DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Verification of Fibonacci Heaps in Imperative HOL

Daniel Stüwe

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Verification of Fibonacci Heaps in Imperative HOL

Verifikation von Fibonacci Heaps in Imperative HOL

Author:	Daniel Stüwe
Supervisor:	Prof. Tobias Nipkow, PhD
Advisor:	Dr. Peter Lammich
Advisor:	Maximilian P. L. Haslbeck, M. Sc.
Date:	April 15, 2019

Declaration

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

Munich, 12 April 2019
Daniel Stüwe

Acknowledgement

I would like to thank my advisors, Peter and Max, for their spontaneous and helpful advice guiding me through this thesis. I would also like to thank my supervisor Prof. Nipkow for awakening my enthusiasm for formal verification. And finally, I want to thank my friends and my family who supported me throughout my entire studies.

Abstract

Fibonacci heaps are one of the most important implementations of priority queues since their discovery led to a lowered worst-case runtime of certain greedy algorithms, e g. Dijskstra's or Prim's algorithm. The necessary proofs employ an amortized runtime analysis. Apart from this, they are considered in general to be one of the more complex data structures. Therefore, we formally verify Fibonacci heaps in this thesis. Our approach is to define and prove correct a functional implementation first, which is then refined to an imperative version. To our knowledge, this is the first formal verification of Fibonacci Heaps including functional correctness, imperative implementation and amortized runtime for the all operations except for decrease-key.

Contents

1 Introduction

In computer science, formal verification is the application of mathematical techniques to prove the correctness of a given algorithm or data structure. Commonly, the usage of a theorem prover or model checker is implied by this term [1]. Correctness is defined by a formal specification, which is a set of precisely defined properties that have to be fulfilled.

In this thesis, Fibonacci heaps are the object of such a formal verification. They were discovered by Tarjan and Fredman [2]. Fibonacci heaps implement a mergeable priority queue that is used in prominent algorithms like Dijkstra's algorithm or the algorithm of Prim [3].

Our approach to verify Fibonacci heaps is to define and prove correct a potentially inefficient functional implementation which is subsequently refined to an imperative version. For this, the theorem prover Isabelle/HOL is used, which already provides a comprehensive library of formally verified theories including a formalization of separation logic [4, 5]. This formalization is based on a framework for the specification of imperative, i. e. heap manipulating, algorithms and data structures called Imperative HOL [6]. Lammich and Lochbihler already used this formalization to verify a multitude of imperative programs [7]. They implemented – amongst other tools – a verification condition generator that works with Hoare triples and assertions in separation logic.

We verify the following operations on Fibonacci heaps: empty, singleton, merge, insert, consolidate and pop-min. This verification includes two functional versions; one is using separate keys for prioritization and one that is directly using the order on the data. We prove both correct. This is motivated by the fact the most text-book descriptions of heaps use separate keys, while most practical implementations ordering relations on the data. Moreover, an imperative implementation is provided for each version. Also, a relation between functional and imperative operations and structures is proven. Unfortunately, the verification of decrease-key was out of scope of our approach.

Additional to the formal correctness proof, we verify the runtimes of the listed operations. This is especially interesting since the runtime analysis is amortized [8]. For this purpose, we use an adapted form of separation logic that is extended with the notion of time-credits. This method was first proposed by Atkin [9]. Zhan and Haslbeck [10] extended Imperative HOL and conducted runtime analyses for multiple complex algorithms and data structures, for instance Karatsuba's algorithm, splay trees etc.

For the imperative implementation, circular doubly linked lists are defined and proven to refine functional lists. For the operations of these lists as well as of Fibonacci heaps, the runtimes have been verified using Imperative HOL with time. Based on the priority queue interface, the heapsort algorithm is implemented abstractly, then proven correct, and finally refined to an imperative program using Fibonacci heaps. Using the amortized runtime of pop-min, we verify that heapsort based on Fibonacci heaps has an asymptotic runtime of $\mathcal{O}(n \log n)$ where n is the number of elements.

To our knowledge, this is the first formal verification of Fibonacci Heaps including functional correctness, imperative implementation and amortized runtime for all operations except for decrease-key.

We begin this thesis with a description of the theorem prover Isabelle and its underlying logic. Then, concepts of program semantics are introduced, particularly Hoare triples and separation logic. The presentation in that section is based on the framework Imperative HOL with time whereon our formal verification of imperative Fibonacci heaps is based. The next section focusses on methods for runtime analysis, concluding the description of the foundations. Thereafter, priority queues are briefly discussed as an abstract datatype. Then, we describe Fibonacci heaps followed by details of the formal verification of each operation listed above. Concluding this thesis, the heapsort algorithm is described to give an example for the application of Fibonacci heaps and the proven theorems.

2 Isabelle/HOL

Isabelle is a proof assistant, also called an interactive theorem prover. This kind of programs processes proof documents, which are a sequence of definitions, theorems followed by a formal proof and comments. In a formal proof, each transformation of the proof goal has to reference a previously formally proven theorem, an axiomatized rule or an assumption.

As a proof assistant, Isabelle consists of two parts: a generic theorem prover, and an integrated development environment (IDE) for interactive theorem proving.

That theorem prover is generic in the sense that it supports multiple logics like classical first-order logic, different sequent calculi etc. This is achieved by separating a core logic, a kernel, upon which the different logics are axiomatized. This kernel is implemented in Standard ML and is relatively compact compared to traditional theorem provers like Coq. Small kernels have the advantage that humans can more easily verify them. [11].

To facilitate the development of formal proofs, certain automatic and semi-automatic procedures have been implemented on top of Isabelle's kernel. These procedures are called proof tactics in Isabelle. They apply previously registered rules and theorems to the proof goal. Not every proof tactic solves a proof goal completely; some tactics even split the proof goal into subgoals usually by applying introduction rules.

Isabelle's IDE supports the development of proof documents by providing syntactical help regarding keywords, styles and frequent patterns. Since the recently worked-on proof document is continuously checked by the theorem prover, the IDE can show intermediate subgoals of a proof that may remain after an application of a semi-automatic proof tactic. Other helpful features of the IDE are the possibility to view types, to navigate to definitions and to search for theorems and terms.

In the following parts of this section, we will briefly describe Isabelle's core logic; the object-logic HOL including datatypes and recursive functions; and the structured proof language Isar.

2.1 Isabelle's Meta-Logic

Higher-Order Logic (HOL) can be seen as a generalization of second-order predicate logic, which is an extension of the well-known predicate logic that is sometimes also called first-order logic. HOL's fundamental idea is that one cannot only quantify unary or binary variables but variables of any fixed arity, i. e. one can define properties of properties. Foundational for HOL and its computational implementation in theorem provers like Isabelle is the lambda calculus as discovered by Church [12] and the findings of Curry [13] and Howard [14].

Isabelle implements such an intuitionistic logic of higher order as its fundamental proof calculus. Paulson calls this calculus Isabelle's meta-logic [11]. This meta-logic is based on the simply typed lambda calculus with type classes – similar to those in Haskell [15]. Type classes allow constraining types to fulfil certain properties, e. g. being a total order or a lattice, forming an Abelian group etc. The Hindley–Milner type system served as a model for the implementation in Isabelle [11]. Hence, Isabelle can infer types automatically.

Isabelle's meta-logic provides the following four connectives and their corresponding intuitionistic rules:

- Implication $P \Longrightarrow Q$, expressing that P entails Q,
- quantification denoted by $\bigwedge x.\, P$, meaning that Q holds for arbitrary x,
- definitional equivalence $P \equiv Q$, which is accepted if P and Q can be (syntactically) unified, and
- conjunction P &&& Q that expresses that P and Q hold.

Based on this meta-logic, multiple so called object-logics are axiomatized for example propositional logic, various sequent calculi, Zermelo–Fraenkel set theory (ZF) etc.

2.2 HOL

This thesis is based on the object-logic HOL. It extends Isabelle's meta-logic, which is also of higher order as described above. "[The axiomatization of this object-logic] is based on Gordons' HOL, which itself is based on Church's original paper." [15]. A complete description of HOL can be found in [16]. Henceforth, the abbreviation HOL is referring to this object-logic of Isabelle and not to higher-order logic in general.

2.2.1 Embedding in Isabelle's Meta-Logic

HOL defines its own types, values and connectives like $\longrightarrow$, $=$, $\neg$, True, $\forall$, THE (similar to Hilbert's ϵ operator) etc. These may closely correspond to their meta-logic equivalents but are not the same. For example, HOL's implication $\longrightarrow$ is axiomatized as following:

$$\frac{\begin{array}{c}[P]\\ \vdots \\ Q\end{array}}{P \longrightarrow Q}\ \text{impI} \qquad\qquad \frac{P \quad P \longrightarrow Q}{Q}\ \text{mp}$$

In Isabelle's meta logic notation, this looks like:

axiomatization where
impI: "(P $\Longrightarrow$ Q) $\Longrightarrow$ P $\longrightarrow$ Q" **and** mp: "P $\longrightarrow$ Q $\Longrightarrow$ P $\Longrightarrow$ Q"

This means that the meta-logic entailment denoted by $\Longrightarrow$ and HOL's implication connective $\longrightarrow$ are in essence semantically the same, but obviously syntactically different and moreover different in type which is hidden in the above axiomatization by implicit type conversion. For this reason, both forms may appear in formal proofs.

Opposed to Isabelle's meta-logic, HOL as is not intuitionistic since the law of excluded middle is also axiomatized:

axiomatization where True-or-False: "(P = True) $\vee$ (P = False)"

2.2.2 Datatypes and Functions

Based on this elementary logic, HOL provides a framework to define (co-)datatypes without any further axiomatization. HOL's datatypes are based on a theory of bounded natural functors [17]. Furthermore, there is an additional framework for defining recursive functions over datatypes. Both frameworks come with mature tooling that automatically proves for each definition of a datatype respectively function well-definedness (e. g. termination, non-emptiness) and a rich set of properties. Of particular note are the induction rules tailored precisely to the datatype or function.

At this point, we will give an example to demonstrate the datatype framework in Isabelle. For this, a polymorphic type 'a tree is defined with two possible forms: A node with two children and a value of type 'a and an empty leaf as a bottom element. Moreover, the framework allows to define selector functions. Here, we declare the selector val to access the value stored in the node.

datatype 'a tree = Leaf | Node "'a tree" (val: 'a) "'a tree"

The notation, as one can see, is resembling Standard ML and related programming languages. The same is true for functions using pattern matching and currying as in the following definition:

fun to-list :: "'a tree ⇒ 'a list" **where**
"to-list Leaf = []" |
"to-list (Node l x r) = (to-list l) @ [x] @ (to-list r)"

The function to-list flattens a tree to a list, which is a predefined datatype of the library accompanying HOL in Isabelle. Because to-list is primitively recursive, termination is proven automatically by the framework. If the procedure could not have constructed a termination proof itself, the definition would have been rejected.

Please take note that application of HOL functions is implicit and therefore not denoted by enclosing parenthesis. Thus, to-list t expresses function application of to-list the argument t which is written in many non-functional programming languages as in traditional mathematics like to-list (t).

Alongside well-definedness, an induction scheme to-list.induct is proven by HOL's function framework. It is the same as the scheme tree.induct that is provided by the datatype framework for 'a tree:

$$\bigwedge P\ t.\ P\ \mathsf{Leaf} \Longrightarrow (\bigwedge l\ x\ r.\ P\ l \Longrightarrow P\ r \Longrightarrow P\ (\mathsf{Node}\ l\ x\ r)) \Longrightarrow P\ t$$

This scheme may look slightly awing to the ones unfamiliar with formal theorem proving. However, it is actually following straightforward from the definition of the datatype: To prove that proposition P holds for an arbitrary tree t, one has to prove a base case and a induction step. In the base case, one has to show that P holds for the bottom element Leaf. In the induction step, it is assumed that P holds for arbitrary, but fixed subtrees l and r. Using these assumptions, a formal proof must be given that P also holds for Node l x r where x is also arbitrary.

The following proof tree presents this scheme more graphically:

$$\frac{\mathsf{P\ Leaf} \qquad \begin{array}{c}[\mathsf{P\ l,\ P\ r}]\\ \vdots \\ \mathsf{P\ (Node\ l\ x\ r)}\end{array}}{\mathsf{P\ t}}\ \mathsf{tree.induct}$$

2.2.3 Résumé

In Isabelle, a collection of formal definitions and proven theorems is called a theory. Compared to other object-logics, HOL comes with the largest library of theories. For example, theories used for this project formalize calculus, groups and rings, orders, lattices, the Landau notation, functional data structures like lists and, naturally, imperative program semantics.

With HOL's classical logic on one side and its capabilities of function definition in the style of functional programming languages on the other, one refers to the proof assistant Isabelle and its object-logic HOL as one proof system by the composition Isabelle/HOL.

To complete this section, we like to quote a statement by Paulson and Nipkow from their introduction to Isabelle/HOL that summarizes its nature [18]:

"HOL = Functional Programming + Logic"

2.3 Formal Proofs

As mentioned in the introduction of this section, proof tactics are used in Isabelle to apply logic rules and rewrite steps [19] to a proof goal. However, sequentially applying those tactics can lead to hardly comprehensible formal proofs. Hence, a proof languages is provided to construct complex formal proofs.

In the following, both methods will be introduced in more detail and contrasted. Thereafter, the above tree example will be continued to demonstrate how to incorporate proof tactics into those structured proofs, concluding the presentation.

2.3.1 Proof Tactics

There is a multitude of proof tactics available in Isabelle. On the one hand, there are tactics that apply a single, explicitly given rule, e. g. `subst` that applies one rewrite step. These basic tactics allow for a precise manual transformation of the proof goal.

On the other hand, powerful tactics are provided that try to solve a proof goal by alternating the application of introduction and elimination rules, rewrite steps, and linear arithmetic. For example, fastforce even backtracks on introduction rules.

For the purpose of demonstration, a formal proof for an elementary fact is given below by sequentially applying proof tactics. In this traditional script-style proof, the very basic tactics rule and erule are used which just apply a single intro or elimination rule:

```
lemma "(A ⟶ B ⟶ C) ⟹ (A ∧ B ⟶ C)"
 apply (rule impI)
 apply (rule impI)
 apply (erule conjE)
 apply (erule impE)
  apply assumption
 apply (erule impE)
  apply assumption
 apply assumption
 done
```

Unfortunately, this is a barely understandable formal proof – even for proficient users of Isabelle/HOL. Since Isabelle's interactive proof style resembles (higher-order) natural deduction, one can visualize the proof above by constructing a proof tree that reflects each proof step:

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{A \Longrightarrow B \Longrightarrow A} \qquad \dfrac{\overline{A \Longrightarrow B \Longrightarrow B} \qquad \overline{C \Longrightarrow A \Longrightarrow B \Longrightarrow C}}{B \longrightarrow C \Longrightarrow A \Longrightarrow B \Longrightarrow C}\;\text{impE}}{A \longrightarrow B \longrightarrow C \Longrightarrow A \Longrightarrow B \Longrightarrow C}\;\text{impE}}{A \longrightarrow B \longrightarrow C \Longrightarrow A \wedge B \Longrightarrow C}\;\text{conjE}}{A \longrightarrow B \longrightarrow C \Longrightarrow A \wedge B \longrightarrow C}\;\text{impI}}{(A \longrightarrow B \longrightarrow C) \longrightarrow (A \wedge B \longrightarrow C)}\;\text{impI}$$

By inspecting this tree, one may notice when the implication elimination rule impE is applied that the proof goal splits, which is resulting into two independent proof goals. That is hinted in the proof script by a slight indentation. Nonetheless, these two new proof goals are not observable in the above proof script without using the interactive capabilities of Isabelle's IDE in contrast to the proof tree. However, proof trees are a hardly better notation for formal proofs than those scripts.

2.3.2 Structured Proofs — Isar

In order to avoid such long proof scrips, a structured proof language named Isar was developed for Isabelle by Wenzel [20]. It focuses on human-readability, and therefore it deploys keywords and structures analogous to expressions used in informal proofs written in natural language. Hence, Isar proofs are to some extent intuitively understandable to those accustomed to the language of mathematics, especially to natural deduction. Isar is structuring formal proofs by providing the capability to split complex proof goals into easier human-readable subgoals that may furthermore be easier solvable by the automatic proof tactics.

Complex Isar proofs are enclosed by the keywords **proof** and **qed** or by braces. They resemble the textbook proofs in the style of: "From Q, we conclude Q' (...) and finally derive our result P."

```
proof
  assumes Q
  then have fact1: Q' by proof-tactic
  then have fact2: P'
    using some-lemma by proof-tactic₂
  ...
  then show P by proof-tactic₃
qed
```

```
{ assumes Q
  then have fact1: Q' by proof-tactic
  then have fact2: P'
    unfolding some-definition
    by proof-tactic₂
  ...
  then have P by proof-tactic₃
}
```

Assumptions are naturally noted after the keyword **assume**, and with the keyword **have**, new subgoals are introduced. One can chain facts in Isar using different styles. In the proof below, a simple form is used with **then**. This adds the previously proven fact to the assumption list of the following to-be-proven subgoal.

To demonstrate this, the example from above will be continued. However, the formal proof is denoted this time in Isar:

```
lemma "(A ⟶ B ⟶ C) ⟹ (A ∧ B ⟶ C)"
proof
  assumes "A ⟶ B ⟶ C"
  then have "A ⟹ B ⟶ C" ..
  then have asm: "A ⟹ B ⟹ C" ..

  { assumes "A ∧ B"
    then have C proof
      assumes A and B
      then show C by fact — Refers to asm
    qed
  }

  then show "A ∧ B ⟶ C" ..
qed
```

Naturally, this example is actually trivial and therefore no structured proof is needed.

The statement can be proven automatically by all most all available tactics, e. g. the tableau prover blast [21]:

lemma "(A ⟶ B ⟶ C) ⟹ (A ∧ B ⟶ C)" **by** blast

Continued Tree Example Isar and the proof tactics do not oppose but complement each other, as can be seen in the following example. For this, we define a function on trees that swaps its left and right branch recursively.

fun mirror :: "'a tree ⇒ 'a tree" **where**
"mirror Leaf = Leaf" |
"mirror (Node l x r) = Node (mirror r) x (mirror l)"

Then, we prove by induction of the structure of t that the tree is as before if the function mirror is applied twice. To achieve this, we invoke an Isar induction proof. It is automatically deducted from the type of t that the induction scheme tree.induct must be used:

lemma mirror-mirror-eq-id: "mirror (mirror t) = t"
by (induction t) auto

The proof tactic auto is, unlike the name suggests, only semi-automatic. It tackles the proof goal with term rewriting, introduction and elimination rules, case splitting and some linear arithmetic. Possibly, it leaves some subgoals that it could not prove entirely, which is usually the case for complex statements. In the above example, auto is applied to both cases that arise from the induction scheme and solves them completely.

Isar supports multiple proof techniques like case distinction, (co-)induction, obtaining and fixing variables, unfolding definitions, etc. An exhaustive list can be found in Wenzel's PhD thesis [20].

3 Program Semantics

The Isabelle libraries this thesis is based on are Imperative HOL [6] and Imperative HOL with Time [10]. In the following, the latter is presented to introduce the used concepts of program semantics since it is an extension of the former.

3.1 Introduction to Imperative HOL with Time

The semantics of an imperative program can be described in multiple fashions: operational, denotational or axiomatic [1]. In this thesis, methods will be used to formalize and verify programs that stem from axiomatic program semantics. However, Imperative HOL does not define a meaningless syntax first that is given a semantics afterwards. On the contrary, the framework provides a polymorphic heap monad Heap very similar to Haskell's ST monad [22]; and commands that manipulate the heap are directly encoded as computable functions in HOL that map heaps to heaps (ignoring time costs). Thus, these commands already have a well-defined mathematical meaning in Isabelle/HOL by their very definition.

Moreover, Isabelle/HOL also provides a monad syntax that is heavily inspired by Haskell, and which takes care of temporarily introduced variables in the program flow. An example of this syntax and its transformation into a sequence of monadic binds is given below:

```
definition "program y = do {
  x ← subprogram₁ y;
  subprogram₂;
  subprogram₃ x
}"
```

```
definition "program y =
  subprogram₁ y ⪢= (λx.
  subprogram₂ ⪢= (λ_.
  subprogram₃ x))
```

Concluding, programs are just presented as usual HOL functions with result type 'a Heap, and for this reason, they are inherently semantically defined. Details about this approach can be found in the original paper describing Imperative HOL [6].

In Imperative HOL, Hoare logic is primarily used to describe the effects of imperative programs. Nevertheless, as explained above, the rules of Hoare logic are not defining but reflecting the semantics of a command or program with one notable exception: Time-steps are indeed defined axiomatically because there is no notion of computation time embedded in HOL. This time-awareness is formalized in Imperative HOL with time.

3.2 The Heap and its Monad

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

```
datatype 'a array = Array addr
datatype 'a ref = Ref addr
```

Imperative HOL introduces a record type heap that maps addresses to values encoded as natural numbers. It differentiates between references and arrays. Moreover, the heap is indexed by the type representatives of the values that are stored in it, as one can see in the listing above. This ensures that natural number which is stored on the heap is always a valid encoding of the actually value, which is of course type dependent. The heap limit lim is a counter variable indicating the next unused address.

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

The monad 'a Heap is then defined, as printed above, to be a wrapped function that maps a heap h to a triple consisting of a computed result x of type 'a, the manipulated heap h' and the number of time-steps n that this operation has taken. If the computation is aborted, the result would be None instead of Some (x, h', n).

3.2.1 Dereferencing — An Example

To show how this set-up is used, we like to explain exemplarily dereferencing. The function get, shown below, takes two arguments of type heap and 'a ref respectively. First, the address is unwrapped out of the reference. Then, the stored value in the refs section of the heap is retrieved using this unwrapped address and the type representative of 'a. Finally, the stored value – a natural number – is decoded.

definition get :: "heap ⇒ 'a::heap ref ⇒ 'a" **where**
"get h = from-nat ∘ refs h TYPEREP('a) ∘ addr-of-ref"

To make this elegantly usable, this function is wrapped into the polymorphic Heap monad:

definition lookup :: "'a::heap ref ⇒ 'a Heap" ("!_" 61) **where**
"lookup r = Heap (λh. Some (get h r, h, 1))"

As one can see in the definition of lookup, the heap h is returned unchanged, the computed value is get h r, and time costs are declared to be 1. Moreover, lookup introduces the prefix notation !r, where r is a references.

Reading and writing to arrays is defined in the same fashion.

Imperative HOL is modelling the behaviour of Standard ML. Hence, there is no command to free any unused reference. Thus, Imperative HOL is primarily suitable to describe programs that use a managed heap with garbage collection.

3.3 Separation Logic with Time

For efficient reasoning about heaps, especially heap segments, one extends classical logic by introducing new connectives that are particularly suitable for the formulation of properties concerning the heap. This extended logic is called separation logic and was introduced by John C. Reynolds and Peter O'Hearn [4]. Quickly, it became popular: The original paper was cited more than 2.300 times by now [23].

There are two reasons for this: On the one hand, separation logic proven to be sufficiently flexible to elegantly describe data structures that use pointers, while on the other, the logic is restrictive enough to scale, s. t. reasoning about large, complex programs is possible. The presentation of separation logic in this section is heavily based on its formulation in Imperative HOL (with time) by Lammich, Meis, Zhan and Haslbeck [5, 10]. In this framework, separation logic is a shallowly embedded, i. e. it is not defined as a completely new object-logic but by usual functions in HOL.

3.3.1 Partial Heaps

In order to formulate separation logic, the concept of partial heaps must be introduced first. The key idea is that programs manipulate just parts of the heap and leave the rest unchanged. As one can see from the definition of the datatype below, a partial heap in Imperative HOL is described by a complete heap, a set of addresses and reserved time-credits.

datatype pheap = pHeap (heapOf: heap) (addrOf: "addr set") (timeOf: nat)

A partial heap is well-defined when the set of addresses is a (proper) subset of the ones of the complete heap. In Imperative HOL, this formalized by in-range (h, as) because this is true if and only if all addresses are below the heap limit. To associate two partial heaps, the relation relH indexed by a set of addresses as is defined. The lemma relH-D2 below clarifies its purpose: If two heaps h and h' stay in relation relH as, then for any address a in the set of addresses as, the heaps do agree on the stored value at this address:

lemma relH-D2:
  assumes "relH as h h'" **and** "a $\in$ as"
  shows "refs h t a = refs h' t a" **and** "arrays h t a = arrays h' t a"

For example, if h and h' relate as before and the address of the references r is in the set addresses as, then r references the same value in both heaps:

lemma relH-ref: "relH as h h' $\Longrightarrow$ addr-of-ref r $\in$ as $\Longrightarrow$ get h r = get h' r"

This implies further that h and h' only have to relate over as = { addr-of-ref r } to fulfil the conditions of this lemma. In all residual memory cells, the heaps could store values different from each other.

3.3.2 Assertions

Assertions are functions that assign partial heaps a boolean value.

type-synonym assn-raw = "pheap $\Rightarrow$ bool"

However, to only permit proper assertions, the type assn is defined. Such a proper assertion respects the heap limit, i. e. it is false for any partial heap that is not well-defined:

typedef assn = "$\{$A' . proper A'$\}$"

In order to obtain assertions of type assn, potentially improper assertions of type assn-raw are wrapped by the constructor Abs-assn. To unwrap an assertion P of type assn, the function Rep-assn can be applied which is the reverse of Abs-assn but only if the assertion P is a proper one.

The functions Abs-assn and Rep-assn, and their properties (e. g. the fact Abs-assn-inverse' below) are defined automatically by the procedure behind the type definition above if one has shown that the universe of assn is a non-empty subset of the one of type assn-raw.

Abs-assn-inverse' $\equiv$ $\bigwedge$ a. proper a $\Longrightarrow$ Rep-assn (Abs-assn a) = a

A partial heap h is called a model for an assertion A if and only if this assertion A is proper and if A applied to h evaluates to true, as stated in the Isabelle definition below. Modelling of assertions is denoted by h $\models$ A.

definition models :: "pheap $\Rightarrow$ assn $\Rightarrow$ bool" (infix "$\models$ " 50) **where**
"h $\models$ A $\longleftrightarrow$ aseval (Rep-assn A) h"

The term model as well as the notation stems from mathematical logic where a model is a structure that satisfies a given set of propositions. Analogously, if a heap satisfies an assertion, it is called a model for this assertion.

3.3.3 Fundamental Assertions

In the following, a number of basic assertions is presented that are essential for understanding separation logic in general and the theorems in later sections in particular.

The simplest assertions is emp. It describes an empty partial heap which is a heap that stores neither values nor time-credits:

lemma one-assn-rule: "h $\models$ emp $\longleftrightarrow$ timeOf h = 0 $\wedge$ addrOf h = $\{\}$"

A so-called pure assertion is basically a classical proposition b lifted to separation logic, i. e. it does not reason about the heap. It is written $\uparrow$b. If h $\models \uparrow$b, the heap must be empty and, furthermore, b must also hold:

lemma pure-assn-rule: "h $\models \uparrow$b $\longleftrightarrow$ (addrOf h = {} $\wedge$ timeOf h = 0 $\wedge$ b)"

The assertion top-assn is true for every well-defined partial heap; as stated in following lemma:

lemma top-assn-rule: "pHeap h as n $\models$ true $\longleftrightarrow$ in-range (h, as)"

Hence, it is aliased by true. It is frequently used if a heap is split and one of the resulting parts is not of interest anymore. In Imperative HOL, this usually means that partial heaps described by true could hypothetically be garbage collected or that asserted time-credits have not been used.

The opposite assertion bot-assn, thus aliased by false, cannot be modelled by any partial heap which follows immediately from its definition that is shown below:

abbreviation bot-assn :: assn ("false") **where** "bot-assn $\equiv \uparrow$False"

As mentioned, separation logic is extended classical logic for heaps. Hence, one can find several connectives that are conical embeddings like existential quantification $\exists_A$ (see lemma mod-ex-dist), disjunction $\vee_A$, conjunction $\wedge_A$ (see lemma and-assn-conv), and entailment $\Longrightarrow_A$ (see definition below).

lemma mod-ex-dist: "h $\models$ ($\exists_A$x. P x) $\longleftrightarrow$ ($\exists$x. h $\models$ P x)"

lemma and-assn-conv: "h $\models$ A $\wedge_A$ B $\longleftrightarrow$ (h $\models$ A $\wedge$ h $\models$ B)"

It is important to note that existential quantification $\exists_A$, disjunction $\vee_A$ and conjunction $\wedge_A$ result in a new assertion. In contrast, entailment on assertions $\Longrightarrow_A$ does not. Its return type is bool; as one can see from its definition.

definition entails :: "assn $\Rightarrow$ assn $\Rightarrow$ bool" (infix "$\Longrightarrow_A$ " 10) **where**
"(P $\Longrightarrow_A$ Q) $\longleftrightarrow$ ($\forall$ h. h $\models$ P $\Longrightarrow$ h $\models$ Q)"

So far, the rules from classical logic transfer to separation logic, for example De Morgan's laws; that true is entailed by arbitrary assertions; that false is neutral element of disjunction etcetera:

lemma entails-true: "A $\Longrightarrow_A$ true" **lemma** true-conj: "true $\wedge_A$ P = P"

Pure assertions also go well together with the classical connectives, as one can see for example in the following lemma:

lemma pure-conj: "$\uparrow P \wedge_A \uparrow Q = \uparrow(P \wedge Q)$"

However, this is not the case for the assertion emp. One of the very few simplification rules is the following:

lemma emp-and: "emp $\wedge_A$ emp = emp"

3.3.4 Assertions for References and Arrays

Next, we present the assertion for a single reference. It is denoted by $r \mapsto_r x$ and states that r points to x stored in the heap. Moreover, if this assertion holds, the partial heap only consists of specifically this single cell which is referenced by r. Also, no time-credits are saved and the references have to be valid otherwise the assertion would not be proper.

lemma sngr-assn-rule:
"pHeap h as n $\models r \mapsto_r x \quad \longleftrightarrow$
get h r = x $\wedge$ as = {addr-of-ref r} $\wedge$ addr-of-ref r $<$ lim h $\wedge$ n = 0"

Analogously, one can define an assertion for arrays $r \mapsto_a xs$. The main differences is that xs is a list instead of a single element:

lemma snga-assn-rule:
"pHeap h as n $\models r \mapsto_a x \quad \longleftrightarrow$
Array.get h r = xs $\wedge$ as = {addr-of-array r} $\wedge$ addr-of-ref r $<$ lim h $\wedge$ n = 0"

3.3.5 Separating Conjunction – Alias Star

Consider a partial heap h, s. t. $h \models r \mapsto_r x \wedge_A r' \mapsto_r y$. By using lemma and-assn-conv from above, one concludes $h \models r \mapsto_r x$ and $h \models r' \mapsto_r y$. From each of this, one can derive that the heap h only consists of exactly one cell because the references assertion $\mapsto_r$ requires this to hold. However, if there is just one cell, only one address points to it and thus reference r must equal r', which implies that the stored values x and y are also the same:

lemma "$h \models r \mapsto_r x \wedge_A r' \mapsto_r r\ y \Longrightarrow$ (card (addrOf h) = 1 $\wedge$ r = r' $\wedge$ x = y)"

Therefore, the connective $\wedge_A$ is usually not convenient for the description of partial heaps that contain multiple cells. Hence, for this purpose, a new connective is introduced: the separating conjunction denoted by $P \star Q$ for arbitrary assertions P and Q. Its notation is the reason why it is commonly pronounced star. This connective is the very innovation of separation logic. It splits the heap into two disjoint parts:

lemma mod-star-convE:
  assumes "pHeap h as n $\models$ P $\star$ Q"
  shows "$\exists$as1 as2 n1 n2. as = as1 $\cup$ as2 $\wedge$ as1 $\cap$ as2 = {} $\wedge$ n = n1 + n2 $\wedge$
    pHeap h as1 n1 $\models$ P $\wedge$ pHeap h as2 n2 $\models$ Q"

As one can see from the lemma above, splitting a partial heap h = pHeap h as n into two heaps h1 = pHeap h as1 n1 and h2 = pHeap h as2 n2 using the separating conjunction means the following: the sets of addresses as1 and as2 must be a partition of the original set as, i. e. as = as1 $\uplus$ as2 has to hold. The same applies for the time-credits where n = n1 + n2. The central idea is that assertion P now is only modelled by heap h1 and Q by h2. Since, these two heaps are disjoint, the assertion cannot interfere with each other since overlapping of the described heap parts is excluded.

Again, consider a partial heap h, however this time h $\models$ r $\mapsto_r$ x $\star$ r' $\mapsto_r$ y. This implies that there exists two disjoint heaps h1 and h2 partitioning h, s. t. h1 $\models$ r $\mapsto_r$ x and h2 $\models$ r' $\mapsto_r$ y. Each of these partial heaps consists of exactly one heap cell. Moreover, it does hold that r $\neq$ r'. The references r and r' are not the same because otherwise the heaps h1 and h2 would not be disjoint. Hence, r = r' would lead to a contradiction, as stated by the lemmata below. Besides, the same is true for arrays:

lemma sngr-same-false: "r $\mapsto_r$ x $\star$ r $\mapsto_r$ y = false"
lemma snga-same-false: "r $\mapsto_a$ x $\star$ r $\mapsto_a$ y = false"

Using the separating conjunction, it is possible to reason about specific parts of the heap which might be of special interest. Let assertion P $\star$ R describe a partial heap. It is possible to reason about P independently since it is ensured that the residual partial heap described by assertion R is disjoint to the one described by P. For this reason, assertion R, respectively its underlying heap, is called a frame because it simply surrounds the heap that models P. This explains the name of the following lemma.

lemma entails-frame: "P $\Longrightarrow_A$ Q $\Longrightarrow$ P $\star$ R $\Longrightarrow_A$ Q $\star$ R"

Furthermore, the separating conjunction forms a commutative monoid. Its neutral element is the assertion emp because the empty heap is trivially disjoint to any arbitrary heap. All of the properties necessary to fulfil these two statements are listed below and can be proven easily.

lemma assn-one-left: "emp $\star$ P = P"
lemma assn-times-comm: "P $\star$ Q = Q $\star$ P"
lemma assn-times-assoc: "(P $\star$ Q) $\star$ R = P $\star$ (Q $\star$ R)"

The star is also well-behaving together with the other assertions like pure ones:

lemma mod-pure-star-dist: "$h \models P \star \uparrow b \quad \longleftrightarrow \quad h \models P \wedge b$"
lemma p-c: "$\uparrow P \star \uparrow Q = \uparrow(P \wedge Q)$"

As shown in the following lemma, a well-defined partial heap can be divided into two of such and vice versa:

lemma top-assn-reduce: "$\mathsf{true} \star \mathsf{true} = \mathsf{true}$"

As mentioned before, the assertion true is commonly used for partial heaps that are not of interest anymore or if two branches of the program flow do not need the same amount of time-credits. For this purpose, a derived connective $\Longrightarrow_t$ is introduced:

definition entailst :: "assn ⇒ assn ⇒ bool" (infix "$\Longrightarrow_t$ " 10) **where**
"entailst A B $\equiv$ A $\Longrightarrow_A$ B $\star$ true"

Since emp $\Longrightarrow_A$ true and the fact that emp is the neutral element of the separating conjunction $P \star \mathsf{emp} = P$, one can conclude the following lemmata using the rule entails-frame that enables reasoning about separate assertions:

lemma entt-refl': "$P \Longrightarrow_A P \star \mathsf{true}$" **lemma** entt-refl: "$P \Longrightarrow_t P$"

However, neither the inverse of entt-refl' nor of $P \star Q \Longrightarrow_A P$ does hold in general which are critical facts for understanding separating logic completely.

So, in order to see the latter, one can unfold the definition of entailment on assertions concluding: If $h \models P \star Q$, then $h \models P$. Next, consider the case that $P = p \mapsto_r x$ and $Q = q \mapsto_r y$ where $p \neq q$. Then, the premise $P \star Q$ describes a heap with two disjoint cells referenced by p respectively q, thus $\mathsf{card}(\mathsf{addrOf}\ h) = 2$. However, the conclusion $h \models P$ implies that the very same heap contains just a single cell: $\mathsf{card}(\mathsf{addrOf}\ h) = 1$. That is a contradiction and thus the proposition is not true for arbitrary P and Q.

Nevertheless, $P \star Q \Longrightarrow_t P$ does hold in general: Any assertion Q entails true (see lemma entails-true in the previous section). Hence, $P \star Q$ entails $P \star \mathsf{true}$ and thus $P \star Q \Longrightarrow_t P$.

3.3.6 Time-Credit Assertions

The last fundamental assertion left to introduce is the time-credit assertion, which is denoted by $m where m is a natural number. This assertion describes a heap that contains exactly n time-credits and no heap cells:

lemma timeCredit-assn-rule: "$h \models \$n \quad \longleftrightarrow \quad \mathsf{timeOf}\ h = n \wedge \mathsf{addrOf}\ h = \{\}$"

The idea to adjunct partial heaps with time-credits introducing this additional assertion was first formulated by Atkey [9]. As one can see by the preceding lemma, if a partial heap models $\mathsf{P} \star \$\mathsf{m}$, then it contains at least m time-credits since P could require even more of those:

lemma mod-timeCredit-dest:
"pHeap h as n $\models$ P $\star$ \$m $\longleftrightarrow$ pHeap h as (n - m) $\models$ P $\wedge$ n $\geq$ m"

This leads to the following lemma that allows us to truly weaken the premise (thus the usage of $\Longrightarrow_t$) by ignoring any amount of assumed time-credits. Again, this is particularly useful if two branches of the program flow have to be joined that consumed different amounts of time-credits.

lemma auxiliar-time:
assumes "F $\Longrightarrow_A$ F'" and "b $\leq$ a" **shows** "F $\star$ \$a $\Longrightarrow_t$ F' $\star$ \$b"

3.4 Hoare Logic with Time

The notation of Hoare triples was proposed by Tony Hoare [24] hence their name. Hoare logic is a set of axiomatized Hoare triples. However, in Imperative HOL, these rules are derived as already mentioned previously. Moreover, all these rules are formulated for a Hoare calculus that uses forward reasoning.

A Hoare triple <P> c <Q> is defined in Imperative HOL with time like as follows: If h $\models$ P holds, there is a new heap h' as the result of successfully executing program c s. t. h' $\models$ Q holds. Execution is thereby solely the application of the heap transforming function wrapped in c :: 'a Heap to the heap h. Moreover, execution is only successful if enough time-credits are stored in h. The assertion P :: assn is called the precondition and Q :: 'a $\Rightarrow$ assn the postcondition, which takes the result of the computation as an input. The actual definition of a Hoare triple in Imperative HOL is more precise about the resulting heap and its properties; however, these details shall be omitted here.

As for entailment, it is also convenient for Hoare triples to discard parts of the postcondition. Therefore, a concise notation for weakened triples is introduced by the separation logic framework for Imperative HOL:

abbreviation hoare-triple' :: "assn $\Rightarrow$ 'r Heap $\Rightarrow$ ('r $\Rightarrow$ assn) $\Rightarrow$ bool" **where**
"<P> c <Q>$_t$ $\equiv$ <P> c <λr. Q r $\star$ true>"

As already said, a valid Hoare triple requires a successful run of the program as well as sufficiently many time-credits stored in the heap before execution. Thus, Hoare triples in Imperative HOL (with time) denote the total correctness of a program.

3.4.1 Basic Rules

The so-called frame rule printed below allows to reason about a part of the heap independently of its frame. This rule is significant for the capability of separation logic to scale.

lemma frame-rule:
assumes "<P> c <Q>"
shows "<P ⋆ R> c <λx. Q x ⋆ R>"

The following bind-rule, which is usually called rule of composition, is analogical to the frame rule in that sense that it permits to reason about parts of the program independently. Thus, it facilitates the reasoning about large programs.

lemma bind-rule:
assumes "<P> f <Q>" **and** "∀x. <Q x> g x <R>
shows "<P> f ⪢= g <R>"

Two further rules are part of the foundation of Hoare logic: A rule to strengthen the precondition and another to weaken the postcondition:

lemma pre-rule:
assumes "P' $\Longrightarrow_A$ P" **and** "<P> f <Q>"
shows "<P'> f <Q>"

lemma post-rule:
assumes "<P> f <Q>"
and "∀x. Q x $\Longrightarrow_A$ R x"
shows "<P> f <R>"

These rules can be combined. This combination is commonly referred by consequence rule. Additionally, these rules are also valid for weakened Hoare triples:

lemma cons-rulet:
assumes "<P> c <Q>$_t$" **and** PRE: "P' $\Longrightarrow_t$ P" **and** POST: "⋀x. Q x $\Longrightarrow_t$ Q' x"
shows "<P'> c <Q'>$_t$"

3.4.2 Rules for Simple Commands

In this part, rules will be given for some essential commands that may manipulate or read from the heap. These rules are derived from their definition, see dereferencing (3.2.1), and can be often proven fully automatically. All of them will be used in the following formalizations of this thesis.

The most basic is the rule for the return command, which assumes that a single time-credit is stored in the input heap.

lemma return-rule: "<$1> return x <λr. ↑(r = x)>"

To aid automatic proving, one can introduce explicit assertions in the code using the command assert. However, these commands will also be executed at runtime and hence costs one time-credit.

```
lemma assert-rule: "<↑(R x) ⋆ $1> assert R x <λr. ↑(r = x)>"
```

References For references, three operations are defined by Imperative HOL: creation ref x, dereferencing !r and manipulation r := x'. Each operation consumes one time-credit. As one can see from the lemma lookup-rule, reading is not manipulating the heap.

```
lemma ref-rule:
  "<$1>
    ref x
   <λr. r ↦r x>"
```

```
lemma lookup-rule:
  "<p ↦r x ⋆ $1>
    !p
   <λr. p ↦r x ⋆ ↑(r = x)>"
```

```
lemma update-rule:
  "<p ↦r y ⋆ $1>
    p := x
   <λr. p ↦r x>"
```

Arrays The same operations are also available for arrays. For creating an array, one can use the new command. It takes as input a natural number n and a value x that will be used to initialize each cell of the array. This operation costs n + 1 time-credits.

```
lemma new-rule: "<$(n+1)> Array.new n x <λr. r ↦a replicate n x>"
```

To access the n-th element of an array storing the sequence xs, it is presumed that n < length xs. Moreover, one time-credit is consumed by this operation.

```
lemma nth-rule: "<a ↦a xs ⋆ $1 ⋆ ↑(i < length xs)>
                  Array.nth a i
                 <λr. a ↦a xs ⋆ ↑(r = xs ! i)>"
```

The only heap manipulating command is Array.upd. It replaces the n-th element of an array and thereby consumes one time-credit.

```
lemma upd-rule: "<a ↦a xs ⋆ $1 ⋆ ↑(i < length xs)>
                  Array.upd i x a
                 <λr. a ↦a list-update xs i x ⋆ ↑(r = a)>"
```

As one can see in the lemma below, a command to simply read the length of an array is also provided.

```
lemma length-rule: "<a ↦a xs ⋆ $1>
                     Array.len a
                    <λr. a ↦a xs ⋆ ↑(r = length xs)>"
```

It is defined to run in constant time. This is similar for example to Java where the length is attached to each array to prevent invalid array access.

3.5 Circular Doubly Linked Lists

Fibonacci heaps, which will be described in a later section, use circular doubly linked lists. Therefore, they are an essential part of the entire formalization. However, circular doubly linked lists are a relatively simple data structure. Thus, they will serve in this section as an extended example to demonstrate the usage of Imperative HOL (with time). This includes their definition and the implementation of their operations. Subsequently, it is proven that they refine functional lists using Hoare logic.

3.5.1 Doubly Linked List Segments

The common approach to define imperative lists is to define first list segments and then, based on this, complete lists. This has two advantages: On the one hand, list segments can be used to define simple lists (i. e. non-circular) as well as circular lists. On the other hand, defining, for example, simple lists directly leads to inelegant proofs. The reason for this is that simple lists are already to specialised for induction since the definition includes fixed start and end elements.

At the beginning, one defines a single node of a doubly linked list segment. It stores a value of type 'a, which is accessible using the selector function val, and two references pointing to the previous list node respectively to the following one. These references are wrapped into the option type to model null pointers:

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

The assertion dll-seg below relates functional list to imperative doubly linked list segments. It takes six arguments: the first argument is another relation R that describes a refinement of the abstract values contained in the functional list to their imperative implementation. In the case of Fibonacci heaps, this relation will be used to refine functional Fibonacci trees to imperative ones.

```
fun dll-seg :: "_ ⇒ assn" where
  "dll-seg _ [] start start-pre  end end-next = ↑(start = end-next ∧ start-pre = end)"
| "dll-seg R (x#xs) (Some start') start-pre  end end-next =
      (∃_A start-next x'.   start' ↦_r Cell x' start-next start-pre
                          ⋆ R x x'
                          ⋆ dll-seg R xs  start-next (Some start')  end end-next)"
| "dll-seg _ _  _ _  _ _ = false"
```

The second argument of dll-seg is the functional list which is set in relation to their imperative refinement. The last four arguments (start, start-pre, end and end-next) are references to list nodes, again wrapped into the option type. The reference start points to the first node of the list segment, end respectively to the last one. As the name suggests, start-pre references the node prior to the first one and end-next the node following the last one. The references start-pre and end-next enable reasoning about the elements preceding respectively following the list segment.

In order to illustrate the application of separation logic, consider exemplarily the second equation in the definition of dll-seg where the functional list is non-empty. In this case, the start reference is non-null, i. e. start = Some start'. It points to the head of the list segment Cell x' start-next start-pre. The stored value x' on the heap is a proper refinement of x according to the given relation R. Succeeding the head node, a new shorter list segment begins. The head of the this shorter segment is the node referenced by start-next, which follows the current node.

Based on dll-seg, one can easily define simple doubly linked lists sdll. Since None represents the null pointer, one sets start-pre = None and end-next = None, which corresponds with the succeeding definition:

```
definition "sdll R start = ∃A end. dll-seg R start None end None"
```

Doubly linked list segments fulfil multiple practical properties, for example, they can be easily split into two distinct segments. Their distinctness follows from the usage of the separating conjunction:

```
corollary dll-seg-conc-split:
  "dll-seg R (xs@ys) start start-pre  start-next end-next' =
     (∃A k k-next.   dll-seg R xs  start start-pre  k k-next
                   ⋆ dll-seg R ys  k-next k  start-next end-next')"
```

Moreover, one can already define a fold operation that iterates over an doubly linked list segment which is specified by two references p and q.

```
partial-function (heap) dll-fold where
  "dll-fold f p q s = do {
      case (p, q) of
        (Some p', Some q') ⇒ do {
           cell ← !p';
           s ← f (val cell) s;
           if p' = q' then return s
           else do {
             let p = following cell;
             dll-fold f p q s }
          }
      | _ ⇒ return s
   }"
```

This fold operation is declared as a partial function, i. e. neither termination is proven nor any induction scheme. This is possible for functions with result type Heap since the Heap monad has a proper bottom element that denotes a failed computation. For this reason, such partial functions are still well-defined.

To reason about dll-fold in a larger program, one can use the following rule:

```
lemma dll-fold-rule-weak:
  assumes START: "P ⟹A dll-seg R xs p pp q qq ⋆ I [] xs s ⋆ F ⋆
                  ↑(xs = [] ⟹ qq = None) ⋆ $(1 + length xs ⋆ 2)"
  assumes STEP: "⋀xs1 x xs2 s x'.
                  <I xs1 (x#xs2) s ⋆ ↑(xs=xs1@x#xs2) ⋆ R x x' ⋆ F>
                   f x' s
                  < λs. I (xs1@[x]) xs2 s ⋆ F >t"
  assumes END: "⋀s. I xs [] s ⋆ F ⋆ true  ⟹t Q s"
  shows "<P> dll-fold f p q s <λs. Q s>t"
```

For this rule to hold, it is required by the START assumption that the precondition implies that p and q are the start and end of a list segment. Furthermore, it must imply that the loop invariant I holds before the iteration of the list begins. Iterating over the list costs two time-credits per list node. These costs have to be stored in the initial partial heap. The condition STEP specifies that the loop invariant I has to appropriately reflect how the operation f manipulates the state s. Assumption END requires the postcondition Q to be a consequence of the final loop invariant.

3.5.2 Definition and Selected Operations

As already mentioned, to define imperative Fibonacci heaps, circular doubly linked lists are needed, which are defined by the assertion cdll. An empty list is represented by None, an non-empty by Some p where p is the head node of a doubly linked list segment. One defines start = end-next and end = start-pre to obtain circularity:

```
fun cdll :: "_ ⇒ _ list ⇒ _ dll ref option ⇒ assn" where
  "cdll _ [] None = emp"
| "cdll R (x#xs) (Some p) = (∃Aend. dll-seg R (x#xs) (Some p) end end (Some p))"
| "cdll _ _ _ = false"
```

Moreover, for convenience, a type synonym is defined for references to doubly linked list segments wrapped into the option type:

```
type-synonym 'a cdll = "'a dll ref option"
```

In the following, some selected operations on circular doubly linked lists are presented starting with the creation of a singleton list.

The operation cdll-singleton constructs such a singleton list by allocating a provisional list node on the heap. For this, the ref operation is used, which returns a references p to this newly created heap cell. Next, start-pre and end-next are set to references node p itself. Hence, the predecessor of a singleton is the singleton itself. The same applies to its successor as well. Finally, the reference to the created node is wrapped and returned.

```
definition cdll-singleton :: "'a :: heap ⇒ 'a cdll Heap" where
  "cdll-singleton x = do {
    p ← ref (Cell x None None);
    p := Cell x (Some p) (Some p);
    return (Some p)
  }"
```

Using the rules ref-rule, update-rule and return-rule from the previous section, one can show that cdll-singleton takes three time-steps to execute. Moreover, it is required by the precondition that the actually stored value x' is refining its corresponding abstract counterpart x according to the relational assertion R:

```
lemma cdll-singleton-rule: "<R x x' ⋆ $3> cdll-singleton x' <cdll R [x]>"
```

Concatenating two non-empty list is done in three phases:

```
fun cdll-append :: "'a cdll ⇒ 'a::heap cdll ⇒ 'a cdll Heap" where
  "cdll-append None p = return p" |
  "cdll-append p None = return p"
| "cdll-append (Some p') (Some q') = do {
    p ← !p';
    q ← !q';
    assert ((≠) None) (previous p);
    let p-end' = the (previous p);
    p-end ← !p-end';

    assert ((≠) None) (previous q);
    let q-end' = the (previous q);
    q-end ← !q-end';

    p-end' := Cell (val p-end) (Some q') (previous p-end);
    q-end' := Cell (val q-end) (Some p') (previous q-end);

    p ← !p';
    q ← !q';
    p' := Cell (val p) (following p) (Some q-end');
    q' := Cell (val q) (following q) (Some p-end');

    return (Some p')
  }"
```

In the first phase, both head nodes of these lists as well as the end nodes are inspected and retrieved from the heap. This can be seen in the above definition in the first two code blocks that are separated by a blank line.

The end nodes are directly accessible from the head nodes by back references. This way, iterating through the list is avoided. In the second phase, the end nodes are updated to forward reference the opposing list heads. The third phase, updates the head nodes respectively, s. t. their back-references point the opposing list ends. It is important to reload the head nodes before updating in step three, since in the case of an singleton list, head and end nodes are identical. Hence, a head node could have been manipulated in phase two.

As the following rule shows, all three phases together cost 13 time-credits assuming the input references p and q point to circular doubly linked lists.

```
lemma cdll-append-rule:
  "<cdll R xs p ⋆ cdll R ys q ⋆ $13> cdll-append p q <cdll R (xs@ys)>t"
```

Using these two operations above, one can define cdll-snoc which inserts a single element at the end of a list:

```
definition cdll-snoc :: "'a::heap cdll ⇒ 'a ⇒ 'a cdll Heap" where
"cdll-snoc p x' = do {
  q ← cdll-singleton  x';
  cdll-append p q
}"
```

The rule for this operations is essentially the accumulation of the rules for cdll-singleton and cdll-append from above. For example, runtime costs are simply the sum of the ones of these both operations:

```
lemma cdll-snoc-rule:
  "<R x x' ⋆ cdll R xs p ⋆ $16> cdll-snoc p x' <cdll R (xs@[x])>t"
```

3.6 Proof Tactics for Separation Logic

Imperative HOL with time is equipped with some specialized proof tactics that will be presented in this short section. There are four tactics that are designed for reasoning about separation logic and Hoare triples: vcg, solve-entails, timeframeinf and sep-auto.

The tactic timeframeinf is performing frame inference. This is the ascertainment of exactly those assertions of a given set of assertions PS that are not affected by an operation which requires the precondition PRE. In general, a frame inference proof goal has the following form: $\mathsf{PS} \Longrightarrow_{\mathsf{A}} \mathsf{?F} \star \mathsf{PRE}$ where ?F is the to-be-determined frame. For example, to use cdll-singleton from above, the following precondition R x x' ⋆ $3 has to be fulfilled. Consider the assertion R y y' ⋆ R x x' ⋆ $4. The goal of the tactic timeframeinf is to determine that R y y' ⋆ $1 are not used and thus forming the frame. The tactic timeframeinf is designed to syntactically infer the frame by matching individual assertion in PRE with assertions in PS and thereby resolving them.

The verification condition generator tactic vcg applies previously registered rules to a proof goal that has the form of a Hoare triple. There are two types of rules: ones that decompose the program, for example the bind-rule, and others that describe pre- and post-condition of an operation like cdll-snoc-rule. The generator is working in a forward style by alternately applying the first matching decomposition rule and then trying to resolve the arisen proof goals by using a command specific rule. For this latter step, the verification condition generator is invoking timeframeinf. If no rule matches or the frame inference fails, the vcg stops and leaves the residual proof goals unsolved.

In order to reason about any kind of entailment of assertions, the tactic solve-entails can be used. It applies certain simplifications and logic rules to prove the entailment.

All these tactics are combined in sep-auto which chooses one of those by the form of the proof goal. Additionally, sep-auto applies traditionally logic and rewriting steps to the proof goal. Thus, sep-auto is the most powerful tactic but also the most unpredictable one.

Completing this presentation, the tool auto2 shall be mentioned. Imperative HOL with time is designed to work well with this potent proof tactic. However, in this thesis project, it is exclusively used to prove the asymptotic behaviour of timing functions, i. e. functions of type nat ⇒ nat.

4 Runtime Analysis

The presentation in this section is based on [3]. It briefly introduces import concepts of runtime analysis.

Runtime analysis is the estimation of the total time taken to execute a program relative to the input size. To abstract from fluctuations of computation time on real computers, e. g. different execution time for operations that access the memory linearly respectively randomly, usually a model of computation is defined that roughly reflects real computation time. In the model of Imperative HOL with time, one hypothetical time-credit is consumed for any operation that manipulates the heap (operations that affect the whole array of length n cost n + 1 time-credits). Moreover, a function call also takes one time-step, which is modelled by requiring one time-credit for the execution of the return command.

4.1 Asymptotic Runtime

For the comparison of the efficiency of algorithms, an even more coarse-grained analysis is commonly performed called asymptotic runtime analysis. The purpose of this kind of analysis is to determine which algorithm takes the fewest time-steps for large inputs ignoring constant factors. This is reasonable because the larger the input is the greater the absolute execution time will be. Frequently, simple algorithms are more efficient for small inputs, but later, with larger inputs, they are outperformed by more complex ones, which have a larger overhead that has long-term benefits. For this reason, the well-known Landau symbols are used. They define sets of functions that have the same order of growth up to a constant. In Isabelle/HOL, the the term big-O (commonly denoted by $\mathcal{O}$) is defined, s. t. the absolute value growth is considered:

lemma "$\mathsf{f} \in \mathcal{O}(\mathsf{g}) \longleftrightarrow (\exists\, \mathsf{c} > 0 :: \mathsf{real}\ .\ \exists\, \mathsf{n_0} :: \mathsf{nat}.\ \forall\, \mathsf{n} > \mathsf{n_0}.\ |\mathsf{f\ n}| \leq \mathsf{c} \cdot |\mathsf{g\ n}|)$"

These sets of functions can be ordered by the (proper) subset relation and thereby forming an hierarchy. For Landau's big-O, this means that a function f grows faster than g if $\mathcal{O}(\mathsf{f}) \subset \mathcal{O}(\mathsf{g})$.

Based on a given computational model, a runtime function T is derived that relates the size of an input and the runtime in time-steps of the program that is executed with this input. For example, the operation dll-fold takes at least two time-credits per list element plus one time-credit for the beginning as shown previously. Hence, the runtime function for this operation is $\mathsf{T}_{\mathsf{dll\text{-}fold}}(\mathsf{n}) = 2 \cdot \mathsf{n} + 1$.

Thus, runtime comparison of algorithms is simply the comparison of the asymptotics of their runtime functions.

4.2 Amortized Runtime

Not for every operation, the runtime exclusively depends on the input size. For this reason, one differentiates between worst-case, average-case and amortized runtime. As the name suggests, worst-case runtime is the longest runtime that could possibly occur for a given input. In contrast to this, average-case runtime considers all possible inputs of a given size and averages the runtimes.

An advanced method is the amortized runtime analysis. It can be used for operations that manipulate a data structure. The key observation is that certain worst-case behaviour cannot occur very frequently. There are three techniques that are used for an amortized runtime analysis: the Banker's method, the potential function method and the aggregate method.

The core of the bankers method is that an operation can be executed multiple times in a row with low runtime costs and is then followed by one expensive execution. The underlying idea is that the former pay extra time-credits on an account to finance the succeeding expensive operation. These operations are not necessarily the same. For example, insertion into a data structure can be cheap but removing elements expensive. For this reason, the banker's method is hard to formalize since one has to consider arbitrary sequences of operations for an analysis.

A potential function assigns to the data structure a number of time-credits based on its form. Each operation that changes the form can either decrease or increase this potential. Cheap operations usually increase the potential and thus must pay extra to charge it while expensive ones decrease it and make use of these saved credits. This corresponds closely to the banker's method. However, this approach is significantly easier to formalize. Hence, it will be used for the verification of Fibonacci Heaps.

The aggregation method considers the total runtime of multiple operations in a larger context as for example the execution of an complex algorithm. Such an algorithm may invoke certain operations multiple times, s. t. the possible worst-case for a single invocation is irrelevant as long as the overall runtime is still low. In these contexts, the potential of the used data structure is usually oscillating and sometimes completely discharged at the end like in the extensive example in section 8.

5 Priority Queues

A priority queue is an abstract datatype that supports operations similar to a basic first-in-first-out queue. However, the `pop` operation does not return the oldest element but the one with the highest associated priority. Highest is defined by a given linear order, e. g. $(\geq)$ on natural numbers where the highest priority would be consequentially 0. If two elements have both the highest priority, any of those may be returned when `pop` is called. Priority queues play an important role in algorithms like Dijkstra's algorithm [25], Prim's algorithm [26] and management of resources like bandwidth or computation time [2].

5.1 Specification

Priority queues can be specified in two fashions: with and without separately defined priority keys. Many standard libraries of common general-purpose programming languages like C++ [27], Java [28] or Python [29] only provide priority queues without separate priority keys. Therefore, the inserted values represent their respective priority themselves. The same applies to the data structure theory of Isabelle/HOL where one can find the a specification at HOL-Data_Structures.Priority_Queue_Specs, which describes so called min priority queues. A min priority queues is a queues where pop returns the element with the lowest priority instead of the highest.

This specification consists of two parts: a description of all operations that are required to implement, and a set of properties these operations have to fulfil. The latter ensures the well-behaving of the implemented priority queue.

The operations can be categorized into two groups: operations that read or manipulate the heap, and functions that are only necessary to describe heaps formally:

```
locale Priority-Queue-Merge =
  fixes empty     :: "'q"
  and   is-empty  :: "'q ⇒ bool"
  and   insert    :: "'a::linorder ⇒ 'q ⇒ 'q"
  and   get-min   :: "'q ⇒ 'a"
  and   merge     :: "'q ⇒ 'q ⇒ 'q"
  and   del-min   :: "'q ⇒ 'q"
  and   invar     :: "'q ⇒ bool"
  and   mset      :: "'q ⇒ 'a multiset"
```

All these operations are declared with a certain behaviour in mind which shall be described first informally in the next paragraph and then formally by the properties thereafter.

The operation empty creates as the name suggests an empty priority queue without any elements and is-empty checks if an arbitrary queue is such an empty one. The operation insert adds exactly one element into the priority queue, whereas merge combines two complete queues into a new one. In order to retrieve the minimal element without removing it, the operation get-min is used. The operation del-min removes this very element and possibly restructures the priority queue.

The invariant is formalized by the proposition invar. It describes all valid forms of the priority queue. In this specification, priority queues are modelling multisets, hence, mset is used to relate a concrete priority queue to a multiset. These two functions are essential to describe a well-behaving priority queue implementation.

In Isabelle/HOL, the notation {# a, ..., z #} is used for multisets and x ∈# M for multiset membership. In particular, {#} denotes the empty multiset. Multisets form an Abelian monoid, and since this is a type-class in Isabelle/HOL, one can use the generic (+) sign to denote multiset unification.

The properties part of specification can also be sorted into two categories: Properties that specify functional correctness, and properties that demand that each operation respects the invariant. All of them will be proven for Fibonacci heaps in later sections.

The following statements regard the multiset of elements that are stored in the priority queue. For example mset-insert states that the abstract meaning of insert x q is adding an element x to the multiset of the priority queue mset q. For this to hold, one can assume the invariant for queue q. Most interestingly is perhaps the property mset-get-min: The operation get-min q has to return exactly the element which is a minimal one in mset q. Since multisets are used in this formalization, the minimal element is not necessarily unique.

```
assumes mset-empty:    "mset empty = {#}"
and     is-empty:      "invar q ⟹ is-empty q = (mset q = {#})"
and     mset-insert:   "invar q ⟹ mset (insert x q) = mset q + {# x #}"
and     mset-del-min:  "invar q ⟹ mset q ≠ {#} ⟹
                        mset (del-min q) = mset q - {# get-min q #}"
and     mset-get-min:  "invar q ⟹ mset q ≠ {#} ⟹
                        get-min q = Min-mset (mset q)"
and     mset-merge:    "invar q1 ⟹ invar q2 ⟹
                        mset (merge q1 q2) = mset q1 + mset q2"
```

The remaining properties are simply stating: If the invariant holds before, it must still hold after the application of an operation:

```
assume  invar-empty:   "invar empty"
and     invar-insert:  "invar q ⟹ invar (insert x q)"
and     invar-del-min: "invar q ⟹ mset q ≠ {#} ⟹ invar (del-min q)"
and     invar-merge:   "invar q1 ⟹ invar q2 ⟹ invar (merge q1 q2)"
```

5.2 Overview of Different Implementations

A prevalent implementation of an abstract priority queue is a so-called heap. In table 1, selected heaps are listed in the order of their first description.

The table shows the according runtime for the most relevant operations for priority queues. As one can see, Fibonacci heaps outpeform the prior binary and binomial heaps using an amortized analysis. Brodal heaps have been proposed significantly later.

However, they have an optimal worst-case runtime behavior [30]. Nevertheless, in an aggregated analysis, as it commonly performed for convoluted algorithms like A* search, the overall runtime is not improving when Brodal heaps are used instead of Fibonacci heaps. For this reason, Fibonacci heaps are still more prominent and influential than Brodal heaps.

	Binary	Binomial	Fibonacci	Brodal
find-min	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
delete-min	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$ †	$\mathcal{O}(\log n)$
insert	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
merge	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
decrease-key	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$ †	$\mathcal{O}(1)$

Table 1: Selected Heaps and According Runtimes Listed by Operation
† Amortized Runtime

6 Introduction to Fibonacci Heaps

A heap is a tree or forest [31] where the priority of the parent node is higher than the ones of its children. This is called the heap property [3]. If the order of the priorities is reversed, one calls the heap a min-heap, as mentioned before. In order to avoid confusion, the term heap in this and the following sections refers to this kind of data structure, not to the heap memory.

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

A tree with multiple children is called a rosetree [32]. In the datatype above, some additional information can be attached to each node like a boolean mark and its rank. Moreover, the value type is constrained to be a linear order.

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

Heaps that are forests of such rose trees can be represented by the datatype roseheap printed above. It annotates the heap with the number of its elements.

Regardless of the invariant that specifies the form of such a rose tree, one can already define the min-heap property recursively as follows:

```
fun min-tree :: "_ ⇒ bool" where
  "min-tree (Node _ _ v ts)   ⟷   (∀t∈set ts. min-tree t ∧ v ≤ val t)"
```

If the heap is a forest of rose trees, all of them have to fulfil the min-heap property:

```
definition min-heap-list :: "_ ⇒ bool" where
  "min-heap-list ts  ⟷  (∀t∈set ts. min-tree t)"

fun min-heap :: "_ ⇒ bool" where
  "min-heap (Heap _ ts)  ⟷  min-heap-list ts"
```

Furthermore, one can already define functions that abstract rose trees and heaps to multisets, i. e. functions that collect all values stored in this concrete data structures.

```
fun mset-tree :: "_ rosetree ⇒ _ multiset" where
  "mset-tree (Node _ _ a c) = {# a #} + (∑t ∈# mset c. mset-tree t)"

definition mset-heap-list :: "_ rosetree list ⇒ _ multiset" where
  "mset-heap-list ts = (∑t ∈# mset ts. mset-tree t)"
```

The function mset-heap is part of the priority queue specification described in the prior section. It returns the multiset of a given rose heap of any particular form.

```
fun mset-heap :: "_ roseheap ⇒ _ multiset" where
  "mset-heap (Heap _ c) = mset-heap-list c"
```

Analogously, one can define some auxiliary functions that count the number of elements in a tree or heap, respectively. These functions are practical for reasoning about the rank of trees in particular given a sufficient tree invariant.

```
fun rosetree-size :: "_ rosetree ⇒ nat" where
  "rosetree-size (Node _ _ _ c) = 1 + (∑t' ← c. rosetree-size t')"

definition heap-list-size :: "_ rosetree list ⇒ nat" where
  "heap-list-size ts = (∑t ← ts. rosetree-size t)"
```

6.1 Derivation from Binomial Heaps

Binomial heaps were invented by Vuillemin [33] and are commonly taught in an introductory data structure course. Moreover, binomial heaps lay the foundation for Fibonacci heaps, and for this reason, they will be presented in this section. Also, it will be shown how to derive Fibonacci heaps from them. A formal verification of functional binomial heaps by Nipkow and Lammich can be found in the data structure theory of Isabelle/HOL in file HOL-Data_Structures.Binomial_Heap which is inspired by the work of Okasaki [34]. The presentation here is based on that theory. Nevertheless, some definitions are slightly adapted to fit the rosetree datatype.

6.1.1 Invariants

A binomial heap is a forest of rose trees which fulfils the binomial property, i. e. the i-th child of such a tree has rank i and fulfils the binomial property itself. This implies that the children of a binomial tree are listed in a dense and strictly ascending order of rank:

```
fun invar-btree :: "_ rosetree ⇒ bool" where
  "invar-btree (Node r _ x ts)  ⟷  map rank ts = [0 ..< r] ∧
                                    (∀t ∈ set ts. invar-btree t) "
```

From this definition, one can conclude that a binomial tree of rank 0 has no children. Moreover, a binomial tree of rank $r + 1$ can be formed by linking two trees of rank r together, i. e. to append one tree to the list of children of the other one. By using these two facts, one shows by induction that a binomial tree of rank r has exactly 2^r elements:

```
lemma size-mset-btree: "invar-btree t ⟹ (mset-tree t) = 2 ^ rank t"
```

A binomial heap is a list of binomial trees also in strictly ascending order of rank. However, the order of this root list has not to be dense.

```
definition invar-bheap :: "_ roseheap ⇒ bool" where
"invar-bheap ts  ⟷  (sorted-wrt (<) (map rank ts)) ∧
                     (∀t ∈ set ts. invar-btree t)"
```

Using lemma size-mset-btree and this definition, one can see that the length of the root list is logarithmic in the number of elements.

The core idea of Fibonacci heaps is to execute most of the binomial heap operations in a lazy fashion, i. e. postponing the restoration of the invariant as long as possible. For that purpose, Fibonacci heaps have a significantly weakened invariant compared to binomial heaps. Fibonacci heaps allow multiple Fibonacci trees (defined later) of the same rank in the root list as opposed to maximal one. Moreover, the root list need not be in order by rank. However, a new condition is replacing the order property as one can see by its definition fibheap below: the first tree of the list must be the one with the smallest priority.

```
fun fibheap :: "_ ⇒ bool" where
  "fibheap (Heap n [])  ⟷  n = 0" |
  "fibheap (Heap n ts)  ⟷  (∀ t ∈ set ts. val (hd ts) ≤ val t ∧ fibtree t) ∧
                            n = size (mset-heap-list ts)"
```

Furthermore, the annotated number n has indeed to be the correct number of elements. This annotation was omitted in the definition of binomial heaps above.

Naturally, a Fibonacci heap has also to fulfil the heap property so that the complete invariant is the following:

definition "invar h $\longleftrightarrow$ fibheap h $\wedge$ min-heap h"

Like the heaps, Fibonacci trees are a degenerated form of binomial trees. The motivation behind this invariant relaxation is to lazily execute decrease-key as explained later. For Fibonacci trees, the order by rank is not necessarily dense anymore. Moreover, a child is allowed to have a rank one less than its position in the list if it is marked. Nevertheless, a tree of rank n still has n children:

lemma fibtree-simp: "fibtree (Node r m x ts) =
((∀t∈set ts. fibtree t) ∧ (r = length ts) ∧
(∀i∈{0 ..< length ts}. let t = ts ! i in i ≤ rank t + of-bool (marked t)))"

One can show, that binomial trees are indeed a stricter form of Fibonacci heaps, i. e. each binomial tree is also a Fibonacci tree:

lemma btree-is-special-case-of-fibtree: "invar-btree t $\Longrightarrow$ fibtree t"

A similar statements does not hold for binomial heaps since they do not fulfil the newly added property that requires the head of the root list to contain the minimal element.

6.1.2 Operations

The four most important operations on binomial heaps are merge, insert, pop-min, decrease-key, which will be briefly presented in the following. Also, for each operation, its Fibonacci heap counterpart is derived.

Insert The insertion of single elements into a binomial heap is based on an insert operation for whole trees. With such, element insertion is simply the tree insertion of a singleton tree, i. e. a tree of rank 0 and one element.

This tree insertion iterates over the root list inserting the tree at the position that is according to its rank. If the heap contains already another tree of the same rank, both are linked together and the resulting tree of incremented rank is inserted recursively by further iterating through the root list.

This operation takes in worst case $\mathcal{O}(\log n)$ time-steps since the length of the list of trees is bound logarithmically by the number of elements n in the heap.

Fibonacci heaps instead delegate insertion to the merge operation. Therefore, in order to perform insertion, the heap is merged with a newly created singleton one containing the element that is to be inserted.

Merge Two binomial heaps can be merged by zipping the two lists of trees together, s. t. the resulting list of trees is again in strict order by rank. This operation is similar to the merging process in the merge sort algorithm. However, if two trees have the same rank, they are linked and the resulting tree is inserted – as described in the previous paragraph – after the rest of the two heaps was merged. Even though it is not so easy to determine, this operation takes in worst case $\mathcal{O}(\log n)$ time-steps where again n is the number of elements in the heap. In order to see, that the runtime is at least logarithmic, one has to notice that the length of the root list, which has to be completely iterated over, is logarithmically bound by n.

Both in merge as in insert, much work has to be spent to restore the binomial heap invariant, whereas Fibonacci heaps postpone this work. In their case, the two lists of trees can be directly concatenated because their weakened invariant does not demand that there is at most one tree of each rank as explained before. If list concatenation takes constant time as it is the case for doubly linked lists, merging Fibonacci heaps takes in worst case constant time. Thus, the same applies to insertion since it is based on merge.

Decrease-Key The operation decrease-key is only supported for imperative heap implementations that use separate keys to establish an order on the elements. If one has stored a reference to an element of the heap, one can decrease its key using this operation. However, decreasing the key of an element leads to violation of the min-heap invariant if the decreased key is smaller than the one of its parent node. For this reason, some restructuring needs to be done to restore the invariant. In the case of binomial trees, the child is swapped with its parent and continuing recursively upwards in the tree which is commonly called sift-up operation. This takes $\mathcal{O}(\log n)$ time-steps because the function has to ascend the complete tree in worst case. As before, n is the number of elements in the heap.

As opposed to this, Fibonacci heaps are allowed to degenerate slightly by losing one child. So, at least in case of a proper node, i. e. its rank equals the number of children, the operation decrease-key just cuts off that child with the to-be-decreased key from its parent node and inserts it into the heap list. This takes only constant time. A detailed explanation and analysis about the other case can be found in the section about decrease-key.

Pop-Min For binomial heaps, pop-min is split into two phases: First, finding the minimal root which requires to iterate over the root list. Second, merging the children of the popped tree into the heap using merge which can be used since the children of a binomial tree are again a binomial heap. The first phase takes logarithmic time as well as the second phase, thus total runtime of this lays in $\mathcal{O}(\log n)$ where n is the number of elements contained in the heap.

For Fibonacci heaps, all postponed work has to be done in this operation. It restructures the degenerated heap in a similar manner to the merge operation of binomial heaps. Nevertheless, the operation pop-min can even take in worst case $\mathcal{O}(n)$ time-steps. For example, this is the case when all trees in the heap are singletons and therefore the length of the root list is exactly n. However, the amortized runtime is $\mathcal{O}(\log n)$ as shown later.

The reason to postpone most of the work to pop-min is that this operation is usually called less frequently then decrease-key. For example, Dijkstra's algorithm calls decrease-key n^2 times whereas pop-min only n times [3] where n is the number of nodes in a graph.

6.2 Imperative Implementation

In this part, an imperative implementation of Fibonacci heaps is formulated and related to its functional counterpart.

Since Fibonacci trees are a special case of rose trees, we can first define an imperative implementation of the latter and directly reuse this for Fibonacci trees.

datatype 'a rosetree-imp = Rose (cval: 'a) (sub: "'a rosetree-imp cdll")

Imperative rose trees can be implemented as a datatype that contains a field accessible by the selector **cval** for its value and a circular doubly linked list to store the children.

Nodes of a Fibonacci tree are annotated with a rank and a mark. These are stored in a triple together with the node value:

type-synonym 'a fibtree-imp = "(nat × bool × 'a) rosetree-imp"

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

Concluding the definition, the assertion fibtree-imp relates the functional and imperative implementation in the canonical way. For this, fibtree-imp calls the higher-order assertion cdll, defined in a prior section, that refines each element in the functional list ts recursively by fibtree-imp itself.

To support the verification generator, an assertion fibtree-imp' based on fibtree-imp is defined. However, they are equivalently expressive:

lemma fibtree-imp'-eq-fibtree-imp: "fibtree-imp' t ti = fibtree-imp t ti"

Imperative Fibonacci heaps are defined as a pair of the number of elements and the root list which is a circular and doubly linked list:

type-synonym 'a fibtree-list-imp = "'a fibtree-imp cdll"

type-synonym 'a fibheap-imp = "nat $\times$ 'a fibtree-list-imp"

Again, the functional and imperative data structure are related canonically by the following assertion:

definition "fibheap-imp h hi = (case (h, hi) of ((Heap n ts), (n', p)) $\Rightarrow$
cdll fibtree-imp' ts p * $\uparrow$(n' = n))"

6.3 Potential Function

As mentioned in a prior section, to formally verify amortized runtimes, the usage of a potential function is a common approach. It is also used in the original description of Fibonacci heaps by Tarjan [2].

The potential of a Fibonacci heap reflects the degree of degeneration compared to binomial heaps, i. e. the number of trees in the root list with the same rank and the number of nodes in all trees that have a rank one too less. The former is slightly over-approximated by simply counting all trees of the root list.

fun marked-num :: "_ rosetree $\Rightarrow$ nat" **where**
"marked-num (Node _ m _ c) = of-bool m + ($\sum$t $\leftarrow$ c. marked-num t)"

By the Fibonacci tree invariant, a node has decremented rank if and only if it is marked, thus above function counts exactly these nodes in a Fibonacci tree.

Using this, one defines the potential φ as follows:

fun φ **where** " (Heap _ ts) = length ts + ($\sum$t $\leftarrow$ ts. marked-num t)"

If one wants to use the result of the amortized runtime analysis, a combined predicate is useful that describes both the relation of imperative and functional implementation and the potential:

definition "fibheap_i h hi $\equiv$ fibheap-imp h hi $\star$ \$(φ h $\cdot$ 37)"

The seemingly mysterious number 37 is the result of the runtime analysis of consolidate-rec-imp (see section 7.6.4).

How this assertion can be used in a runtime analysis is demonstrated section 8 about heap sort.

6.4 Fibonacci Numbers

Fibonacci numbers are an important part of the runtime analysis of Fibonacci heaps as elucidated in the next section. Therefore, they are eponymous for this data structure. For this reason, we will describe the relevant properties of this sequence of numbers, which was already described by ancient Indian mathematicians and then popularized together with the Hindu–Arabic number system by Leonard Bonacci (later named Fibonacci) in western Europe [35].

The ith Fibonacci numbers is denoted by F_i. Nowadays, the sequence is usually defined as following with with a leading zero:

$$F_i = \begin{cases} 0 & \text{if } i = 0 \\ 1 & \text{if } i = 1 \\ F_{i-1} + F_{i-2} & \text{otherwise} \end{cases}$$

In the number theory library of Isabelle/HOL, one can already find some properties about Fibonacci numbers that describe its close relation to the golden ratio. However, we had to add some basic facts like monotonicity, i. e. if $n \leq m$ then $F_n \leq F_m$, formally stated in Isabelle:

lemma fib-mono: "n $\geq$ m $\Longrightarrow$ fib n $\geq$ fib m"

The conditional strict monotonicity is also proven easily by induction:

lemma fib-strict-mono: "1 < n $\Longrightarrow$ n < m $\Longrightarrow$ fib n < fib m"

It follows as contrapositive from the lemma fib-mono:

lemma fib-less: "fib m < fib n $\Longrightarrow$ m < n"

The following lemma fib-sum is of great importance for the runtime analysis of Fibonacci heaps. It states that $F_{n+2} - 1$ is the sum of its predecessors in the sequence up to the nth element. Statement and proof are taken from [3]. As one can see, it can be proven fully automatically by induction:

lemma fib-sum: "fib (n + 2) = 1 + ($\sum_{i=0}^{n}$ fib i)"

Also by induction, one can show that the sequence grows exponentially at least by the factor the golden ratio φ. This can easily be restated using the logarithm:

lemma fib-lower-bound:
  fixes φ :: real
  defines "$\varphi \equiv$ (1 + sqrt 5) / 2"
  shows "φ ^ n $\leq$ fib (n + 2)"

corollary fib-lower-bound-log:
  fixes φ :: real
  defines "$\varphi \equiv$ (1 + sqrt 5) / 2"
  shows "n $\leq$ log φ (fib (n + 2))"

6.5 Maximally Degenerated Fibonacci Trees

That the size of binomial trees grows exponentially in its rank could be seen quite easily by their invariant. In contrast to this, Fibonacci trees have a relaxed invariant which complicates the analysis.

To see that a Fibonacci tree of rank n has at least size fib (rank n + 2), an informal proof by strong induction over the rank shall be sketched. Consider $n = 0$: A Fibonacci tree of rank 0 has no children and thus, it is of size 1. Moreover, $1 =$ fib $(0 + 2)$ concluding this case. In the induction step, a tree t of rank $n + 1$ is considered. Hence, it has $n + 1$ children. By the Fibonacci tree invariant, the i-th child has at least rank $i \dot{-} 1$. Hence, its size is at least fib $(i + 1)$. Shifting the sum and then using lemma fib-sum, one can finally conclude that rosetree-size t $\geq$ fib $((n + 1) + 2)$ and thus rosetree-size t $\geq$ fib (rank t + 2).

theorem fibtree-bounded-tree-size: "fibtree t $\Longrightarrow$ rosetree-size t $\geq$ fib (rank t + 2)"

The formal proof of this theorem uses induction over the form of t, which gives an easier applicable induction hypothesis compared to strong induction over n.

From this theorem, one can see using lemma fib-lower-bound-log that the rank of a Fibonacci tree is logarithmically bound by its size:

corollary fibtree-bounded-tree-size-log:
  fixes φ :: real
  defines "$\varphi \equiv (1 + \text{sqrt } 5) / 2$"
  assumes invar: "fibtree t"
  shows "rank t $\leq$ log φ (rosetree-size t)"

lemma fibtree-bounded-rank-aux:
  assumes "fibtree t"
  assumes "rosetree-size t $\leq$ fib (n - 1)"
  and "$2 < n$"
  shows "rank t + 1 $<$ n"

However, the proofs, especially the ones for consolidate, in the sections hereafter make use of the auxiliary version fibtree-bounded-rank-aux of the previous lemma to avoid the notion of logarithm so that all numbers involved are natural ones.

7 Verification of Fibonacci Heaps

The used approach to verify imperative Fibonacci heaps is as follows: First, a functional version of them is defined and proven correct. Then, an imperative implementation is derived which is shown to be a refinement of the functional one. Lastly, Hoare logic with time is used to verify the amortized runtimes of the imperative operations.

The definitions, the formal correctness proofs and the amortized runtime will be presented in detail for each operation following the scheme of the above approach. First, operations will be explained that require a constant amount of time-steps. Thereafter, the more complex, recursive operations are described.

7.1 Empty and IsEmpty

The most basic operations provided for Fibonacci heaps are empty and is-empty and their imperative equivalents.

Fibonacci heaps are empty when their number of elements is zero.

```
lemma empty-mset: "mset-heap empty = {#}"
```

This is only the case if the root list contains no trees, hence:

```
definition empty where "empty = Heap 0 []"
```

It is trivial to show that the above definition fulfils the invariant, i. e. the empty heap is a min-heap as well as a Fibonacci heap:

```
lemma invar-empty: "invar empty"
```

Also trivial is the fact that the empty heap has no inherent potential:

```
lemma empty-pot: "φ empty = 0"
```

The imperative implementation is straightforward. An empty imperative Fibonacci heaps is a pair of 0 and a representative of an empty circular doubly linked list:

```
definition "empty-imp = cdll-empty ≫= λp. return (0, p)"
```

To imperatively create an empty Fibonacci heap, we need two time-credits. This applies also to the amortized analysis since, as aforementioned, the potential is zero:

```
lemma empty-imp-rule:
  "<$2>
     empty-imp
   <fibheap-imp empty>"
```

```
lemma empty-imp-rule-amo:
  "<$2>
     empty-imp
   <fibheap_i empty>"
```

To determine if a Fibonacci heap is empty, one can simply check if the annotated number of elements is zero:

```
definition is-empty :: "_ roseheap ⇒ bool" where
  "is-empty h = (case h of (Heap 0 _) ⇒ True | _ ⇒ False)"
```

Again trivially to prove is functional correctness:

```
lemma is-empty-mset-heap:
  assumes "fibheap h"
  shows "is-empty h ⟷ mset-heap h = {#}"
```

The imperative version is as simple as the functional one:

```
definition is-empty-imp :: "_ fibheap-imp ⇒ bool Heap" where
  "is-empty-imp hi = return (case hi of (0 :: nat, _) ⇒ True | _ ⇒ False)"
```

Computation of this plain operation costs only a single time-credit.

```
lemma is-empty-imp-rule:
  "<fibheap-imp h hi ⋆ $1>
     is-empty-imp hi
   <λb. fibheap-imp h hi ⋆ ↑(b = is-empty h)>"
```

The same applies for the amortized analysis since the structure of the heap is not changed:

```
lemma is-empty-imp-rule-amo:
  "<fibheap_i h hi ⋆ $1>
     is-empty-imp hi
   <λb. fibheap_i h hi ⋆ ↑(b = is-empty h)>"
```

7.2 Get-Min

The operation get-min does not change the heap. Just the minimum element is retrieved, which is in the root of the first tree in the list:

```
fun get-min :: "('a :: linorder) roseheap ⇒ 'a"  where
  "get-min (Heap _ ts) = val (hd ts)"
```

Regarding functional correctness: If the Fibonacci heap is not empty, get-min indeed returns its minimal element.

```
lemma get-min-correct:
  assumes "invar h" and "mset-heap h ≠ {#}"
  shows "get-min h = Min-mset (mset-heap h)"
```

As before, the imperative version is easily derived. The used operation front-value returns the value of the list head:

```
definition "get-min-imp h = (case h of (n, Some p) ⇒ front-value p)"

lemma front-value-rule:
  "<cdll fibtree-imp (t#ts) (Some p) ⋆ $2>
   front-value p
   <λy. cdll fibtree-imp (t#ts) (Some p) ⋆ ↑(y = rosetree.val t)>"
```

Inspecting the front element of the list takes two time-steps. No structures are changed. Hence, the amortized runtime for get-min is the same as the worst-case one.

```
lemma get-min-imp-rule:
  "<fibheap-imp (Heap c (t#ts)) hi ⋆ $2>
    get-min-imp hi
   <λy. fibheap-imp (Heap c (t#ts)) hi ⋆ ↑(y = get-min (Heap c (t#ts)))>"

lemma get-min-imp-rule-amo:
  "<fibheap_i (Heap c (t#ts)) hi ⋆ $2>
     get-min-imp hi
   <λy. fibheap_i (Heap c (t#ts)) hi ⋆ ↑(y = get-min (Heap c (t#ts)))>"
```

As one can see, these rules can just be applied to non-empty heaps, i. e. heaps that contain at least one tree.

7.3 Merge

More interesting than the previous operations is merge which is sometimes also referred by meld. However, this operation is also non-recursive. Two Fibonacci heaps are merged by concatenation of the root lists. There is only one aspect of the invariant that might be broken which is that the head of the root list must contain the minimal value of the heap. Nevertheless, one can avoid this violation by inspecting values of the heads of both lists first and concatenate them appropriately.

```
fun merge :: "_ roseheap ⇒ _ roseheap ⇒ _ roseheap" where
  "merge h1 (Heap _ []) = h1" |
  "merge (Heap _ []) h2 = h2" |
  "merge (Heap c1 (t1#ts1)) (Heap c2 (t2#ts2)) =
      Heap (c1+c2) ((if val t1 ≤ val t2 then (t1#ts1)@(t2#ts2)
                                        else  (t2#ts2)@(t1#ts1)))"
```

It can fully automatically proven that two merged Fibonacci heaps still comply with the invariant:

```
corollary invar-merge[intro!]: "invar h1 ⟹ invar h2 ⟹ invar (merge h1 h2)"
```

Furthermore, the potential of the merged heap is plainly the sum of both input heaps:

```
lemma merge-pot: "φ (merge h1 h2) = φ h1 + φ h2"
```

Also, the merge operation does not discard any elements, which completes their functional correctness:

```
lemma merge-mset: "mset-heap (merge h1 h2) = mset-heap h1 + mset-heap h2"
```

The imperative merge-imp follows the form of its functional counterpart closely: If one of the input heaps is empty, the respective other is returned. Otherwise, after retrieving the values of the heads of both root lists, they are concatenated in correct order.

```
fun merge-imp :: "_ fibheap-imp ⇒ _ fibheap-imp ⇒ _ fibheap-imp Heap" where
  "merge-imp h1 (_, None) = return h1" |
  "merge-imp (_, None) h2 = return h2" |
  "merge-imp (c1, Some p) (c2, Some q) = do {
     l1 ← front-value p;
     l2 ← front-value q;
     r ← (if l1 ≤ l2
           then cdll-append (Some p) (Some q)
           else cdll-append (Some q) (Some p));
     return (c1+c2, r)
  }"
```

The imperative merge operation is faster than the functional one. The reason for this is that cdll-append takes only a constant amount of time-steps. Opposed to this, the runtime of the functional implementation is linear in the number of elements of the first input list.

By case distinction, one can show that merge-imp is in fact a correct imperative refinement of functional merge:

```
lemma merge-imp-rule:
  "<fibheap-imp h1 h1i ⋆ fibheap-imp h2 h2i ⋆ $(append-time + 5)>
     merge-imp h1i h2i
   <fibheap-imp (merge h1 h2)>t"
```

Since concatenation of circular doubly linked lists takes already 13 time-credits, merge-imp takes 18 time-steps in total. The amortized analysis gives the same costs as the worst-case analysis because the potential of the merged heap is the sum of the input heaps.

```
lemma merge-imp-rule-amo:
  "<fibheap_i h1 h1i ⋆ fibheap_i h2 h2i ⋆ $(append-time + 5)>
     merge-imp h1i h2i
   <fibheap_i (merge h1 h2)>_t"
```

7.4 Singleton

As mentioned in section 6.3 Derivation from Binomial Heaps, insertion is based on merge using singleton heaps. For this reason, the creation of a singleton heap will be presented here.

Such a singleton Fibonacci heap consists of one tree of rank 0 that itself contains exactly a single element. Hence, one defines the following constructor function for singletons.

```
definition "singleton v = Heap 1 [Node 0 False v []]"
```

```
lemma mset-singleton: "mset-heap (singleton x) = {# x #}"
```

Directly from definition, we can derive that the potential of a singleton is one:

```
lemma singleton-pot: "φ (singleton v) = 1"
```

Moreover, a singleton heap obviously fulfils the invariant:

```
lemma invar-singleton: "invar (singleton v)"
```

The constructor singleton-imp of an imperative singleton heap makes use of the operation cdll-singleton' which constructs a singleton circular doubly linked list. The latter takes the auxiliary annotations fibtree-imp and (Node 0 False v []) to support the verification condition generator. The actual data stored in the list is (Rose (0, False, v) None).

```
definition singleton-imp :: "'a :: {linorder, heap} ⇒ 'a fibheap-imp Heap" where
  "singleton-imp v = do {
    q ← cdll-singleton' fibtree-imp (Node 0 False v []) (Rose (0, False, v) None);
    return (1, q)
  }"
```

The imperative creation of a singleton heap costs four time-steps and indeed creates a refined functional singleton heap.

```
lemma singleton-imp-rule: "<$4> singleton-imp v <fibheap-imp (singleton v)>"
```

This time, the amortized runtime analysis is more interesting. Since the potential of a singleton is one, 37 extra time-credits for charge has to be paid, thus:

```
lemma singleton-imp-rule-amo: "<$41> singleton-imp v <fibheap_i (singleton v)>"
```

7.5 Insert

Finally, one can define insert based on the previous two operations as described priorly:

```
definition insert :: "'a::linorder ⇒ 'a roseheap ⇒ 'a roseheap" where
  "insert v h = merge (singleton v) h"
```

Preservation of the invariant is again proven fully automatically as well as correct behaviour on the abstracted multiset level:

```
corollary invar-insert[intro!]: "invar h ⟹ invar (insert v h)"
```

```
lemma insert-mset: "mset-heap (insert x q) = mset-heap q + {# x #}"
```

The same is true for the potential, which is increment. This follows immediately from the lemmata about singleton and merge, respectively:

```
corollary insert-pot: "φ (insert v h) = φ h + 1"
```

The imperative refinement makes also use of the priorly defined operations:

```
definition "insert-imp v hi = do {
  x ← singleton-imp v;
  merge-imp x hi
}"
```

As the following rule shows, the imperative version resembles the functional one. It takes 22 time-steps in total.

```
lemma insert-imp-rule:
  "<fibheap-imp h hi ⋆ $22> insert-imp v hi <fibheap-imp (insert v h)>_t"
```

As for singleton-imp, the operation has to pay the increment in potential with 37 additional time-credits, as before:

```
lemma insert-imp-rule-amo:
  "<fibheap_i h hi ⋆ $59> insert-imp v hi <fibheap_i (insert v h)>_t"
```

7.6 Pop-Min

The operation pop-min has two phases: removing the minimal root and then restructuring the heap, s. t. the root list contains no tree of the same rank twice or more. In this section, the operations necessary for restructuring are presented first because they lay the foundation for the amortized runtime analysis of pop-min.

The operation for restructuring is called consolidate. It folds over the root list with a complex accumulator storing trees ordered by rank. While folding, it is checked if there is a tree already included in the accumulator that has the the same rank as current one. If so, they are linked. The resulting tree is of incremented rank and is added to the accumulator. Since there could be again a tree of the same rank – one higher as before – this process is repeated. In this thesis, this recursive procedure is named join. Finally, the accumulator is used to recreate a proper root list.

7.6.1 Link

As for binomial trees, link connects two trees of the same rank by hanging one below the other according to their priority. The tree that is added to the list of children of the other will be untagged for the following reason: A marked node has a decremented rank compared to its position. However, we assume the trees have the same rank and therefore the inserted tree has the appropriate rank for its position.

```
definition link where "link t1 t2 =
   (case t1 of (Node r1 m1 v1 ts1) ⇒ case t2 of (Node r2 m2 v2 ts2)
      ⇒ if v1 ≤ v2 then (Node (Suc r1) m1 v1 (ts1 @ [Node r2 False v2 ts2]))
         else (Node (Suc r2) m2 v2 (ts2 @ [Node r1 False v1 ts1])))"
```

By nested case distinction, one proves that link indeed preserves the Fibonacci tree invariant:

```
lemma link-fibtree:
  assumes "fibtree t1" "fibtree t2" and rank: "rank t2 = rank t1"
  shows "fibtree (link t1 t2)"
```

The following lemmata can even be proven fully automatically:

lemma link-min-tree: "min-tree t1 $\Longrightarrow$ min-tree t2 $\Longrightarrow$ min-tree (link t1 t2)"

lemma link-mset: "mset-tree (link t1 t2) = mset-tree t1 + mset-tree t2"

From these lemmata above, one can see that link also preserves the min-tree property and discards no elements. Furthermore, one shows easily that the new root is in fact the smallest of the input ones:

lemma link-val: "val (link t1 t2) = min (val t1) (val t2)"

As already said, linking two trees increments the rank which is proven automatically:

lemma link-rank: "rank t2 = rank t1 $\Longrightarrow$ rank (link t1 t2) = rank t1 + 1"

Since one tree is potentially unmarked, the number of marks may decrease while linking:

lemma link-marked-num:
shows "marked-num (link t1 t2) ≤ marked-num t1 + marked-num t2"

As the previously examined operations, link-imp follows closely its functional version:

```
definition link-imp :: "_ fibtree-imp ⇒ _ fibtree-imp ⇒ _ Heap" where
  "link-imp t1 t2 = do {
   (case t1 of Rose (r1, m1, v1) ts1 ⇒ case t2 of Rose (r2, m2, v2) ts2
      ⇒ if v1 ≤ v2 then do {
            r ← cdll-snoc ts1 (Rose (r2, False, v2) ts2);
            return (Rose ((Suc r1), m1, v1) r)
          } else do {
            r ← cdll-snoc ts2 (Rose (r1, False, v1) ts1);
            return (Rose ((Suc r2), m2, v2) r)
        }
  )}"
```

However, executing cdll-snoc takes only constant time. In contrast, the functional implementation is linear in the number of list elements as already explained in the section about the merge-imp operation.

Again by case distinction, it is shown that the imperative version link-imp is a refinement of the functional one linking two trees in 17 time-steps:

```
lemma link-imp-rule: "<fibtree-imp t1 t1i ⋆ fibtree-imp t2 t2i ⋆ $17>
                          link-imp t1i t2i
                        <fibtree-imp' (link t1 t2)>_t"
```

7.6.2 Work List

As mentioned, consolidate uses a complex accumulator which will be called work list in the following. This work list stores at position i a tree of rank i. If no such tree was encountered yet, None is used as a place-holder at this position instead:

```
type-synonym 'a worklist = "'a option list"
```

To verify consolidate, one has to formulate several assertions about the work list. For this purpose, a general fold operation is defined that skips empty positions:

```
fun (in ordered-cancel-comm-monoid-diff) work-list-fold where
  "work-list-fold _ [] = 0" |
  "work-list-fold f (Some t # ts) = f t + work-list-fold f ts" |
  "work-list-fold f (None # ts) = work-list-fold f ts"
```

Using this, one defines with ease a function that collects all elements in the work list, one counting the number of marked nodes, and one the number of non-empty positions:

```
definition "mset-work-list = subset-mset.work-list-fold mset-tree"

definition "work-list-marked = work-list-fold marked-num"

definition "work-list-count = work-list-fold (λ_ . 1 :: nat)"
```

The advantage of this generalization is that certain interactions with other operations can be proven for all functions at once. For example, updating an empty position in the work list is described by the lemma below for all those functions.

```
lemma (in ordered-cancel-comm-monoid-diff) work-list-fold-upd-None-to-Some:
  shows "i < length acc ⟹ acc ! i = None ⟹
           work-list-fold f (acc[i := Some t]) = work-list-fold f acc + f t"
```

The work list can be efficiently implemented by an array. The following assertion worklist-imp relates the functional work list to the imperative one in the following way: If the functional work list is empty at position i, the same has to hold for the imperative representative. Moreover, if this position is non-empty, the imperative tree must be a proper refinement of the functional one.

```
fun worklist-imp :: "_ rosetree worklist ⇒ _ fibtree-imp worklist ⇒ assn" where
  "worklist-imp [] [] = emp" |
  "worklist-imp (None#ts)    (None#tsi)    = worklist-imp ts tsi" |
  "worklist-imp (Some t#ts)  (Some ti#tsi) = fibtree-imp' t ti ⋆ worklist-imp ts tsi" |
  "worklist-imp _ _ = false"
```

Creating an empty work list is done functionally by calling the already defined replicate function that is of type nat ⇒ 'a ⇒ 'a list. The imperative equivalent is Array.new whose type has the same form nat ⇒ 'a ⇒ 'a array Heap. The operation worklist-empty wraps this function call:

```
definition "worklist-empty n = Array.new n None"
```

So, one can give a specialized rule and add it to the set of rules used by the verification condition generator:

```
lemma worklist-empty-rule:
  "<$(n + 1)>
    worklist-empty n
   <λa. ∃Axs. a ↦a xs ⋆ worklist-imp (replicate n None) xs ⋆ ↑(length xs = n)>"
```

Similarly, one defines an imperative function updating an array that refines a worklist:

```
definition "worklist-upd n a v = do {
    len ← Array.len a;
    if n ≥ len then return a else Array.upd n (Some v) a
  }"
```

Furthermore, one proves this rule below that is especially customized for the worklist assertion:

```
lemma worklist-upd-rule:
  "<a ↦a ys ⋆ worklist-imp xs ys ⋆ fibtree-imp t ti ⋆ $3>
    worklist-upd n a ti
   <λ_ . a ↦a ys[n := Some ti] ⋆
          worklist-imp (xs[n := Some t]) (ys[n := Some ti])>t"
```

Also, a simple swap operation on lists is implemented that can only exchange the i-th element with the list head. This operation is needed for a later operation.

```
definition swap :: "_ list × nat ⇒ _ list" where
  "swap v = (case v of (acc, i) ⇒
              if i < length acc then (acc[0 := acc ! i, i := acc ! 0]) else acc)"
```

For the purpose of an unconditional relation between the imperative and functional implementation of swap, it is asserted that i is smaller than the length of the list. This operation can be implemented very efficiently by arrays in contrast to functional lists:

```
definition swap-imp :: "'a :: heap array ⇒ nat ⇒ 'a array Heap" where
  "swap-imp p i = do {
    len ← Array.len p;
    if i < len then do {
      p0 ← Array.nth p 0;
      pi ← Array.nth p i;
      Array.upd 0 pi p;
      Array.upd i p0 p;
      return p
    } else return p
  }"
```

As one can see from the following Hoare triple, computation of swap-imp takes only six time-steps. Moreover, if the input array represents a work list, it is proven that the swapping on the imperative representative corresponds to swapping on the related functional list.

```
lemma swap-imp-rule-strong:
  "<p ↦a arr ⋆ worklist-imp acc arr ⋆ $6>
     swap-imp p i
   <λq. p ↦a swap (arr, i) ⋆ worklist-imp (swap (acc, i)) (swap (arr, i)) ⋆
        ↑(p = q)>t"
```

For the purpose of transforming a work list to a root list, one defines the function collapse. First, it exchanges the i-th tree with the one at the list head; the reason for this will be explained later. Second, all empty positions are removed, and third, the trees are extracted from the constructor Some that signalizes a non-empty position:

```
definition "collapse = (map the) o (filter ((≠) None)) o swap"
```

Since swap is already implemented imperatively, one just needs to define a function that iterates over the array to find the non-empty positions adding the contained trees to a circular doubly linked list:

```
partial-function (heap) wltl :: "_ cdll ⇒ _ array ⇒ nat ⇒ _ cdll Heap" where
  "wltl ts arr i = do {
    len ← Array.len arr;
    if i ≥ len then return ts
    else do {
      t ← Array.nth arr i;
      if (t = None) then
        wltl ts arr (i + 1)
      else do {
        ts ← cdll-snoc ts (the t);
        wltl ts arr (i + 1)
      }
    }
  }"
```

As one can prove by induction, each step of this iteration costs 20 time-credits and if correctly initialized, the work-list-to-list operation wltl computes the wanted result:

```
corollary wltl-rule:
  shows "<p ↦a arr ⋆ worklist-imp acc arr ⋆ $(2 + length acc · 20)>
           wltl None p 0
         <λ l'. cdll fibtree-imp (map the (filter ((≠) None) acc)) l'>t"
```

The function collapse-imp is plainly calling swap-imp and wltl consecutively. Thus, the following statement is proven almost automatically:

```
lemma collapse-imp-rule:
  shows "<p ↦a arr ⋆ worklist-imp acc arr ⋆ $(8 + length acc · 20)>
          collapse-imp (p, n)
         <λl' . cdll fibtree-imp (collapse (acc, n)) l'>t"
```

7.6.3 Join

Adding a tree t of rank i to the work list acc is performed by the operation join. It accesses the i-th cell of the work list, and if it contains a tree t', it is linked with t. The resulting tree is then added to the working list recursively. If the i-th cell is empty, tree t is plainly stored there. Then, the final rank is returned together with the updated work list.

The work list is of constant length. If the rank i is out of bounds of the work list, execution is aborted. Since the rank of a Fibonacci tree is bound logarithmically by the number of elements in the heap, one can calculate the maximal rank and initialize the work list appropriately. However, this complicates the proofs insomuch this condition has to be assumed throughout the whole verification process.

```
function join :: "_ rosetree ⇒ _ worklist ⇒ nat ⇒ _ worklist × nat" where
  "join t acc i = (let t' = acc ! i
                   in if i < length acc ∧ t' ≠ None then
                        join (link t (the t')) (acc[i := None]) (i + 1)
                      else
                        (acc[i := Some t], if i < length acc then i else i + 1))"

  by pat-completeness auto
```

No well-founded order on the input arguments of join is found automatically by the underlying procedure for the function definition. However, this is needed to prove termination. Nevertheless, one can declare such an order manually and then prove it to be well-founded:

```
termination join by (relation "measure (λ(_, acc, i). length acc - i)") auto
```

For reasoning about join, it is quite useful to distinguish three cases: i is out of bounds (case overflow); cell i is empty (case empty); and cell i is not empty. In the last case, join calls itself recursively (thus recursive case).

```
lemma join-cases [case-names over empty rec]:
  obtains "i ≥ length acc" |
          "i < length acc ∧ acc ! i = None" |
          "i < length acc ∧ acc ! i ≠ None"
```

To prove that all trees contained in the work list are Fibonacci trees after a call of join, one has to assume the invariant of the work list: At a non-empty position j, a Fibonacci tree is stored with rank j. Moreover, the input tree t must be a Fibonacci tree of rank i as well:

```
lemma join-fibtree:
  assumes "fibtree t" and "∀t∈set acc. t ≠ None ⟶ fibtree (the t)"
  assumes "∀j∈{0..<length acc}. acc ! j ≠ None ⟶ rank (the (acc ! j)) = j"
  assumes "rank t = i"
  shows "let (acc', i') = join t acc i in ∀t∈set acc'. t ≠ None ⟶ fibtree (the t)"
```

If the length of the work list is computed appropriately and assuming the work list invariant, it can be proven that no overflow will occur:

```
lemma join-bound:
  assumes "∀t∈set acc. t ≠ None ⟶ fibtree (the t)" and "fibtree t"
  assumes "∀i∈{0 ..< length acc}. (acc ! i) ≠ None ⟶ rank (the (acc ! i)) = i"
  assumes "rank t < length acc"
  assumes "work-list-size acc + rosetree-size t ≤ fib (length acc - 1)"
  assumes "2 < length acc"
  shows "let (acc', i') = join t acc (rank t) in i' < length acc"
```

In case that no the overflow arises, one can show that join does not discard any elements of the work list:

```
lemma join-mset: "let (acc', i') = join t acc i
                  in i' < length acc ⟶
                     mset-work-list acc' = mset-work-list acc + mset-tree t"
```

The imperative implementation join-imp follows the form of its functional counterpart. It is declared as a partial function to avoid the redundant termination proof which was already done above for join. Hence, a separate induction scheme is not necessary.

```
partial-function (heap) join-imp where
  "join-imp t acc i = do {
     len ← Array.len acc;
     (if i < len then do {
       t' ← Array.nth acc i;
       if t' ≠ None then do {
            t ← link-imp t (the t');
            Array.upd i None acc;
            join-imp t acc (i + 1)
          } else do {
            worklist-upd i acc t;
            return i
          }
     } else return (i + 1))
  }"
```

Assuming that the input tree ti is refining t and the array arr represents a lists that is related to the functional work list acc, one shows that join-imp is indeed correctly implementing join. Moreover, since Hoare triples in Imperative HOL with time are only valid for totally correct programs, the following lemma implies termination for join-imp under the aforementioned assumptions.

```
lemma join-imp-rule-aux:
  "<fibtree-imp t ti ⋆ p ↦a arr ⋆ worklist-imp acc arr ⋆
     $(30 + work-list-count acc · 20)>
     join-imp ti p i
   <λj'. let (acc', j) = (join t acc i)
         in (∃Aarr'. p ↦a arr' ⋆ worklist-imp' acc' arr' ⋆ ↑(j = j')) ⋆
            $((work-list-count acc') · 20)>t"
```

For each step, join-imp takes 20 time-credits. The number of steps can be at most the number of contained trees in the work list. This number is the difference between input i and resulting j plus the one for the inserted tree t:

```
lemma join-work-list-count:
  "let (acc', j) = join t acc i
   in work-list-count acc' = work-list-count acc + 1 + i - j"
```

As the postcondition of join-imp-rule reveals, the runtime analysis of join-imp is amortized. This is necessary to avoid overestimation of the worst-case runtime of consolidate.

In order to simplify the following runtime proofs that are necessary for consolidate-imp, one defines the combined assertion for work lists:

```
definition "worklisti acc arr = worklist-imp' acc arr ⋆ $((work-list-count acc) · 20)"
```

Using this, one can restate the prior rule more elegantly:

```
lemma join-imp-rule:
  "<fibtree-imp t ti ⋆ p ↦a arr ⋆ worklisti acc arr ⋆ $30>
     join-imp ti p i
   <λj'. let (acc', j) = (join t acc i)
         in (∃Aarr'. p ↦a arr' ⋆ worklisti acc' arr' ⋆ ↑(j = j'))>t"
```

7.6.4 Consolidate

The consolidate-rec operation folding the root list using join, which is wrapped in consolidate-rec-core:

```
lemma consolidate-rec-fold:
    show "consolidate-rec acc i ts = foldl consolidate-rec-core (acc, i) ts"
```

Nevertheless, consolidate-rec keeps track of the position of the tree with the smallest priority in the working list. Its position can change for two reasons:

First, the recently joined tree has a smaller value. Second, while joining the current tree with the smallest root value is linked and thus eventually stored as part of a tree of larger rank at higher position:

```
definition "consolidate-rec-core c t = (let
    (acc, i) = c;
    (acc', r') = join t acc (rank t);
    i' = if r' < length acc' ∧ i < length acc ∧ acc ! i ≠ None ∧ acc' ! r' ≠ None ∧
          val (the (acc' ! r')) ≤ val (the (acc ! i))
       then r' else i
    in (acc', i'))"
```

The condition acc ! i ≠ None ∧ acc' ! r' ≠ None in the above definition is only added to assure that the selector the :: 'a option ⇒ 'a is always executable. That is necessary to show unconditionally that the imperative version consolidate-rec-core refines this function.

Proving consolidate-rec correct, one has to assume that all trees in the work list as in the root list are Fibonacci trees (see assumption fibtree below); that all trees in work list have the correct rank according to their position (assumption rank); and that the length of the list is sufficient to avoid an overflow (assumption bound). To bundle all these assumptions, a context is set for the lemmata hereafter:

```
context
 fixes acc :: "'a :: linorder rosetree worklist" and ts :: "'a rosetree list"
 assumes bound: "work-list-size acc + heap-list-size ts ≤ fib (length acc - 1)"
                "2 < length acc"
 assumes fibtree: "∀t∈set ts. fibtree t"
                  "∀t'∈set acc. t' ≠ None ⟶ fibtree (the t')"
 and rank: "∀i∈{0 ..< length acc}. (acc ! i) ≠ None ⟶ rank (the (acc ! i)) = i"
```

First, we prove that these assumptions are indeed sufficient to prove that no overflow occurs:

```
lemma consolidate-rec-bound:
  assumes "i < length acc"
  shows "let (acc', i') = consolidate-rec acc i ts in i' < length acc"
```

This is needed to show that all elements will be joined into the work list while executing consolidate-rec:

```
lemma consolidate-rec-mset:
  "let (acc', i') = consolidate-rec acc i ts
   in mset-work-list acc' = mset-work-list acc + mset-heap-list ts"
```

Moreover, one can show using join-fibtree that also consolidate-rec preserves the invariant of all trees contained in the work list:

lemma consolidate-rec-fibtree:
"let (acc', i') = consolidate-rec acc i ts
in $\forall$t $\in$ set acc'. t $\neq$ None $\longrightarrow$ fibtree (the t)"

For a later analysis of the potential of the complete consolidate operation, one shows that the number of marked nodes will be smaller as before counting both trees of the root lists as well as the trees in the work list:

lemma consolidate-rec-marked:
"let (acc', i') = consolidate-rec acc i ts
in work-list-marked acc + ($\sum$t $\leftarrow$ ts. marked-num t) $\geq$ work-list-marked acc'"

Based on consolidate-rec, one defines consolidate' that initializes the work list with a sufficient length. Furthermore, this operation transfers the first tree of the root list to the work list to obtain an initial value for the position of the minimal element.

```
definition consolidate' :: "nat ⇒ _ rosetree list ⇒ _ rosetree worklist × nat" where
"consolidate' n ts' = (casets' f [] ⇒ ([], 0) | (t#ts) ⇒
                          let l   = ⌈ext-log ((1 + sqrt 5) / 2) n⌉ + 3;
                              acc = replicate l None
                          in consolidate-rec (acc[rank t := Some t]) (rank t) ts)"
```

To compute the length of the work list, the golden ration (1 + sqrt 5) / 2 is used as the base of the logarithm. The latter was wrapped to prevent invalid input arguments. If consolidate' is called on a list of Fibonacci heaps and n is the number of elements of this list, then it is ensured that n will always be greater zero.

definition "ext-log b n $\equiv$ (if n = 0 then 0 else log b n)"

Finally, one can define consolidate which generates at first the work list using consolidate' and then recreates the root list using the work list by calling collapse:

definition "consolidate n ts = Heap n (collapse (consolidate' n ts))"

Again, a proof context is used to compactly state the correctness lemmata. However this time, one only assumes that invar (Heap n ts) holds.

context
fixes l n :: nat **and** ts :: "'a :: linorder rosetree list"
defines "l $\equiv$ nat $\lceil$ext-log ((1 + sqrt 5) / 2) (real n)$\rceil$ + 3"
assumes "invar (Heap n ts)"

From this assumption, one can show that consolidate retains the invariant, i. e. the min-heap proposition as well as the Fibonacci heap property:

lemma consolidate-invar: "invar (consolidate n ts)"

Since no overflow will occur while joining the trees into the work list, one can prove that consolidate preserves all elements:

lemma consolidate-mset: "mset-heap (consolidate n ts) = mset-heap-list ts"

Again, for the later runtime analysis, one proves that the number of marked nodes might have decreased:

lemma consolidate-marked:
"($\sum$ t $\leftarrow$ nodes (consolidate n ts). marked-num t) $\leq$ ($\sum$ t $\leftarrow$ ts. marked-num t)"

The most important result is lemma consolidate-length. It states that the resulting root list will be logarithmically bound in length with respect to the number of contained elements. The number l is defined in the context above.

lemma consolidate-length: "length (nodes (consolidate n ts)) ≤ l"

Subsequently, the definitions of the imperative counterparts are again tightly following the functional implementations except for the next operation consolidate-rec-core-imp. It splits the executions paths after each evaluation of a part of the conditions, which leads to a branch diversification that does not appear in the functional version. This is for the reason that functional programs are composed out of expressions as opposed to imperative ones that chain operations linearly.

```
definition consolidate-rec-core-imp where
  "consolidate-rec-core-imp acc i t = do {
      let r = rank-imp t;
      len ← Array.len acc;
      if (i ≥ len) then do {
        join-imp t acc r;
        return i }
      else do {
        ti ← Array.nth acc i;
        r' ← join-imp t acc r;
        if (r' ≥ len) then return i
        else do {
            t ← Array.nth acc r';
            (if ti ≠ None ∧ t ≠ None ∧ value-imp (the t) ≤ value-imp (the ti)
             then return r' else return i)
        }
      }
}"
```

Nevertheless, consolidate-rec-core-imp can be proven correct. Even though its definition seems quite long, only an overhead of five time-steps are added to the costs of the wrapped join-imp:

```
lemma consolidate-rec-core-imp-aux:
  "<fibtree-imp' t ti ⋆ p ↦_a arr ⋆ worklist_i acc arr ⋆ $35>
    consolidate-rec-core-imp p i ti
  <λj. let (acc', i') = consolidate-rec-core (acc, i) t
       in ∃_A arr'. p ↦_a arr' ⋆ worklist_i acc' arr' ⋆ ↑(j = i')>_t"
```

Based upon this, one can define consolidate-rec-imp that iterates through the root list. For this, it calls the function cdll-fold that was introduced in the section about circular doubly linked lists.

```
definition consolidate-rec-imp where
 "consolidate-rec-imp acc i ts = cdll-fold (λi t. consolidate-rec-core-imp acc t i) i ts"
```

Inserting the loop invariant into the rule specific for cdll-fold, one can then show that the above operation takes 37 time-steps for each iteration plus an overhead:

```
lemma consolidate-rec-imp-rule:
  "<cdll fibtree-imp ts tsi ⋆ p ↦_a arr ⋆ worklist_i acc arr ⋆ $(37 + length ts · 37)>
    consolidate-rec-imp p tsi i
  <λj. let (acc', i') = consolidate-rec acc i ts
       in ∃_A arr'. p ↦_a arr' ⋆ worklist_i acc' arr' ⋆ ↑(j = i')>_t"
```

The following consolidate'-imp calls consolidate-rec-imp appropriately – as described already for the functional version.

```
definition consolidate'-imp where
  "consolidate'-imp n p = (case p of
     None ⇒ do {
        arr ← Array.new 0 None;
        return (arr, 0)
     }
   | Some p' ⇒ let l = nat ⌈ext-log ((1 + sqrt 5) / 2) n⌉ + 3 in do {
        arr ← worklist-empty l;
        r ← front-rank p';
        (t, p) ← cdll-pop-front p;
        worklist-upd r arr t;
        i ← consolidate-rec-imp arr p r;
        return (arr, i)
     })"
```

One can then show that consolidate'-imp in fact refines consolidate'. To run this operation, it takes a logarithmic amount of time-credits to initialize the array plus 37 credits for each tree in the root list as shown for consolidate-rec-imp already.

```
lemma consolidate'-imp-rule:
  fixes n :: nat
  defines "l ≡ nat ⌈ext-log ((1 + sqrt 5) / 2) n⌉ + 3"
  shows "<cdll fibtree-imp ts tsi ⋆ $(73 + l + length ts · 37)>
           consolidate'-imp n tsi
         <λ(p, j). let (acc', i) = consolidate' n ts
                   in ∃Aarr'. p ↦a arr' ⋆ worklisti acc' arr' ⋆ ↑(j = i)>t"
```

Finally, one defines consolidate-imp that encapsulates consolidate'-imp and collapse-imp:

```
definition "consolidate-imp n tsi = do {
  tsi ← consolidate'-imp n tsi ≫= collapse-imp;
  return (n, tsi)
}"
```

Ultimately, the rule below can be proven. Runtime costs are the sum of the ones for consolidate'-imp and collapse-imp as one naturally expects.

```
lemma consolidate-imp-rule:
  fixes n :: nat
  defines "l ≡ nat ⌈ext-log ((1 + sqrt 5) / 2) n⌉ + 3"
  shows "<cdll fibtree-imp ts tsi * $(81 + l * 21 + length ts * 37)>
           consolidate-imp n tsi
         <λ h. fibheap-imp (consolidate n ts) h>t"
```

7.6.5 Pop-Min

After all, one can define pop-min. It removes the root node of the first tree since this is the one with the minimal value. Afterwards, the list of children of this deleted node and the residual root list is joined and consolidated. Thereby, the number of elements is also decremented. If the heap contains no tree, no action is performed:

```
fun pop-min where
  "pop-min (Heap n (t#ts)) = consolidate (n - 1) (children t @ ts)" |
  "pop-min h = h"
```

Also for this operation, a proof context is opened. It is assumed that invariant holds for the input heap h and that this heap is not empty:

```
context
  fixes l n :: nat and ts :: "'a :: linorder rosetree list" and h
  defines "l ≡ nat ⌈ext-log ((1 + sqrt 5) / 2) (real n)⌉ + 3"
  defines : "h ≡ Heap n ts"
  assumes invar: "invar h" and non-empty: "ts ≠ []"
```

With these two assumptions, one shows that pop-min preserves the invariant.

```
lemma pop-min-invar: "invar (pop-min h)"
```

Moreover, exactly the element is removed that is extracted by get-min.

```
lemma pop-min-mset: "mset-heap (pop-min h) = mset-heap h - {# get-min h #}"
```

Most importantly, the potential is discharged so that an upper bound for it can be given that is logarithmic in the number of elements and linear in the number of marked nodes:

```
lemma pop-min-pot: "φ (pop-min h) ≤ l + sum-list (map marked-num ts)"
```

This result will be quite useful in the proof of the amortized runtime of imperative pop-min-imp, which is defined as follows:

```
definition "pop-min-imp h = (case h of (n, tsi) ⇒ do {   b ← cdll-is-empty tsi;
  if b then return h
  else do {
    (ti, tsi) ← cdll-pop-front tsi;
    tsi ← cdll-append (sub ti) tsi;
    consolidate-imp (n _ 1) tsi
  }
})"
```

Since pop-min-imp is not recursive, the rule is essential resembling the Hoare triple for consolidate. However, one thing complicates the proof significantly. Before calling consolidate-imp, the root list and the list containing the children of the deleted node are concatenated. This must be included in the accounting of the time-credits. Thus, the time assertion is rather long containing a constant overhead, a logarithmic amount for the initialization of the work list, and a number of credits for each tree in the root list that was a child of the deleted node. Since the root list could contain only singleton nodes, in worst case, the runtime of pop-min-imp is linear in the number of elements contained in the heap.

```
lemma pop-min-imp-rule:
  fixes h :: "_ roseheap"
  defines "l ≡ nat ⌈ext-log ((1 + sqrt 5) / 2) (count h - 1)⌉ + 3"
  shows "<fibheap-imp h hi ⋆
           $(105 + l · 21 + (degree (hd (nodes h)) + length (nodes h)) · 37)>
           pop-min-imp hi
         <λ hi'. fibheap-imp (pop-min h) hi'>t"
```

For an amortized analysis, one defines pop-min-time that takes into account the logarithmic expense to initialize the array, the costs resulting from processing the children and the constant overhead. Since the rank of a Fibonacci tree is bound logarithmically in the number of its elements, the time-credits used for the children are also bound logarithmically:

```
definition fib-log' :: "nat ⇒ nat" where
  "fib-log' n = nat ⌈ext-log ((1 + sqrt 5) / 2) n⌉ + 3"

definition pop-min-time :: "nat ⇒ nat" where
  "pop-min-time n = 105 + fib-log' n · 95"
```

Using this definition and the results concerning the potential, one can derive from the upper rule for pop-min-imp the following amortized one:

```
lemma pop-min-imp-rule-amo:
  fixes h :: "_ roseheap"
  assumes INVAR: "invar h" and NONEMPTY: "nodes h ≠ []"
  shows "<fibheap_i h hi ⋆ $(pop-min-time (count h))>
           pop-min-imp hi
         <fibheap_i (pop-min h)>t"
```

As explained, pop-min-time is asymptotically logarithmic:

```
lemma pop-min-time-in-log-n: "pop-min-time ∈ Θ(ln)"
```

Therefore, corollary pop-min-imp-rule-amo-alt can be derived from the rule above. Finally stating that pop-min-imp takes amortized only a logarithmic amount of time-steps:

```
corollary pop-min-imp-rule-amo-alt:
  fixes h :: "_ roseheap"
  assumes INVAR: "invar h" and NONEMPTY: "nodes h ≠ []"
  shows "∃f :: nat ⇒ nat. real o f ∈ Θ(ln)   ∧
         <fibheap_i h hi ⋆ $(f (count h))>
          pop-min-imp hi
         <fibheap_i (pop-min h)>t"
```

7.7 Interface Instantiation

Finally, one can instantiate the specification of priority queues of section 5 using functional Fibonacci heaps. This ensures on the one hand that no proofs regarding correctness are incomplete or insufficient. On the other hand, one can use the specification like an interface and implement an generic algorithm based on the abstractly declared operation. This algorithm can then subsequently be refined using Fibonacci heaps as a concrete instance.

interpretation fibonacci-heap: Priority-Queue-Merge
  where empty = empty **and** is-empty = is-empty
  and insert = insert **and** del-min = pop-min
  and get-min = get-min **and** merge = merge
  and invar = invar **and** mset = mset-heap

This approach is demonstrated in section 8 about heapsort where the resulting functional algorithm is further refined to an imperative algorithm.

7.8 Fibonacci Heaps with Separate Priority Keys

In the previous sections, Fibonacci heaps were described with no extra priority key. However, we additionally verified a version with such keys. This version can form the ground for possible further work. In order to define Fibonacci trees with separate priority keys, we have to attach to rose trees a new field for the priority:

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

Fibonacci heaps with separate priority keys are just slightly more complicated than the heaps without. In general, for any property concerning the order of the elements, the selector val is simply changed to prio as exemplified in the following definition:

fun min-tree :: "_ ⇒ bool" **where**
  "min-tree (Node p _ _ _ ts) $\longleftrightarrow$ ($\forall t \in$ set ts. min-tree t $\wedge$ p $\leq$ prio t)"

Moreover, some operations have to be adapted marginally to handle this additional priority appropriately for example:

fun insert :: "'a ⇒ 'b::linorder ⇒ ('a, 'b) heap ⇒ ('a, 'b) heap" **where**
  "insert v p h = merge (singleton v p) h"

The essential difference is the abstract interpretation of Fibonacci heaps with keys. They represent a multiset of pairs that consists of the value and its assigned priority:

```
fun mset-tree :: "('a, 'b :: linorder) tree ⇒ 'a × 'b multiset" where
  "mset-tree (Node p _ _ a c) = {# (a, p) #} + (∑ t ∈# mset c. mset-tree t)"
```

The theorem that is significantly influenced by this change is the correctness statement for get-min that has to retrieve the element with the lowest associated priority:

```
lemma get-min-correct:
  assumes INVAR: "invar (Heap n c)"
  assumes NONEMPTY: "mset-heap (Heap n c) ≠ {#}"
  shows "snd (get-min (Heap n c)) =
          Min-mset (image-mset snd (mset-heap (Heap n c)))"
```

The priority queue specification from section 5 is unaware of priority keys. For this reason, we identify values with priority keys to derive a rather unaesthetic instance. However, the aforementioned benefits of instantiation still apply.

```
interpretation fibonacci-heap: Priority-Queue-Merge
  where empty = empty and is-empty = is-empty
  and insert = "λv. insert v v" and del-min = pop-min
  and get-min = "fst ∘ get-min" and merge = merge
  and invar = "λh. invar h ∧ (∀(v, p) ∈ set-mset (mset-heap h). p = v)"
  and mset = "(image-mset fst) ∘ mset-heap"
```

Furthermore, all operations have been refined imperatively. However, the changes are insignificant. Thus, there are not illustrated here.

7.9 Decrease-Key

As previously mentioned, the operation decrease-key has not been formalized in this project. In the following, the reasons for this circumstance will be clarified. Nonetheless, the operation will be described first to complete the presentation of Fibonacci heaps.

7.9.1 Description

The operation decrease-key lowers the priority key of a node in the heap. Therefore, decrease-key is only sensible for Fibonacci heaps with separate priority keys as described in the previous section. In addition to this, each node must be attached by a second reference that points to its parent.

```
datatype 'a rosetree-imp' = Rose (cval: 'a) (sub: "'a rosetree-imp' cdll")
                                              (parent: "'a rosetree-imp' ref option)"
```

This is necessary to enable decrease-key to climb up the tree starting from any arbitrary node.

The operation takes three arguments: a references to the Fibonacci heap, another to the node that is to be updated and the new priority which must be less or equal to the one currently stored in the referenced node.

There are three cases to be considered while decreasing a priority key:

In the first case, the priority of the parent node is still less or equal than the decreased priority of its child. Thus, nothing beside updating the key has to be done.

The second case has been already mentioned in a prior section. In this case, the parent node is not marked. However, the min-heap property is violated. To restore this property, decrease-key cuts the node with lowered priority from its parent subsequently inserting it directly into the root list. Hence, the rank of the parent has to be decremented and therefore it must be marked in order to preserve the Fibonacci tree invariant. This is possible because this invariant allows a child to have rank one less than its position in the list of children if it is marked. Moreover, the potential mark of the cut node is removed since it is inserted into the root list and roots need not be marked.

The third case is like the second one but the parent node is already marked. If this applies, decreasing the rank of the parent without any further actions would break the Fibonacci tree invariant. For this reason, the parent is cut from its own parent node and inserted into the root list. The operation recursively climbs upwards the tree until an unmarked parent is found or the root of this tree. In the latter case, the rank can always be decremented since the Fibonacci heap invariant does not require, opposed to the binomial heap invariant, that the trees in the root list have to be of different rank.

The first and second case obviously take only constant time. However, in the last case, decrease-key might climb up the complete tree. However, its depth is logarithmically bound by the number of elements contained in the heap. So, in worst-case, the operation takes logarithmically many time-steps. Nevertheless, decrease-key only has to also cut a parent node if it is marked. For each marked node the potential has been increased by two, so by removing this mark by cutting it from its parent and inserting it into the root list, the potential is decreased in total by one for each marked node. This is because two units are freed by removing the mark but one is taken for the newly inserted tree in the root list. Thus, amortized, decrease-key takes only a constant amount of time-credits.

As already explained, Fibonacci heaps are therefore especially efficient in complex algorithms compared to other priority queue implementations.

7.9.2 Difficulties

Since there are no references in purely functional programs, this operation cannot be implemented in a functional fashion and therefore not directly in Isabelle/HOL either. This opposes to our approach to verify Fibonacci heaps by first implementing a functional version.

Furthermore, describing the relation of parent node and its child to a functional equivalent is considerably more complicated than for trees without a reference to its parent. This is for this reason that the partial heap containing the tree with the child as its root and the one containing the tree with the parent as its root do overlap. This opposes the key idea of separation logic to find disjoint heaps.

However, there are possibilities in separation logic to relate parts of a data structure to the their whole using for instance the separating implication denoted by ($\mathbin{-\!\!*}$). As for separating conjunction, the alias for this connective is inspired by its the notation: magic wand. $\mathsf{P} \mathbin{-\!\!*} \mathsf{Q}$ means if there is a heap described by assertion P, it is part of a heap that is described by assertion Q. However, this does not imply that either the former nor the latter heap exists. Existence of such heaps is formalized by $\mathsf{P} \star \mathsf{P} \mathbin{-\!\!*} \mathsf{Q}$. Therefore, an adapted assertion that relates the references to a parent node and its child to a functional Fibonacci tree could look like this:

```
function fibtree-imp where
  "fibtree-imp (Node p r m v ts) (Rose p (r', m', v') c' f)
    = (cdll fibtree-imp ts c' ⋆ ↑(r' = r ∧ m' = m ∧ v' = v)) ⋆
      (cdll fibtree-imp ts c' ⋆ ↑(r' = r ∧ m' = m ∧ v' = v)) —∗
       (∃A ft fp fr fm fv fts1 fts2. f ↦r ft ⋆
           fibtree-imp (Node fp fr fm fv (fts1@[(Node p r m v ts)]@fts2)) ft)"
```

Besides hardly proving termination for this kind of assertion, we lack in general experience with the separating implication. Moreover, this connective has not even been defined in Imperative HOL with time. Hence, no proof automation is provided by this framework for using the magic wand. By industriousness, this issues could have been partially eliminated.

Furthermore, there exits an additional problem that is even more intricate to solve. Consider the cdll-snoc-rule from the section about doubly linked lists:

```
"<R x x' ⋆ cdll R xs p ⋆ $16> cdll-snoc p x' <cdll R (xs@[x])>t"
```

This Hoare triple above only states that the result of the operation cdll-snoc relates to the outcome of the functional snoc. However, it does not describe how the imperative result is related to its input p. Moreover, a relation between arbitrary references into a data structure and the resulting one after applying an operation is actually required.

For the cdll-snoc-rule this means that the pre- and post-condition of the Hoare triple have to be strengthened, s. t. a node that is pointed to by an arbitrary reference must also be part of the resulting list. To us, it is unclear how to formalize this appropriately in separation logic.

All in all, there are to many obstructions in order to include decrease-key in this formalization in the given scope. Thus, the residual time has been spent by improving the already existing theories and by adding the following use case.

8 Heapsort as an Use Case of Fibonacci Heaps

Heapsort is one of the applications of heaps. It was invented by Floyd [36]. He called it tree sort. The algorithm is simple: First, all elements of a list are inserted into a heap. Second, all of them are popped out of the heap again and inserted in a list. Since a popped element is smaller than the remaining ones into the heap, the resulting list will be sorted.

This is an interesting case study for applying Fibonacci heaps insofar that the amortized runtime analysis is necessary to prove optimality of the algorithm. The worst-case runtime of pop-min of Fibonacci heaps is in $\mathcal{O}(\mathsf{n})$, hence overall runtime of heapsort with Fibonacci heaps could only be proven to be in $\mathcal{O}(\mathsf{n}^2)$, which is suboptimal. In contrast to this, an amortized runtime analysis gives $\mathcal{O}(\log \mathsf{n})$ and therefore $\mathcal{O}(\mathsf{n} \log \mathsf{n})$ for heapsort.

At first, a functional version will be defined and proven correct, then this will be refined to an imperative program which is finally used to analyse the runtime.

8.1 To-Heap

As aforementioned, the functional version of the algorithm is implemented abstractly using the interface Priority-Queue.

The function to-heap inserts all elements into heap by folding the list:

definition "to-heap = foldl (λq x. insert x q) empty"

Following from the specification of empty and insert, one can prove that no element is lost and the final heap fulfils its invariant:

corollary to-heap-mset: "Multiset.mset xs = mset (to-heap xs)"

corollary to-heap-invar: "invar (to-heap xs)"

The imperative version is based on a tail-recursive function:

```
fun to-heap-imp-rec :: " _ list ⇒ _ ⇒ _ Heap" where
  "to-heap-imp-rec [] hi = ureturn hi" | "to-heap-imp-rec (x # xs) hi = do {
    hi ← insert-imp x hi;
    to-heap-imp-rec xs hi
  }"
```

This function is then called with the initial accumulator, which is in this case an empty heap:

```
definition "to-heap-imp xs = empty-imp ≫= to-heap-imp-rec xs"
```

First, one shows by induction that to-heap-imp-rec is a correct refinement, then one can easily derive the following rule:

```
lemma to-heap-imp-rule:
  "<$(1 + 22 · length xs)>
     to-heap-imp xs
   <λhi . fibheap-imp (fibonacci-heap.to-heap xs) hi>_t"
```

For an amortized analysis, one has to obtain that the root list is maximally degenerated, i. e. each element is stored in a singleton tree. Hence, the following fact can be derived:

```
corollary to-heap-pot: "φ (fibonacci-heap.to-heap xs) = length xs"
```

Using this, one can see that to-heap-imp is linear in the number of elements. However, a significant amount of time-credits have to be stored to charge the potential:

```
lemma to-heap-imp-rule-alt:
  "<$(1 + 59 · length xs)>
    to-heap-imp xs
   <λhi . fibheap_i (fibonacci-heap.to-heap xs) hi>_t"
```

8.2 Pop-All

The operation pop-all is easily defined albeit its termination cannot be proven unconditionally. Thus, a separate proof is required for this, and subsequently, proving properties of this function is more complex than usual.

```
function (domintros) pop-all where
    "pop-all xs q = (if is-empty q then xs
                     else pop-all (get-min q # xs) (del-min q))"
  by pat-completeness auto
```

Only when the queue q fulfils the demanded invariant, one can show that pop-all will terminate:

```
lemma pop-all-termination: "invar q ⟹ pop-all-dom (xs, q)"
```

Using this, one can show that pop-all does discard any elements and the resulting list is indeed sorted in descending order:

```
corollary mset-pop-all: "invar q ⟹ Multiset.mset (pop-all [] q) = mset q"
```

```
lemma pop-all-sorted': "invar q ⟹ sorted (rev (pop-all [] q))"
```

The imperative refinement is analogous to the functional version. It retrieves the next minimal element of the heap as long it is not empty.

```
partial-function (heap) pop-all-imp where
  "pop-all-imp xs hi = do {
   b ← is-empty-imp hi;
   if b then return xs
   else do {
     x ← get-min-imp hi;
     hi ← pop-min-imp hi;
     pop-all-imp (x#xs) hi
    }
  }"
```

By using lemma pop-min-imp-rule-amo from the previous section, we can prove by induction that pop-all needs for each element in h at most pop-min-time (count h) + 3 time-credits, which is as already shown a logarithmic amount:

```
lemma pop-all-imp-rule:
  assumes "invar h"
  shows "<fibheap_i h hi ⋆
            $(3 + heap-list-size (nodes h) · (pop-min-time (count h) + 3))>
            pop-all-imp xs hi
          <λxs'. ↑(fibonacci-heap.pop-all xs h = xs')>_t"
```

8.3 Heap-Sort

Combining these functions is easy:

```
definition "heap-sort xs = pop-all [] (to-heap xs)"
```

Therefore, the proofs for correctness of heap-sort are fully automatic:

```
lemma heap-sort-mset: "Multiset.mset (merge-sort xs) = Multiset.mset xs"

lemma heap-sort-sorts: "sorted-wrt (≥) (merge-sort xs)"
```

The definition of heap-sort-imp follows the form of heap-sort:

```
definition "heap-sort-imp xs = do {
  hi ← to-heap-imp xs;
  pop-all-imp [] hi
}"
```

From this, the corresponding rule follows directly:

```
lemma heap-sort-imp-rule:
  "<$(4 + length xs · (pop-min-time (length xs) + 62))>
     heap-sort-imp xs
   <λxs' . ↑(fibonacci-heap.heap-sort xs = xs')>_t"
```

Finally, we can prove from the above lemma that heap-sort-imp has an optimal runtime for comparison based sorting [3]:

```
lemma heap-sort-imp-rule-alt:
  "∃f :: nat ⇒ nat. f ∈ Θ(λn. n · ln n) ∧
      <$(f (length xs))>
        heap-sort-imp xs
      <λxs'. ↑(fibonacci-heap.heap-sort xs = xs')>_t"
```

Even though, an amortized runtime analysis was used in between, this final result is the worst case behaviour as one can see from the post-condition, which contains neither time assertions nor assertions about any data structure adjunct with a potential.

9 Further Work

As explained in the section about decrease-key, this operation is unfortunately not part of our formalization. However, this would be definitely desirably for its importance, e. g. in greedy algorithms.

Nevertheless, the described obstacles have to be overcome first. Using a simpler data structure, one has to find a good formalization of the earlier characterized issue of continued validity of references pointing into data structures after applying operations. Furthermore, this may imply that some work has to be done to extent the capabilities of Imperative HOL (with time) and perhaps the accompanying tool chain. Based thereon, the refinement lemmata then can be strengthened culminating hopefully in the verification of decrease-key.

Eventually, when decrease-key is formalized, one can join Fibonacci heaps into the Imperative Isabelle Collection Framework (IICF) which would increase the accessibility of the this work substantially. Furthermore, this would finally conclude the work on this data structure.

As a first step, to bridge the time until complete verification, we will publish the theories produced for this thesis in the archive of formal proofs. As mentioned, to the best of our knowledge, this is still the first time that Fibonacci heaps have been verified complete for the functions empty, singleton, merge, insert, consolidate and pop-min including functional correctness, correct imperative refinement and amortized runtime.

10 References

[1] Tobias Nipkow and Gerwin Klein. “Concrete Semantics”. In: *A Proof Assistant Approach* (2014).

[2] Michael L Fredman and Robert Endre Tarjan. “Fibonacci heaps and their uses in improved network optimization algorithms”. In: *Journal of the ACM (JACM)* 34.3 (1987), pp. 596–615.

[3] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844, 9780262033848.

[4] John C Reynolds. “Separation logic: A logic for shared mutable data structures”. In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE. 2002, pp. 55–74.

[5] Peter Lammich and Rene Meis. “A separation logic framework for Imperative HOL”. In: *Archive of Formal Proofs* 2012 (2012).

[6] Lukas Bulwahn et al. “Imperative Functional Programming with Isabelle/HOL”. In: *Theorem Proving in Higher Order Logics*. Ed. by Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 134–149.

[7] Peter Lammich and Andreas Lochbihler. “The Isabelle collections framework”. In: *International Conference on Interactive Theorem Proving*. Springer. 2010, pp. 339–354.

[8] Robert Endre Tarjan. “Amortized computational complexity”. In: *SIAM Journal on Algebraic Discrete Methods* 6.2 (1985), pp. 306–318.

[9] Robert Atkey. “Amortised resource analysis with separation logic”. In: *European Symposium on Programming*. Springer. 2010, pp. 85–103.

[10] Bohua Zhan and Maximilian PL Haslbeck. “Verifying asymptotic time complexity of imperative programs in Isabelle”. In: *International Joint Conference on Automated Reasoning*. Springer. 2018, pp. 532–548.

[11] Lawrence C Paulson. “The foundation of a generic theorem prover”. In: *Journal of Automated Reasoning* 5.3 (1989), pp. 363–397.

[12] Alonzo Church. “A formulation of the simple theory of types”. In: *The journal of symbolic logic* 5.2 (1940), pp. 56–68.

[13] Haskell B Curry. “Functionality in combinatory logic”. In: *Proceedings of the National Academy of Sciences of the United States of America* 20.11 (1934), p. 584.

[14] William A Howard. “The formulae-as-types notion of construction”. In: *To HB Curry: essays on combinatory logic, lambda calculus and formalism* 44 (1980), pp. 479–490.

[15] Lawrence C Paulson. *Isabelle: A generic theorem prover*. Vol. 828. Springer Science & Business Media, 1994.

[16] Peter B Andrews. *An introduction to mathematical logic and type theory*. Vol. 27. Springer Science & Business Media, 2002.

[17] Jasmin Christian Blanchette et al. “Truly modular (co) datatypes for Isabelle/HOL”. In: *International Conference on Interactive Theorem Proving*. Springer. 2014, pp. 93–110.

[18] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*. Vol. 2283. Springer Science & Business Media, 2002.

[19] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge university press, 1999.

[20] Markus M Wenzel. “Isabelle/Isar—a versatile environment for human-readable formal proof documents”. PhD thesis. Technische Universität München, 2002.

[21] Lawrence C Paulson. “A generic tableau prover and its integration with Isabelle”. In: *Journal of Universal Computer Science* 5.3 (1999), pp. 73–87.

[22] The University of Glasgow. *Control.Monad.ST*. Last accessed 20 March 2019. 2019. URL: `http://hackage.haskell.org/package/base-4.12.0.0/docs/Control-Monad-ST.html`.

[23] Google LLC. *Google Scholar — Separation logic: A logic for shared mutable data structures*. Last accessed 20 March 2019. URL: `https://scholar.google.com/scholar?cites=11032510923081803901&as_sdt=2005&sciodt=0,5`.

[24] Charles Antony Richard Hoare. “An axiomatic basis for computer programming”. In: *Communications of the ACM* 12.10 (1969), pp. 576–580.

[25] Edsger W Dijkstra. “A note on two problems in connexion with graphs”. In: *Numerische mathematik* 1.1 (1959), pp. 269–271.

[26] Robert Clay Prim. “Shortest connection networks and some generalizations”. In: *The Bell System Technical Journal* 36.6 (1957), pp. 1389–1401.

[27] Authors of the cppreference wiki. *std::priority_queue — cppreference.com*. Last accessed 28 March 2019. URL: `https://en.cppreference.com/w/cpp/container/priority_queue`.

[28] Inc. Oracle America. *PriorityQueue (Java Platform SE 8)*. Last accessed 28 March 2019. URL: `https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html`.

[29] Python Software Foundation. *heapq — Heap queue algorithm*. Last accessed 28 March 2019. URL: `https://docs.python.org/3/library/heapq.html`.

[30] Gerth Stølting Brodal. “Worst-Case Efficient Priority Queues.” In: *SODA*. Vol. 96. 1996, pp. 52–58.

[31] Eric W. Weisstein. *“Forest” From MathWorld – A Wolfram Web Resource*. Last accessed 28 March 2019. URL: `http://mathworld.wolfram.com/Forest.html`.

[32] Richard Bird et al. *Introduction to functional programming using Haskell.* Vol. 2. Prentice Hall Europe Hemel Hempstead, UK, 1998.

[33] Jean Vuillemin. "A data structure for manipulating priority queues". In: *Communications of the ACM* 21.4 (1978), pp. 309–315.

[34] Chris Okasaki. *Purely functional data structures.* Cambridge University Press, 1999.

[35] Parmanand Singh. "The so-called fibonacci numbers in ancient and medieval India". In: *Historia Mathematica* 12.3 (1985), pp. 229 –244. ISSN: 0315-0860. DOI: `https://doi.org/10.1016/0315-0860(85)90021-7`. URL: `http://www.sciencedirect.com/science/article/pii/0315086085900217`.

[36] Robert W Floyd. "Algorithm 245: treesort". In: *Communications of the ACM* 7.12 (1964), p. 701.

[37] Daniel Stüwe. *GitLab repository containing the Fibonacci heap verification.* URL: `https://gitlab.lrz.de/stuewe/fibonacci.git`.

11 Attachment

LoP	Theory File	Content
400	Basics.thy	rose trees, Fibonacci heaps and sanity functions
250	Circular_Doubly_Linked_List.thy	circular doubly linked lists and their operations using Imperative HOL
300	Circular_Doubly_Linked_List_With_Time.thy	circular doubly linked lists and operations using Imperative HOL with time
250	Doubly_Linked_List.thy	circular doubly linked lists segments and list fold using Imperative HOL
300	Doubly_Linked_List_With_Time.thy	circular doubly linked lists segments and list fold using Imperative HOL with time
400	FibheapSort.thy	functional heap sort, subsequent imperative refinement using Fibonacci Heaps (with time)
1500	FibonacciHeaps.thy	functional Fibonacci heaps and their operations
350	Imperative_Basics.thy	general relation between functional and imperative Fibonacci heaps
350	Imperative_Basics_With_Time.thy	as above
550	Imperative_FibonacciHeaps.thy	imperative operations and refinement proofs using Imperative HOL
850	Imperative_FibonacciHeaps_With_Time.thy	imperative operations and refinement proofs using Imperative HOL with time
75	More_Fib.thy	additional facts about Fibonacci numbers
175	Time_Basics.thy	missing foundational facts about separation logic using Imperative HOL with time
250	Work_List.thy	functional worklists and their operations (used in pop-min)
$\sum$ 6000		all theories for Fibonacci heaps without separate priority keys

Table 2: List of all proof documents for Fibonacci heaps without separate priority keys, Lines of Proof (LoP) is rounded down

The proof documents listed in table 2 are hosted online in a git-repository by the LRZ [37] including the version of Fibonacci heaps that uses separate priority keys.