28–32. Springer, 2008.

[8] HOL4 Contributers. Hol4 reference page. `https://hol-theorem-prover.org/trindemossen-2-helpdocs/help/`, 2026. Last accessed: 24h March 2026.

[9] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[10] John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th annual IEEE symposium on logic in computer science*, pages 55–74. IEEE, 2002.

[11] Thomas Tuerk. A separation logic framework for hol. Technical report, University of Cambridge, Computer Laboratory, 2011.

[12] Magnus O Myreen. Formal verification of machine-code programs. Technical report, University of Cambridge, Computer Laboratory, 2009.

[13] Michael Norrish, Konrad Slind, Hasan Amjad, Jens Brandt, Anthony Fox, Mike Gordon, Peter Homeier, Joe Hurd, Chun Tian, and Tjark Weber. *The HOL System Description*. HOL4 Theorem Prover Project, trindemossen-2 edition, 2025. HOL4 system documentation.

[14] Gérard Huet. The zipper. *Journal of functional programming*, 7(5):549–554, 1997.